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: «Разработка онлайн-игры»

Выполнил студент 1 курса гр. ИС-34 Шульпина П.О.

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

Содержание

[Введение 3](#_Toc133767646)

[Задачи 3](#_Toc133767647)

[1.Теоретическая часть 4](#_Toc133767648)

[Графы 4](#_Toc133767649)

[Обход графа 4](#_Toc133767650)

[Жадный алгоритм 5](#_Toc133767651)

[Для чего используются жадные алгоритмы? 6](#_Toc133767652)

[Алгоритм А\* 6](#_Toc133767653)

[Приоритетная очередь 6](#_Toc133767654)

[Реализация очереди приоритетов на Python 7](#_Toc133767655)

[2. Реализация алгоритма 8](#_Toc133767656)

[Пример работы 9](#_Toc133767657)

[Заключение 10](#_Toc133767658)

[Список литературы 11](#_Toc133767659)

[Приложение 1 12](#_Toc133767660)

[Листинг программы 12](#_Toc133767661)

# Введение

Алгоритмы обхода графа являются одной из важнейших задач в программировании. Возможность формального моделирования множества различных реальных систем посредством графов позволяет программисту решать широкий круг прикладных задач (например, построение маршрута). Важно учитывать, что все основные служебные операции по работе с графами (такие как распечатка или копирование графов или преобразования графа из одного представления в другое) являются приложениями обхода графа.[1]

В последнее время задача обхода ориентированных графов стала особенно актуальной в связи с тестированием конечных автоматов, точнее, объектов, рассматриваемых как конечные автоматы.

Примерами алгоритмов обхода графа являются поиск в глубину, поиск в ширину и жадный алгоритм.

Использование жадного алгоритма помогает решить задачи, принимая последовательность близоруких и необратимых решений. Преимуществом этого алгоритма является более легкая разработка и скорость работы, поэтому важным навыком является умение его грамотно использовать и внедрять.[8]

**Цель работы**: реализовать алгоритмы обхода графа: жадный алгоритм 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: текстовый файл с лабиринтом, координаты точек входа и выхода, ключевая точка;
* Реализовать жадный алгоритм и А\* с заданными параметрами;
* Сохранить результаты обходов лабиринта и получившиеся маршруты в отдельный файл “maze-for-me.txt”.

# 1.Теоретическая часть

# Графы

Графом называется упорядоченная пара множеств G = (V, E). где V — подмножество вершин (или узлов), а E = Fx, V - упорядоченное или неупорядоченное подмножество ребер, соединяющих пары вершин из V. В моделировании дорожной сети вершины представляют населенные пункты (или пересечения), определенные пары которых соединены дорогами (ребрами). В анализе исходного кода компьютерной программы вершины могут представлять операторы языка, а ребра будут соединять операторы х в у, если у выполняется после х.

Графы – это абстракция, благодаря которой можно описывать различные существующие системы: транспортные, навигационные, спутниковые и т.д..[7]

# Обход графа

Обход графа — это переход от одной его вершины к другой в поисках свойств связей этих вершин. Связи (линии, соединяющие вершины) называются направлениями, путями, гранями или ребрами графа. Вершины графа также именуются узлами.[4]

Лабиринты обычно представляются в виде графов, где вершины обозначают пересечения путей, а ребра — пути лабиринта. Таким образом, любой алгоритм обхода графа должен быть достаточно мощным, чтобы вывести нас из произвольного лабиринта.

Чтобы обеспечить эффективность такого алгоритма, мы должны гарантировать, что не будем постоянно возвращаться в одну и ту же точку, оставаясь в лабиринте навечно.

А для правильности нашего алгоритма нам нужно выполнять обход лабиринта системным образом, который гарантирует, что мы выберемся из лабиринта. В поисках выхода нам нужно посетить каждую вершину и каждое ребро графа.

Ключевая идея обхода графа - пометить каждую вершину при первом ее посещении и помнить о том, что не было исследовано полностью.

Каждая вершина будет находиться в одном из следующих трех состояний:

• закрытая (undiscovered) - первоначальное, нетронутое состояние вершины;

• открытая (discovered) - вершина обнаружена, но мы еще не проверили все инцидентные ей ребра;

обработанная (processed) - все инцидентные данной вершине ребра были посещены

Очевидно, что вершину нельзя обработать до того, как она открыта, поэтому в процессе обхода графа состояние каждой вершины начинается с неоткрытого, переходит в открытое и заканчивается обработанным.

Нам также нужно иметь структуру, содержащую все открытые, но еще не обработанные вершины. Первоначально открытой считается только одна вершина — начало обхода графа. Для полного исследования вершины и нужно изучить каждое исходящее из нее ребро. Если какие-либо ребра идут к неоткрытой вершине х, то эта вершина помечается как открытая и добавляется в список для дальнейшей обработки. Ребра, идущие к обработанным вершинам, игнорируются, т. к. их дальнейшее исследование не сообщит нам ничего нового о графе. Также можно игнорировать любое ребро, идущее к открытой, но не обработанной вершине, т. к. эта вершина уже внесена в список вершин, подлежащих обработке.

Каждое неориентированное ребро рассматривается дважды, по одному разу при исследовании каждой из его вершин. Ориентированные ребра рассматриваются только один раз, при исследовании его источника. В конечном счете. все ребра и вершины в связном графе должны быть посещены.[7]

# Жадный алгоритм

**Жадные алгоритмы** нацелены на то, чтобы сделать оптимальный выбор в данный момент. На каждом шаге он выбирает оптимальный вариант, не зная будущего.

Жадные алгоритмы касаются только локального оптимального решения. Это означает, что **общее оптимальное решение может отличаться от решения, выбранного алгоритмом.**

Они никогда не оглядываются назад на то, что они сделали, чтобы увидеть, могли ли они оптимизировать глобально. В этом основное различие между жадным и динамическим программированием.

## Для чего используются жадные алгоритмы?

Жадные алгоритмы работают намного быстрее, чем две другие альтернативы (Разделяй и властвуй и динамическое программирование).

Иногда жадные алгоритмы каждый раз выдают глобальное оптимальное решение. Некоторые из этих алгоритмов являются:

* Алгоритм Дейкстры
* Алгоритм Крускала
* Алгоритм Прима
* Деревья Хаффманах[3]

# Алгоритм А\*

Алгоритм А\* находит оптимальный вариант благодаря вычислению суммарной стоимости всех путей между начальной и конечной точкой. Кстати, этот способ быстрее алгоритма Дейкстры благодаря эвристической функции.

g (n) : фактический путь затрат от начального узла до текущего узла.

h ( n) : фактический путь затрат от текущего узла до целевого узла.

f (n) : фактический путь затрат от начального узла до целевого узла.[6]

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.

Для реализации алгоритма A \* мы должны использовать два массива. Первый содержит узлы, которые были сгенерированы, но еще не были проверены до сих пор. Второй содержит проверяемые узлы.[1]

# Приоритетная очередь

Приоритетная очередь в python - это расширенный тип структуры данных очереди. Вместо удаления из очереди самого старого элемента приоритетная очередь сортирует и удаляет из очереди элементы на основе их приоритетов. Приоритетная очередь обычно используется для решения проблем с планированием. Это дает приоритет задачам с более высокой срочностью.

## Реализация очереди приоритетов на Python

Чтобы **реализовать приоритетную очередь в Python**, мы должны объявить пустой список Python, в который вставляются элементы с помощью **метода** класса list, в который вставляются элементы. Затем список сортируется в порядке возрастания. Цикл While используется для извлечения элементов с использованием метода **pop().**[5]

# 2. Реализация алгоритма

Сначала происходит импорт библиотек, необходимых для работы алгоритмов.

Исходный файл лабиринта считывается и преобразуется в список, вводятся координаты входа и выхода, а также ключевая точка. Список затем передается в функцию поиска пути. Функция создает новый объект, инициализирует его атрибуты и возвращает ссылку на новый объект.[2] Для этого используется две реализации жадного алгоритма и алгоритма A\*: первая - функция "greedy\_algorithm", вторая - функция "astar". Первая реализация является более простой и менее точной по сравнению со второй, но более быстрой.

"Greedy\_algorithm" принимает лабиринт, координаты начальной и конечной точек, и находит путь от стартовой до конечной точки. Для этого используется эвристическая функция, которая вычисляет расстояние между текущей точкой и конечной. Она добавляет координаты точек в приоритетную очередь в соответствии с их расстоянием до конечной точки и посещает каждую координату, чтобы найти путь до конечной точки. Если путь не найден, функция возвращает None.

"astar" реализует алгоритм А\*, который также находит путь от начальной до конечной точки, но использует g-издержки, чтобы учитывать стоимость перемещения между точками.

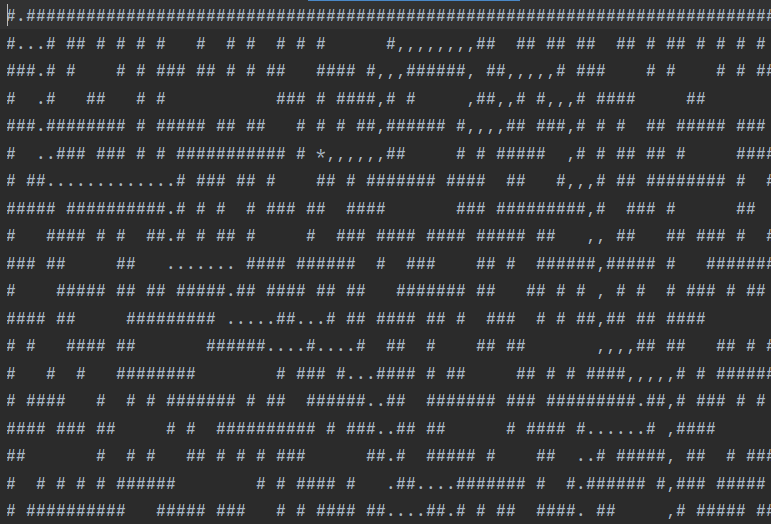
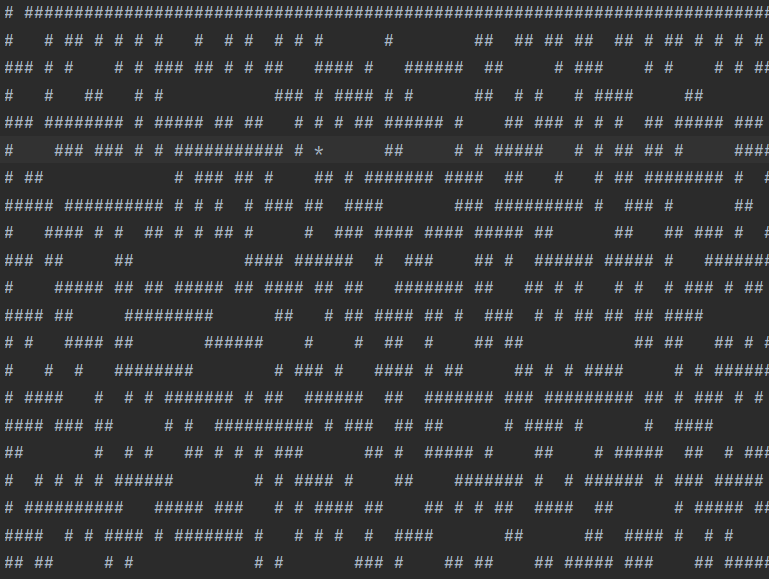
Вспомогательные функции - "heuristic" для вычисления эвристической функции, и "construct\_path" для построения пути от начальной точки до конечной на основе предыдущих шагов.

Далее в программе вызываются определенные функции с помощью переданных аргументов, чтобы найти путь в лабиринте и пометить его на карте.

Измененная карта записывается в файл с помощью функции "final\_file".

В результате работы программы создается файл, который содержит карту лабиринта с помеченными путями от стартовой точки до ключа (.) и от ключа до выхода (,).

# Пример работы



# Заключение

В ходе проделанной работы был создан алгоритм построения маршрута в графе, реализованный с помощью жадного алгоритма и алгоритма А\*. Разработка программы потребовала изучения таких аспектов, как: работа с графами, алгоритмы обхода графов, приоритетная очередь.

В результате была разработана программа поиска пути в лабиринте от начальной точки до конечной на языке программирования Python. С помощью жадного алгоритма и А\* был создан файл, который содержит карту лабиринта с помеченными путями от стартовой точки до ключа (.) и от ключа до выхода (,).

# Список литературы

1. Алгоритм А\* и его реализация на Python [Электронный ресурс]. URL: <https://pythonist.ru/algoritm-a-star-i-ego-realizacziya-na-python/> (Дата обращения: 30.04.2023).
2. Аллен Б. Дауни, основы Python [Электронный ресурс]. URL: <https://library.samdu.uz/files/b0c333b5613b7c1710d62e0194dfc40e_Дауни%20А.%20-%20Основы%20Python%20-%202021.pdf> (Дата обращения: 30.04.2023).
3. Жадные алгоритмы в Python [Электронный ресурс]. URL:<https://skerritt.blog/greedy-algorithms/> (Дата обращения: 30.04.2023).
4. Обход графа: поиск в глубину и поиск в ширину простыми словами на примере JavaScript [Электронный ресурс]. URL: <https://habr.com/ru/articles/504374/> (Дата обращения: 30.04.2023).
5. Очередь приоритетов в Python [Электронный ресурс]. URL:<https://pythonguides.com/priority-queue-in-python/>(Дата обращения: 30.04.2023).
6. Руководство для инсайдеров по алгоритму A\* в Python [Электронный ресурс]. URL: <https://www.pythonpool.com/a-star-algorithm-python/> (Дата обращения: 30.04.2023).
7. Стивен Скиена, Алгоритмы. Руководство по разработке [Электронный ресурс]. URL: <https://djvu.online/file/C4YRblGTe2IRh> (Дата обращения: 30.04.2023).
8. Тим Рафгарден, «Совершенный алгоритм. Жадные алгоритмы и динамическое программирование» [Электронный ресурс]. URL: <https://дз.ею/tmp/Bibl_progr_Sb_187kn/Совершенный%20алгоритм.%20Жадные%20алгоритмы%20и%20динамическое%20программирование.%202020.pdf> (Дата обращения: 30.04.2023).

# 

# Приложение 1

## Листинг программы

import heapq  
# импорт и преобразование исходного файла  
filename = "maze-for-u.txt"  
maze = []  
  
with open(filename, 'r') as f:  
 for line in f:  
 maze.append(list(line.strip()))  
  
# Функция определения начальной точки  
def find\_start\_point(lst):  
 first\_line = lst[0]  
 for i in range(len(first\_line)):  
 if first\_line[i] == ' ':  
 start=(0,i)  
 return start  
 return -1  
start = find\_start\_point(maze)  
  
# Функция нахождения точки выхода  
def end\_point(lst):  
 first\_line = maze[len(maze)-1]  
 for i in range(len(first\_line)):  
 if first\_line[i] == ' ':  
 start=(len(maze)-1,i)  
 return start  
 return -1  
end = end\_point(maze)  
  
  
  
# определение ключевой точки  
  
for x in range(len(maze)):  
 for y in range(len(maze[0])):  
 if maze[x][y] == "\*":  
 key = (x, y)  
 break  
  
# Определяем функцию для поиска выхода из лабиринта.  
def greedy\_algorithm(maze, start, goal):  
 # Создаем приоритетную очередь  
 queue = []  
 heapq.heappush(queue, (heuristic(start, goal), start)) #начинаем с координат старта  
 # Создаем список посещенных координат  
 visited = set()  
 # Создаем словарь для хранения предыдущих шагов  
 came\_from = {}  
  
 while queue:  
 # Получаем текущую координату  
 current = heapq.heappop(queue)[1]  
  
 # Если текущая координата соответствует цели, мы нашли выход из лабиринта  
 if current == goal:  
 return construct\_path(came\_from, start, goal)  
  
 # Добавляем текущую координату в список посещенных  
 visited.add(current)  
  
 # Получаем список доступных координат из текущего местоположения  
 possible\_moves = get\_possible\_moves(maze, current)  
  
 # Добавляем координаты в очередь в соответствии со значением эвристической функции  
 for move in possible\_moves:  
 if move not in visited:  
 heapq.heappush(queue, (heuristic(move, goal), move))  
 came\_from[move] = current  
  
 # Если выход из лабиринта не найден  
 return None  
  
def get\_possible\_moves(maze, current):  
 # Получаем размеры лабиринта  
 rows, cols = len(maze), len(maze[0])  
 moves = []  
  
 # Проверяем возможные шаги в каждом направлении  
 for dx, dy in ((0, 1), (1, 0), (0, -1), (-1, 0)):  
 row, col = current[0] + dx, current[1] + dy  
 # Проверяем, что координата находится в лабиринте и доступна  
 if 0 <= row < rows and 0 <= col < cols and maze[row][col] != "#":  
 moves.append((row, col))  
  
 return moves  
  
def heuristic(current, goal):  
 # Вычисляем Евклидово расстояние от текущей координаты до целевой координаты  
 return ((current[0] - goal[0]) \*\* 2 + (current[1] - goal[1]) \*\* 2) \*\* 0.5  
  
def construct\_path(came\_from, start, goal):  
 current = goal  
 path = [current]  
  
 while current != start:  
 current = came\_from[current]  
 path.append(current)  
  
 # Переворачиваем путь, так как мы добавляли координаты в обратном порядке  
 path.reverse()  
  
 return path  
  
  
def heuristic(a, b):  
 *"""  
 Реализация эвристической функции для A\*  
 """* return abs(b[0] - a[0]) + abs(b[1] - a[1])  
  
def astar(array, start, goal):  
 *"""  
 Реализация алгоритма A\*  
 """* # Инициализируем очередь, очередь приоритетов и список посещенных узлов  
 neighbors = [(0,1),(0,-1),(1,0),(-1,0)]  
 close\_set = set()  
 came\_from = {}  
 gscore = {start:0}  
 fscore = {start:heuristic(start, goal)}  
 oheap = []  
  
 # Помещаем начальную вершину в очередь  
 heapq.heappush(oheap, (fscore[start], start))  
  
 # Цикл пока очередь не опустеет  
 while oheap:  
  
 # Берем первый элемент очереди  
 current = heapq.heappop(oheap)[1]  
  
 # Если мы достигли цели, то возвращаем путь  
 if current == goal:  
 data = []  
 while current in came\_from:  
 data.append(current)  
 current = came\_from[current]  
 return data  
  
 # Кладем текущую вершину в список посещенных  
 close\_set.add(current)  
  
 # Просматриваем все соседние вершины  
 for i, j in neighbors:  
 neighbor = current[0] + i, current[1] + j  
  
 # Выбрасываем вершины, которые выходят за границы лабиринта  
 if neighbor[0] < 0 or neighbor[0] >= len(array):  
 continue  
 if neighbor[1] < 0 or neighbor[1] >= len(array[0]):  
 continue  
  
 # Выбрасываем блокированные вершины  
 if array[neighbor[0]][neighbor[1]] == "#":  
 continue  
  
 # Вычисляем стоимость пути через текущую вершину  
 tentative\_g\_score = gscore[current] + heuristic(current, neighbor)  
  
 # Добавляем новый узел в очередь приоритетов  
 if neighbor in close\_set and tentative\_g\_score >= gscore.get(neighbor, 0):  
 continue  
  
 # Обновляем лучший путь до вершины  
 came\_from[neighbor] = current  
 gscore[neighbor] = tentative\_g\_score  
 fscore[neighbor] = tentative\_g\_score + heuristic(neighbor, goal)  
 heapq.heappush(oheap, (fscore[neighbor], neighbor))  
  
 # Если путь не был найден, возвращаем None  
 return None  
Before\_the\_exit = astar(maze,key, end)  
  
def Signs(maze, path, symbol):  
 for cord in path:  
 x, y = cord  
 maze[x][y] = symbol  
 x, y = key  
 maze[x][y] = "\*"  
 return maze  
  
res = greedy\_algorithm(maze, start, key)  
  
  
maze\_to\_the\_key = Signs(maze, res, ".")  
maze\_to\_exit = Signs(maze, Before\_the\_exit, ",")  
  
  
  
def final\_file(maze, filename):  
 with open(filename, "w") as file:  
 for row in maze:  
 for elem in row:  
 file.write(str(elem))  
 file.write("\n")  
  
final\_file(maze, "maze-for-me.txt")