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-34 Рылова Е.Д.

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

Оглавление

Введение

Алгоритмы обхода графа являются одной из важнейших задач в программировании.

Поиск пути — определение компьютером самого оптимального маршрута между двумя точками. Чаще всего можно столкнуться при разработке игр, разработке роботов-манипуляторов, например, если в них есть свободное пространство. Компьютеру нельзя просто сказать куда нужно идти, что нужно сделать. Разработчику необходимо указать чёткий набор команд (найти синус угла между соседними клетками, повернуть персонажа(манипулятор) на угол, переместиться на 3 метра) или указать полный путь по точкам от A до Z. Но ведь разработчик не может написать один паттерн поведения для всех случаев, нам необходимо чтобы программа сама находила промежуточные точки и шла по ним. Иногда задача должна быть решена идеально, т.е. программа обязана найти лучший путь.

В данной работе я рассматривала множество способов, однако применила лишь два: Алгоритм поиска в ширину (Breadth-first search) и алгоритм A*.

Цель работы: реализовать алгоритмы обхода графа: в ширину и A* 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.

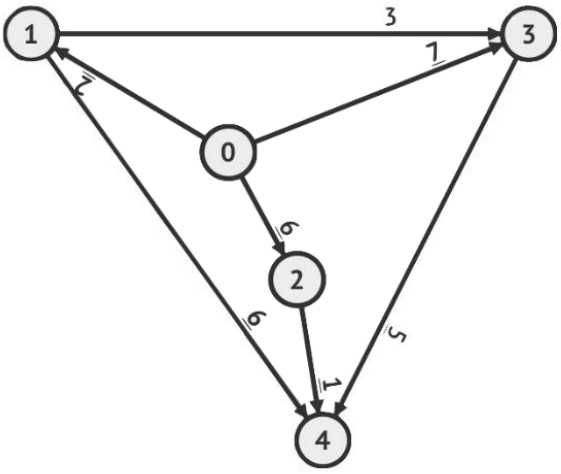
1.Теоретическая часть

Графы

Граф - это топологичекая модель, которая состоит из множества вершин и множества соединяющих их рёбер. При этом значение имеет только сам факт, какая вершина с какой соединена.

Вершина - точка в графе, отдельный объект, для топологической модели графа не имеет значения координата вершины, её расположение, цвет, вкус, размер; однако при решении некоторых задач вершины могут раскрашиваться в разные цвета или сохранять числовые значения.

Ребро - неупорядоченная пара двух вершин, которые связаны друг с другом. Эти вершины называются концевыми точками или концами ребра. При этом важен сам факт наличия связи, каким именно образом осуществляется эта связь и по какой дороге - не имеет значения; однако рёбрам может быть присвоен “вес”, что позволит говорить о “нагруженном графе” и решать задачи оптимизации [1].



Алгоритмы поиска пути

Поиск в ширину

Поиск в ширину – это алгоритм поиска пути на графе, который работает в ширину, то есть просматривает все вершины на одном уровне, перед тем как перейти на следующий уровень [2].

Алгоритм начинает с начальной вершины, отмечает её как посещённую и ставит её в очередь. Затем он перебирает все соединенные вершины, отмечает их как посещенные и ставит их в конец очереди пройденных вершин, и так для всех непройденных вершин (это необходимо для гарантии, что алгоритм не пройдёт по одной вершине несколько раз). Алгоритм работает пока не дойдёт до конечной вершины. Именно так алгоритм идёт в ширину и рассматривает каждый столбец/уровень графа [3].

Поиск в ширину автоматически находит кратчайший путь между начальной и конечной вершинами, так как обходит вершины в порядке их удаления от начала.

Алгоритм особенно выгоден, если длины рёбер одинаковые.

Плюсы:

- Гарантирует нахождение кратчайшего пути;
- Простота реализации. Алгоритм поиска в ширину - это простой алгоритм, который легко реализовать;
- Подходит для поиска пути в небольших графах. Хорошо работает для небольших графов, которые могут храниться в памяти целиком;
- Удобный для поиска всех вершин, находящихся на расстоянии k от начальной вершины.

Минусы:

- Неэффективность в случае больших графов и поиска пути в длинном списке;
- Затратное хранение информации о посещённых вершинах при работе с большими графами;
- Невозможность работать с цикличными графами, так как возникает опасность зацикливания алгоритма [4].

Алгоритм может быть неэффективным при использовании на графах с большим количеством рёбер, так как он медленно ищет путь, который может быть найден более эффективными алгоритмами.

В целом, поиск в ширину является полезным инструментом для обхода и нахождения кратчайшего пути в графе, особенно если граф небольшой или имеет несколько разных уровней. Однако этот алгоритм не лучший выбор для работы с большими графами, цикличными графами или графами с ребрами отрицательного веса.

A* (A Star)

Алгоритм A* - это эвристический алгоритм поиска кратчайшего пути. Основная идея заключается в том, чтобы оценить расстояние от текущей вершины до конечной вершины с помощью эвристической функции. Эта функция позволяет оценить стоимость пути от текущей вершины до цели через остальные вершины графа. Алгоритм A* учитывает как пройденное расстояние до текущей вершины, так и эвристическую оценку до цели. В отличие от жадного алгоритма, он выбирает на каждом шаге не только

наилучший вариант, но и учитывает пройденный путь до текущей вершины [5].

Плюсы:

- Оптимальность. A* гарантирует нахождение кратчайшего пути, если эвристика правильно выбрана, т.е. она должна быть допустимой (ни в коем случае не завышать длину пути);
- Скорость. A* работает быстрее, чем многие другие алгоритмы поиска пути, благодаря чему он может использоваться в реальном времени, например, при поиске пути в видеоиграх, робототехнике и т.д.;
- Модифицируемость. A* может быть адаптирован к различным типам графов и задач.

Минусы:

- Неприменимость для больших графов. (В случае больших графов, A* может потребовать большие вычислительные затраты);
- Не всегда находит решение. (A* не сможет найти путь если он ведет через область, которая закрыта от начальной точки эвристикой);
- Зависимость от выбора эвристики. (Эффективность работы алгоритма напрямую зависит от качества выбора эвристической функции)[5].

В целом, A* является универсальным алгоритмом поиска пути и может быть применен к большинству задач, требующих нахождения кратчайшего пути в графах.

2. Реализация алгоритма

До реализации алгоритмов поиска пути:

Нам необходимо передать в программу наш лабиринт и задать ему вид, более понятный для компьютера. Мы считываем файл и заменяем стены и коридоры на единицы и нули. Узнаём размеры лабиринта, чтобы передавать точку входа и точку сокровища(ключа). Вспомогательная функция для определения окрестности активной точки, какие пути доступны дальше. Создаём функцию для определения выхода из лабиринта.

Алгоритм поиска в ширину

1. Создаём очередь и добавляем в неё вершину, с которой начинается поиск. Сохраняем начальную вершину в множество посещённых.
2. Извлекаем из очереди первую вершину и проверяем, смежна ли она с другими вершинами. Если да, то добавляем их в очередь и множество посещённых.
3. Повторяем шаг 2 для каждой вершины из очереди, пока очередь не станет пустой.
4. Если сокровище найдено, то выходим из цикла поиска[2].

Вычислим эврестическую функцию, для алгоритма A*, поскольку он будет учитывать выгодность каждого шага при приближении к выходу. Она вычисляется суммой эврестического расстояния H от текущей вершины до конечной вершины и расстояния G, от начальной вершины до текущей вершины (то есть пройденного пути). Например, расстояние по прямой от входа до выхода.

Алгоритм поиска A*

1. Инициализируем пустое множество открытых вершин и закрытое множество вершин.
2. Добавляем начальную вершину в открытый список.

3. Инициализируем значения G (расстояние от начальной вершины) и H (эврестическое расстояние до конечной вершины) для начальной вершины.
4. Пока открытый список не пуст, выполняем следующее действие: Выбираем вершину с наименьшей суммой F(G+H) из открытого списка (оптимальный шаг). Если оптимальная вершина - это конечная вершина, то мы нашли путь. Возвращаем путь от начальной вершины до конечной вершины. Для каждой соседней вершины с оптимальной вершиной: если же соседняя вершина закрыта, пропускаем ее и переходим к следующей.
5. Если соседняя вершина еще не добавлена в открытый список, добавляем ее туда и вычисляем ее значение G и H.
6. Если же она уже в открытом списке, то сравниваем значение G нового пути с уже имеющимся. Если новый путь короче, то заменяем старый путь на новый.
7. Перемещаем оптимальную вершину в закрытый список.
8. Если открытый список пуст и мы не нашли путь, то путь не найден [5].

Найденные пути заменяем в нашем преобразованном массиве на точки (от входа до сокровища (ключа)) и на запятые (от сокровища до выхода из лабиринта). Сохраняем наш файл и меняем в нём единицы на стенки, ноли на проходы.

Пример работы

Исходный файл:

test_maze.txt

```
# ##################
# # ## # ##   #    #
#   #       ### ## #
### ## ####   #  ###
#      ## ## ## ####
### ##     #     # #
#    ## # ## # #   #
## # #  # ## ### ###
#  #### #  #  ##   #
##   ## ## # ### ###
## #### ## # #     #
#     # ## # # # # #
## #### ## # # # ###
#     #  ### ###   #
## #### ##   ##### #
#   ### ## #    ## #
# #   #  ### ##  # #
# # ######    ## # #
# #      # # ##  # #
################## #
```

Итоговый файл:

itog.txt

```
#.##################
#.# ## # ##   #    #
#...# ......### ## #
###.##.####.. #  ###
#  ....## ##.## ####
### ##     #.....# #
#    ## # ## # #.  #
## # #  # ## ###.###
#  #### #  #  ##.  #
##   ## ## # ###.###
## #### ## # #  .  #
#     # ## # # #.# #
## #### ## # # #.###
#     #  ### ###...#
## #### ##   #####*#
#   ### ## #    ##,#
# #   #  ### ##  #,#
# # ######    ## #,#
# #      # # ##  #,#
##################,#
```

Заключение

В ходе проделанной работы была создана программа для поиска пути в лабиринте. Поиск пути был осуществлен с помощью алгоритмов обхода графа в ширину и A Star. Разработка программы потребовала изучение алгоритмов поиска пути и были реализованы 2 из них: Поиск в ширину, алгоритм A* (A Star).

Список литературы

1. Волосатов Е., «Теория графов. Термины и определения в картинках» [Электронный ресурс]. URL: https://habr.com/ru/companies/otus/articles/568026/ (Дата обращения: 15.05.2023).
2. “Geek for Geeks”, «Breadth First Search or BFS for a Graph» [Электронный ресурс]. URL: https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/ (Дата обращения: 14.05.2023).
3. “Try Khow”, «Algorithms on Graphs: Let’s talk Depth-First Search (DFS) and Breadth-First Search (BFS) | by Try Khov | Medium» [Электронный ресурс]. URL: https://trykv.medium.com/algorithms-on-graphs-lets-talk-depth-first-search-dfs-and-breadth-first-search-bfs-5250c31d831a (Дата обращения: 13.05.2023).
4. Hans-Peter Störr, «What is Breadth First Search and how can I implement it?» [Электронный ресурс]. URL: https://stackoverflow.com/questions/3332947/what-are-the-practical-factors-to-consider-when-choosing-between-depth-first-sea (Дата обращения: 13.05.2023).
5. Амит Пратап, «Amit's Thoughts on Pathfinding» [Электронный ресурс]. URL: https://theory.stanford.edu/~amitp/GameProgramming/ (Дата обращения: 15.05.2023).

Приложение 1

Листинг программы

```
from queue import Queue

def read_maze(fileMaze="maze-for-me.txt"):
    maze = [list(line.replace("#", "1").replace(" ", "0")) for line in
open(fileMaze).read().split("\n")[:-1]]
    return maze

# Узнаём размеры лабиринта
try:
    maze = read_maze()
    size_x = len(maze)
    size_y = len(maze[0])
except Exception:
    print("Исходный файл не найден!")

def near(x, y, maze):
    near = []
    if y == 0:
        if x == 0:
            near = [["#", maze[x][y + 1]],
                    [maze[x + 1][y], maze[x + 1][y + 1]]]
        elif x == (size_x - 1):
            near = [[maze[x - 1][y], maze[x - 1][y + 1]],
                    ["#", maze[x][y + 1]]]
        else:
            near = [[maze[x - 1][y], maze[x - 1][y + 1]],
                    ["#", maze[x][y + 1]],
                    [maze[x + 1][y], maze[x + 1][y + 1]]]
    elif y == (size_y - 1):
        if x == 0:
            near = [[maze[x][y - 1], "#"],
                    [maze[x + 1][y - 1], maze[x + 1][y]]]
        elif x == (size_x - 1):
            near = [[maze[x - 1][y - 1], maze[x - 1][y]],
                    [maze[x][y - 1], "#"]]
        else:
            near = [[maze[x - 1][y - 1], maze[x - 1][y]],
                    [maze[x][y - 1], "#"],
                    [maze[x + 1][y - 1], maze[x + 1][y]]]
```

```
    elif x == 0:
        near = [[maze[x][y - 1], "#", maze[x][y + 1]],
                [maze[x + 1][y - 1], maze[x + 1][y], maze[x + 1][y + 1]]]
    elif x == (size_x - 1):
        near = [[maze[x - 1][y - 1], maze[x - 1][y], maze[x - 1][y + 1]],
                [maze[x][y - 1], "#", maze[x][y + 1]]]
    else:
        near = [[maze[x - 1][y - 1], maze[x - 1][y], maze[x - 1][y + 1]],
                [maze[x][y - 1], "#", maze[x][y + 1]],
                [maze[x + 1][y - 1], maze[x + 1][y], maze[x + 1][y + 1]]]
    return near # Возможные пути

def sel_start(maze): # узнаем координаты входа
    while 1:
        try:
            x = int(input("Введите точку входа X: "))
            if x >= 0 and x < size_x:
                y = int(input("Введите точку входа Y: "))
                if y >= 0 and y < size_y:
                    if str(maze[x][y]) == "1":
                        print("\nТут нет входа..")
                    else:
                        return (x, y)
                else:
                    print("\nВы ушли за пределы карты")
                    continue
            else:
                print("\nВы ушли за пределы карты")
                continue
        except Exception:
            print("\nЧто-то не то, давайте попробуем ещё раз")

def sel_tresuaerr(maze): # узнаём координаты нашего сокровища
    while 1:
        try:
            x = int(input("Координата ключа X: "))
            if x >= 0 and x < size_x:
                y = int(input("Координата ключа Y: "))
                if y >= 0 and y < size_y:
                    if str(maze[x][y]) == "1":
                        print("\nТут стенка")
                    else:
                        return (x, y)
```

```
                else:
                    print("\nВы ушли за пределы карты")
                    continue
            else:
                print("\nВы ушли за пределы карты")
                continue
        except Exception:
            print("\nЧто-то не то, давайте попробуем ещё раз")

def sel_exit(maze): # находим доступный выход
    for y in range(size_y):
        if int(maze[size_x-1][y]) == 0:
            return (size_x - 1,y)

start = sel_start(maze) # задаём точку старта
zvezda = sel_tresuaerr(maze) # задаём точку сокровища
end = sel_exit(maze) # задаём выход


def get_moves(pos, maze): # доступные движения (вверх, вниз, влево, вправо)
    moves = []
    if pos[0] > 0 and not int(maze[pos[0] - 1][pos[1]]):
        moves.append((pos[0] - 1, pos[1]))
    if pos[0] < size_x - 1 and not int(maze[pos[0] + 1][pos[1]]):
        moves.append((pos[0] + 1, pos[1]))
    if pos[1] > 0 and not int(maze[pos[0]][pos[1] - 1]):
        moves.append((pos[0], pos[1] - 1))
    if pos[1] < size_y - 1 and not int(maze[pos[0]][pos[1] + 1]):
        moves.append((pos[0], pos[1] + 1))
    return moves

def bfs(start, end): # делаем поиск в ширину
    queue = Queue()
    queue.put((start, [start]))
    while not queue.empty():
        pos, path = queue.get()
        if pos == end:
            return path
        for move in get_moves(pos, maze):
            if move not in path:
                queue.put((move, path + [move]))
    return None
```

```
way_bfs = bfs(start, zvezda) # поиск в ширину ДО сокровища

def heuristic(current, end): # узнаём цену выхода
    return abs(current[0] - end[0]) + abs(current[1] - end[1])

def a(maze, start, end): # делаем метод А звёздочка
    open_list = [start]
    closed_list = []
    came_from = {}
    g_score = {start: 0}
    f_score = {start: heuristic(start, end)}

    while open_list:
        current = min(open_list, key=lambda x: f_score[x])
        if current == end:
            path = [end]
            while current in came_from:
                current = came_from[current]
                path.append(current)
            path.reverse()
            return path

        open_list.remove(current)
        closed_list.append(current)

        for neighbor in get_moves(current, maze):
            if neighbor in closed_list:
                continue
            tentative_g_score = g_score[current] + 1
            if neighbor not in open_list or tentative_g_score < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g_score
                f_score[neighbor] = tentative_g_score + heuristic(neighbor, end)
                if neighbor not in open_list:
                    open_list.append(neighbor)

    return None

way_a = a(maze, zvezda, end) # находим путь от сокровища до выхода

try:
    for point in way_bfs:
        maze[point[0]][point[1]] = "." # От входа, до сокровища
```

```
    # Заносим путь от ключа до выхода
    for point in way_a:
        maze[point[0]][point[1]] = "," # От сокровища, до выхода

    maze[zvezda[0]][zvezda[1]] = "*"

    with open("maze-for-me-done.txt", "w") as f:
        f.write('\n'.join([''.join(map(str, line)) for line in maze]))
except Exception:
    print("Ошибка поиска пути")

with open("maze-for-me-done.txt") as f:
    text = f.read()

text = text.replace("0", " ").replace("1", "#")

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