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-33 Белкин М.А.

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

Содержание

[Введение 3](#_Toc134639968)

[Задачи 3](#_Toc134639969)

[1.Теоретическая часть 4](#_Toc134639970)

[2. Реализация алгоритма 5](#_Toc134639971)

[Пример работы 8](#_Toc134639972)

[Заключение 10](#_Toc134639973)

[Список литературы 11](#_Toc134639974)

[Приложение 1 12](#_Toc134639975)

[Листинг программы 12](#_Toc134639976)

# Введение

Алгоритмы обхода графа являются одной из важнейших задач в программировании, которые широко используется для решения многих разнообразных задач.

Многие прикладные задачи могут быть сформулированы в терминах графа, и в таких случаях решение задачи может потребовать обход графа. Например, анализ социальных сетей, маршрутизация пакетов в сетях, поиск кратчайшего пути в графе дорог или маршрутизация полёта космического аппарата. Эти задачи могут быть решены эффективно с помощью алгоритмов обхода графа.

Одним из таких является алгоритм Greedy Best-First.

Жадные алгоритмы не смотрят в будущее, чтобы выбрать глобальное оптимальное решение. Их интересует только лучшее решение в данный момент. Но общее оптимальное решение может отличаться от решения, которое выбирает алгоритм на каждом шаге своей работы. Так же они никогда не оглядываются назад на то, что сделали, чтобы понять, нужна ли глобальная оптимизация.

**Цель работы**: реализовать алгоритмы обхода графа: Greedy Best-First 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;
* Реализовать алгоритмы с заданными параметрами;
* Сохранить результаты обходов лабиринта и получившиеся маршруты в файл.

# 1.Теоретическая часть

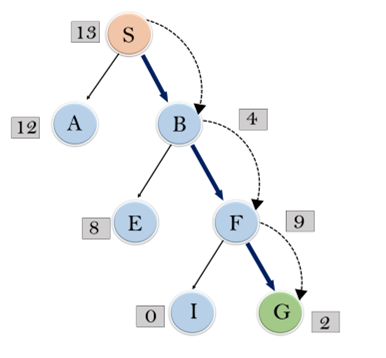
Для реализации алгоритма Greedy Best-First необходимо разобраться как он работает.

Жадный поиск работает, оценивая стоимость каждого возможного пути, а затем расширяя путь с наименьшей стоимостью. Этот процесс повторяется до тех пор, пока цель не будет достигнута.

Алгоритм использует эвристическую функцию, чтобы определить, какой путь является наиболее перспективным. Эвристическая функция учитывает стоимость текущего пути и расчетную стоимость оставшихся путей. Если стоимость текущего пути ниже оценочной стоимости остальных путей, то выбирается текущий путь.

Недостатком алгоритма Greedy Best-First является его низкая точность. Иногда он может выбрать неоптимальный путь из-за эвристической функции, которая не всегда оценивает расстояние между двумя вершинами точно. Также алгоритм может попадать в локальные минимумы и не находить оптимальный путь.

Рассмотрим приведенную ниже задачу поиска, и пройдем ее, используя жадный поиск с поиском по первому наилучшему. На каждой итерации каждый узел расширяется с использованием функции оценки.



# 2. Реализация алгоритма

1. Жадный алгоритм

Создаем функцию жадного алгоритма поиска пути в лабиринте от точки start до точки end, добавляя все необходимые параметры.

def Search\_BestFirst(maze, height, width, start, end):  
 path = [start] # Путь до выхода  
 x, y = start

Проверяем соответствие точки координатам

while path: # Пока есть путь  
 if (x, y) == end:   
 return path

Перебираем все возможные перемещения

for dx, dy in [(0, 1), (1, 0), (0, -1), (-1, 0)]:  
 nx, ny = x + dx, y + dy

Если соседняя позиция не выходит за границы лабиринта и не является стеной, то добавляем точку в путь до выхода, закрываем путь назад.

if 0 <= nx < height and 0 <= ny < width and maze[nx][ny]:  
 path.append((nx, ny))  
 maze[nx][ny] = 0  
  
 x, y = nx, ny  
  
 break

Если пришли в тупик, удаляем последнюю точку из пути и возвращаемся в предыдущую.

else:  
 path.pop()  
 if path:  
 x, y = path[-1]  
  
return []

1. Алгоритм A\*

Создаем стартовый и конечный узел, а так же список посещенных и не посещенных узлов.

def Search\_Astar(maze, height, width, start, end):  
 start\_node = Node(None, start) # стартовый узел  
 end\_node = Node(None, end) # конечный узел  
  
 start\_node.g = 0  
 start\_node.h = (((start\_node.pos[0] - end\_node.pos[0]) \*\* 2) +  
 ((start\_node.pos[1] - end\_node.pos[1]) \*\* 2)) \*\* 0.5  
 start\_node.f = start\_node.g + start\_node.h  
  
 not\_visited = [start\_node] # Список непосещенных узлов  
 visited = [] # Список посещенных узлов

Пока список не посещенных узлов непустой, берем первый узел в списке не посещенных.

while len(not\_visited) > 0:  
 cur\_node = not\_visited[0]  
 cur\_ind = 0 # Его индекс

Перебираем индексы не посещенных узлов, чтобы найти узел с наименьшей стоимостью перемещения.

for ind in range(len(not\_visited)):  
 if not\_visited[ind].f < cur\_node.f:  
 cur\_node = not\_visited[ind]  
 cur\_ind = ind

Удаляем узел с наименьшей стоимостью перемещения из списка не посещенных и добавляем в список посещенных.

not\_visited.pop(cur\_ind)  
visited.append(cur\_node)

Если узел, в котором мы сейчас находимся является финальным узлом, проходимся проходимся по ветке узлов и генерируем путь.

if cur\_node == end\_node:  
 path = []  
  
 cur = cur\_node  
 while cur is not None:  
 path.append(cur.pos)  
 cur = cur.parent  
  
 return path

Перебираем все возможные перемещения и вычисляем позицию соседнего узла.

for new\_pos in [(0, 1), (1, 0), (0, -1), (-1, 0)]:  
 node\_pos = (cur\_node.pos[0] + new\_pos[0], cur\_node.pos[1] + new\_pos[1])

Если координаты соседнего узла не за границами лабиринта и не является стеной, создаем новый узел.

if (0 <= node\_pos[0] < height and  
 0 <= node\_pos[1] < width and  
 maze[node\_pos[0]][node\_pos[1]]):  
 new\_node = Node(cur\_node, node\_pos)  
  
 if new\_node not in visited:  
 # Вычисляем веса  
 new\_node.g = cur\_node.g + 1  
 new\_node.h = (((new\_node.pos[0] - end\_node.pos[0]) \*\* 2) +  
 ((new\_node.pos[1] - end\_node.pos[1]) \*\* 2)) \*\* 0.5  
 new\_node.f = new\_node.g + new\_node.h  
  
 eq\_nodes = []

Ищем этот узел в списке не посещенных и удаляем те, у которых узлы имеют стоимость перемещения больше

for i in range(len(not\_visited)):  
 if new\_node == not\_visited[i]:  
 if new\_node.f > not\_visited[i].f:  
 eq\_nodes.append(not\_visited[i])  
 else:  
 not\_visited.pop(i)

Если нет узлов которые соответствуют данному узлу, до добавляем го в список не посещенных.

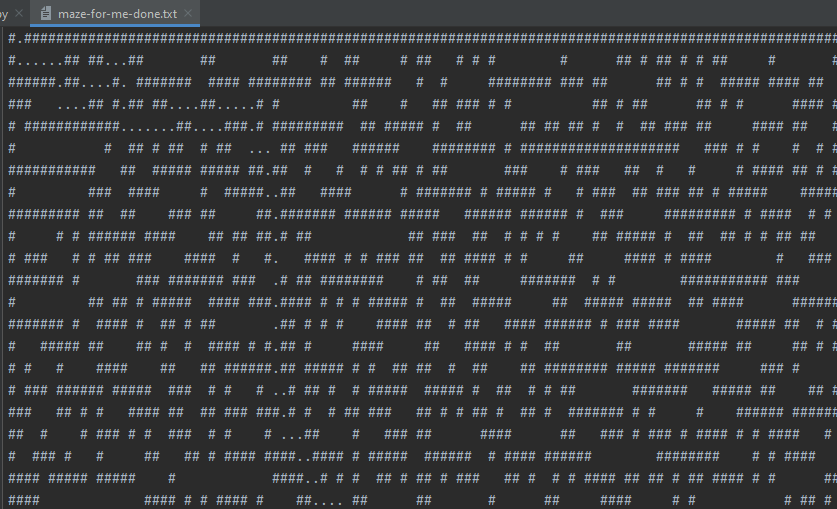
if len(eq\_nodes) <= 0:  
 not\_visited.append(new\_node)  
  
return []

# Пример работы

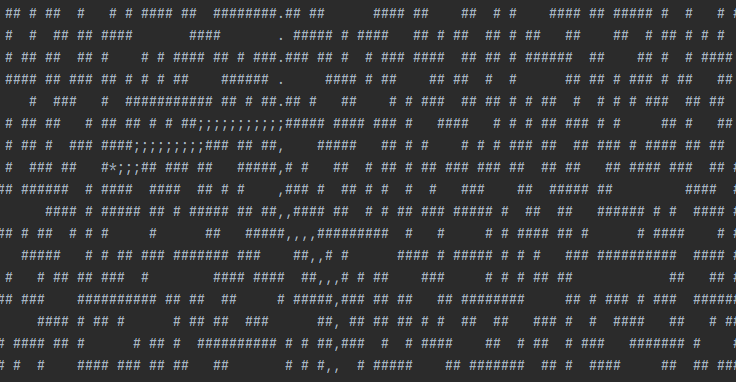
Вводим данные



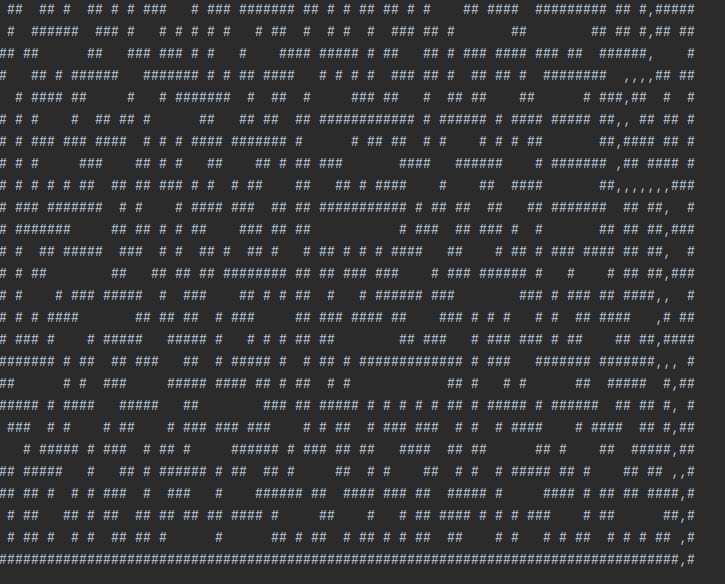
В текстовом документе создается путь выхода из лабиринта



Ключ



Выход



# Заключение

В ходе проделанной работы был создан код жадного алгоритма для прохождения маршрута от начальной координаты аватара до ключа, а после с помощью A\* от ключа до ближайшего выхода.

В результате мы получили траекторию прохождения лабиринта.

# Список литературы

1. Система вопросов и ответов о программировании : электронный сайт.

URL: <https://stackoverflow.com/> (дата обращения: 22.04.2023).

1. Веб-форум: электронный сайт.

URL: <https://www.cyberforum.ru/> (дата обращения: 22.04.2023).

1. Универсальная интернет энциклопедия: официальный сайт.

URL: https://ru.wikipedia.org/ (дата обращения: 21.04.2023).

1. Сайт технологий программирования: электронный сайт.

URL: https://metanit.com/ (дата обращения: 20.04.2023).

1. Веб-форум: электронный сайт.

URL: https://pmg.org.ru/ (дата обращения: 20.04.2023).

1. Система вопросов и ответов: электронный сайт.

URL: https://habr.com/ru/all/ (дата обращения: 20.04.2023).

1. Платформа современных технологий: электронный сайт.

URL: https://dtf.ru/ (дата обращения: 20.04.2023).

1. Сайт технологий программирования: электронный сайт.

URL: https://tproger.ru/ (дата обращения: 20.04.2023).

# Приложение 1

# Листинг программы

class Node:

def \_\_init\_\_(self, parent=None, pos=None):

self.parent = parent

self.pos = pos

self.g = 0

self.h = 0

self.f = 0

def \_\_eq\_\_(self, other):

return self.pos == other.pos

def Search\_Astar(maze, height, width, start, end):

start\_node = Node(None, start)

end\_node = Node(None, end)

start\_node.g = 0

start\_node.h = (((start\_node.pos[0] - end\_node.pos[0]) \*\* 2) +

((start\_node.pos[1] - end\_node.pos[1]) \*\* 2)) \*\* 0.5

start\_node.f = start\_node.g + start\_node.h

not\_visited = [start\_node]

visited = []

while len(not\_visited) > 0:

cur\_node = not\_visited[0]

cur\_ind = 0

for ind in range(len(not\_visited)):

if not\_visited[ind].f < cur\_node.f:

cur\_node = not\_visited[ind]

cur\_ind = ind

not\_visited.pop(cur\_ind)

visited.append(cur\_node)

if cur\_node == end\_node:

path = []

cur = cur\_node

while cur is not None:

path.append(cur.pos)

cur = cur.parent

return path

for new\_pos in [(0, 1), (1, 0), (0, -1), (-1, 0)]:

node\_pos = (cur\_node.pos[0] + new\_pos[0], cur\_node.pos[1] + new\_pos[1])

if (0 <= node\_pos[0] < height and

0 <= node\_pos[1] < width and

maze[node\_pos[0]][node\_pos[1]]):

new\_node = Node(cur\_node, node\_pos)

if new\_node not in visited:

new\_node.g = cur\_node.g + 1

new\_node.h = (((new\_node.pos[0] - end\_node.pos[0]) \*\* 2) +

((new\_node.pos[1] - end\_node.pos[1]) \*\* 2)) \*\* 0.5

new\_node.f = new\_node.g + new\_node.h

eq\_nodes = []

for i in range(len(not\_visited)):

if new\_node == not\_visited[i]:

if new\_node.f > not\_visited[i].f:

eq\_nodes.append(not\_visited[i])

else:

not\_visited.pop(i)

if len(eq\_nodes) <= 0:

not\_visited.append(new\_node)

return []

def Search\_BestFirst(maze, height, width, start, end):

path = [start] # Путь до выхода

x, y = start

while path:

if (x, y) == end:

return path

for dx, dy in [(0, 1), (1, 0), (0, -1), (-1, 0)]:

nx, ny = x + dx, y + dy

if 0 <= nx < height and 0 <= ny < width and maze[nx][ny]:

path.append((nx, ny))

maze[nx][ny] = 0

x, y = nx, ny

break

else:

path.pop()

if path:

x, y = path[-1]

return []

# Имя файла с лабиринтом

# input\_file\_name = "test\_maze.txt"

input\_file\_name = "maze-for-u.txt"

output\_file\_name = "maze-for-me-done.txt"

# Лабиринты

maze = []

key\_maze = []

exit\_maze = []

# Открываем файл с лабиринтом и считываем его в матрицу

with open(input\_file\_name, "r", encoding="utf-8") as f:

file = f.readlines()

height = len(file)

width = len(file[0]) - 1

for line in file:

tmp\_line = [0 if sym == "#" else 1 for sym in line[:-1]]

maze.append(tmp\_line[:])

key\_maze.append(tmp\_line[:])

exit\_maze.append(tmp\_line[:])

print("Координаты аватара:")

avatar\_ = (int(input(f"[0 <= x <= {width - 1}]: ")), int(input(f"[0 <= y <= {height - 1}]: ")))

print("\nКоординаты ключа:")

key\_ = (int(input(f"[0 <= x < {width - 1}]: ")), int(input(f"[0 <= y < {height - 1}]: ")))

print("\nКоординаты выхода:")

exit\_ = (int(input(f"[0 <= y < {width - 1}]: ")), int(input(f"[0 <= y < {height - 1}]: ")))

path\_to\_key = Search\_BestFirst(key\_maze, height, width, avatar\_, key\_)

path\_to\_exit = Search\_Astar(exit\_maze, height, width, key\_, exit\_)

with open(output\_file\_name, "w", encoding="utf-8") as f:

for x in range(height):

for y in range(width):

if (x, y) == key\_:

f.write("\*")

elif ((x, y) in path\_to\_exit and

(x, y) in path\_to\_key):

f.write(";")

elif (x, y) in path\_to\_key:

f.write(".")

elif (x, y) in path\_to\_exit:

f.write(",")

elif maze[x][y] == 0:

f.write("#")

else:

f.write(" ")

f.write("\n")