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 Олейник Д. О.

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

Содержание

Введение:

Современный мир программирования и информационных технологий требует разработки эффективных алгоритмов и методов, способных решать различные задачи. Одной из наиболее распространенных задач является поиск пути в различных структурах данных. Алгоритмы поиска пути играют важную роль в таких областях, как робототехника, компьютерные игры, логистика и маршрутизация, анализ сетей и многое другое.

Цель работы: изучение основных алгоритмов поиска пути и реализация некоторых на языке программирования Python.

Для достижения данной цели необходимо выполнить следующие задачи:

1. Изучить основные алгоритмы поиска пути
2. Создать готовый продукт реализуя два алгоритма
3. Провести сравнительный анализ
4. Выявить преимущества и недостатки каждого из алгоритмов

Выполнение этих задач позволит достичь поставленной цели работы и получить полное представление о принципах алгоритмов поиска пути, их реализации на языке Python и применении структурного программирования при разработке программного кода.

Глава 1. Теоретическая часть

Поиск пути является важной задачей во многих областях, где требуется нахождение оптимального маршрута от одной точки к другой. Существует несколько основных алгоритмов, которые эффективно решают эту задачу. В данной главе мы рассмотрим некоторые из них.

1.1 Алгоритм A*:

Алгоритм A* является эффективным алгоритмом поиска пути, который комбинирует преимущества алгоритма Дейкстры и эвристической функции для оценки расстояния до целевой вершины. Он использует информацию о расстоянии от начальной вершины и эвристической оценке расстояния до цели для выбора следующей вершины. Алгоритм A* обеспечивает оптимальность и эффективность во многих ситуациях поиска пути [7].

1.2 Алгоритм Дейкстры:

Алгоритм Дейкстры предназначен для нахождения кратчайшего пути во взвешенном графе. Он начинает с исходной вершины и последовательно обрабатывает соседние вершины, присваивая им наименьшую известную длину пути от начальной вершины. Алгоритм Дейкстры продолжает расширяться от текущей вершины к ближайшим не посещённым вершинам, пока не достигнет конечной вершины или пока не будут обработаны все вершины [6].

1.3 Алгоритм поиска в глубину (DFS):

Алгоритм поиска в глубину также является одним из базовых алгоритмов поиска пути. Он работает путем исследования пути до тех пор, пока не достигнет конечной точки или не встретит тупиковую ситуацию. Алгоритм начинает с исходной вершины и рекурсивно исследует каждую доступную соседнюю вершину. Если в какой-то момент алгоритм достигает конечной точки или не может продолжать движение, он возвращается к предыдущей вершине и исследует другой путь. Поиск пути в глубину не

гарантирует нахождение кратчайшего пути, но может быть полезен в случаях, когда не требуется оптимальный результат или когда необходимо обнаружить все возможные пути [4].

1.4 Алгоритм поиска в ширину (BFS):

Алгоритм поиска в ширину является одним из простейших алгоритмов поиска пути. Он работает путем последовательного поиска всех вершин графа на одной глубине, прежде чем переходить на следующую глубину [5]. Алгоритм начинает с исходной вершины и помечает ее как посещенную. Затем он рассматривает все соседние вершины текущей вершины и помечает их как посещенные. Процесс продолжается, пока не будут посещены все достижимые из исходной вершины графа. Алгоритм BFS гарантированно находит кратчайший путь в невзвешенном графе. гарантированно находит кратчайший путь в невзвешенном графе [8].

1.5 Алгоритмы поиска пути в графах:

В графах с большим количеством вершин и ребер часто применяются специализированные алгоритмы поиска пути, такие как алгоритмы Джонсона или Флойда-Уоршелла. Эти алгоритмы предназначены для нахождения всех кратчайших путей между каждой парой вершин в графе и находят применение в сетевых технологиях и транспортных системах.

В данной главе мы описали основные алгоритмы поиска пути, включая алгоритм поиска в ширину, алгоритм поиска в глубину, алгоритм Дейкстры, алгоритм A* и некоторые алгоритмы поиска пути в графах. В следующих разделах мы более подробно рассмотрим реализацию этих алгоритмов на языке программирования Python и рассмотрим примеры их применения.

Глава 2. Сферы применения

Алгоритмы поиска пути являются одним из ключевых элементов во многих областях, где требуется определение наиболее оптимального пути для достижения конечной цели. Их широкий спектр применения охватывает различные сферы деятельности, включая транспортную логистику, навигацию, игровую разработку, робототехнику, анализ сетей и многое другое [3].

В навигационных системах и мобильных приложениях алгоритмы поиска пути помогают пользователям определить наиболее эффективный путь от точки А до точки Б. Они учитывают текущее положение пользователя, дорожные условия, наличие препятствий и другие факторы, чтобы предложить оптимальный маршрут. Это особенно полезно в городах, где есть много альтернативных маршрутов, и водителям необходимо выбрать оптимальный путь, чтобы избежать пробок и сэкономить время [1].

В игровой разработке алгоритмы поиска пути применяются для управления движением персонажей в игровом мире. Они определяют оптимальные пути для навигации персонажей через сложные уровни или преодоления препятствий. Такие алгоритмы позволяют создавать реалистичное поведение и искусственный интеллект для персонажей в играх, делая их более адаптивными и умными [2].

В робототехнике алгоритмы поиска пути используются для планирования движения роботов. Они позволяют роботам находить оптимальные пути в незнакомых или изменяющихся средах, избегать столкновений с препятствиями и выполнять задачи автономно. Например, роботы-пылесосы используют алгоритмы поиска пути для определения оптимального маршрута для уборки помещения без повторного прохождения через одни и те же участки.

Алгоритмы поиска пути также находят применение в анализе сетей и связей. Они помогают определить оптимальные пути или связи между узлами сети. Например, в телекоммуникационных системах алгоритмы поиска пути используются для определения оптимальных маршрутов передачи данных или оптимального размещения сетевых узлов [8].

Это лишь некоторые из многочисленных сфер применения алгоритмов поиска пути. Они являются важным инструментом для оптимизации и автоматизации маршрутных задач в различных областях, способствуя улучшению эффективности, экономии времени и ресурсов, а также созданию интеллектуальных и адаптивных систем.

Глава 3. Сравнение различных алгоритмов

В данном разделе мы проведем сравнительный анализ результатов экспериментов для различных алгоритмов поиска пути, таких как Алгоритм Дейкстры, Алгоритм A*, Алгоритм поиска в ширину (BFS) и Алгоритм поиска в глубину (DFS).

Алгоритм	Плюсы	Минусы
Дейкстры	Находит кратчайший путь в неотрицательном графе	Неэффективен для больших графов или графов с отрицательными весами
A*	Находит оптимальный путь с использованием эвристики	Может не находить оптимальный путь в зависимости от эвристики или достижимости цели
BFS	Находит путь с минимальным количеством ребер	Может быть неэффективен для больших графов или графов с весами ребер
DFS	Простота и понятность реализации	Не гарантирует нахождение оптимального пути, может зациклиться на циклах в графе

Эта таблица представляет основные преимущества и недостатки каждого алгоритма поиска пути [1]. Однако, важно учитывать, что выбор алгоритма зависит от конкретных требований и характеристик задачи, а также от особенностей графа или среды, в которой происходит поиск пути.

Глава 4. Реализация алгоритмов

Для реализации алгоритма поиска в ширину (BFS) и алгоритма A*, которые использовались в готовом продукте, потребуется следовать определенным шагам. Ниже приведены пошаговые инструкции для каждого из алгоритмов.

Реализация алгоритма BFS:

1. Создайте пустую очередь (queue) и добавьте в нее начальную вершину.
2. Создайте пустой список (visited), чтобы отслеживать уже посещенные вершины.
3. Пока очередь не пуста:
    - Извлеките вершину из очереди.
    - Если вершина уже посещена, пропустите ее.
    - Иначе, добавьте вершину в список посещенных.
    - Проверьте, является ли текущая вершина целевой. Если да, то поиск пути завершен.
    - Иначе, добавьте все смежные вершины текущей вершины в очередь (если они еще не были посещены).
4. Если целевая вершина не найдена и очередь опустела, значит пути нет [5].

Реализация алгоритма A* (A-star):

1. Создайте пустой открытый список (open list) и закрытый список (closed list).
2. Добавьте начальную вершину в открытый список.
3. Пока открытый список не пуст, выполните следующие действия:

5. Выберите вершину с наименьшей оценкой F из открытого списка. Это будет текущая вершина.
6. Если текущая вершина является целевой вершиной, вы нашли путь.

7. Переместите текущую вершину из открытого списка в закрытый список.
    - Для каждой соседней вершины текущей вершины выполните следующие действия:

        a) Если соседняя вершина уже находится в закрытом списке, проигнорируйте ее.
        b) Если соседняя вершина еще не находится в открытом списке, добавьте ее туда и вычислите оценку F.
        c) Если соседняя вершина уже находится в открытом списке, обновите ее оценку F, если новый путь короче.

8. Если открытый список стал пустым, и вы так и не нашли целевую вершину, значит путь не существует.

Обратите внимание, что реализация алгоритма A* также включает в себя функцию оценки F, которая учитывает стоимость пути от начальной вершины (G) и эвристику, оценивающую расстояние до целевой вершины (H) [7].

Заключение

В заключении курсовой работы по алгоритмам поиска пути, мы рассмотрели различные алгоритмы поиска пути. Эти алгоритмы широко применяются в области искусственного интеллекта, робототехники, игровой разработки и других областях, где требуется нахождение оптимального пути в графе или сетке.

В рамках нашего готового продукта мы успешно реализовали и применили оба этих алгоритма для решения задачи поиска пути. (приложение 1)

BFS является простым алгоритмом, который гарантирует нахождение пути с минимальным количеством ребер. Он хорошо подходит для графов с небольшим количеством узлов и не требует оценки стоимости пути или весов на ребрах. Однако, он может быть неэффективен для больших графов или графов с весами ребер.

A* является более сложным алгоритмом, который комбинирует эвристику и стоимость пути для нахождения оптимального пути. Он может эффективно работать для графов с большим количеством узлов и различными типами ребер. A* дает возможность контролировать производительность и точность алгоритма путем выбора соответствующей эвристики.

Алгоритмы поиска пути играют ключевую роль во многих областях и приложениях. Они предлагают различные подходы к нахождению пути и имеют свои преимущества и ограничения. Выбор конкретного алгоритма зависит от требований задачи, характеристик графа и контекста применения.

Список литературы

1. Алгоритмы: построение и анализ: учебник / Т. Х. Кормен, Ч. И. Лейзерсон, Р. Л. Ривест, К. Штайн. — 2-е изд. — М.: Вильямс, 2006. — 1296 с.
2. Бхаргава А. Грокаем алгоритмы. СПб.: Питер, 2018. 288 с.
3. Евстигнеев В.А. Применение теории графов в программировании. 1985. — С. 138—150.
4. Левитин А. В. Глава 5. Метод уменьшения размера задачи: Поиск в глубину // Алгоритмы. Введение в разработку и анализ — М.: Вильямс, 2006. — С. 212—215
5. Левитин А. В. Глава 5. Метод уменьшения размера задачи: Поиск в ширину // Алгоритмы. Введение в разработку и анализ — М.: Вильямс, 2006. — 576 с
6. Левитин А. В. Глава 9. Жадные методы: Алгоритм Дейкстры // Алгоритмы. Введение в разработку и анализ — М.: Вильямс, 2006. — С. 189—195
7. Лорьер Ж.-Л. Системы искусственного интеллекта / Пер. с фр. и ред. В. Л. Стефанюка. — М.: Мир, 1991. — С. 238—244.
8. Тердос Е., Клейберг Д. Алгоритмы. Разработка и применение. Классика Computers Science. СПб.: Питер, 2016. 800 с.

Приложение 1
Листинг программы

```
from collections import deque
import heapq
POSSIBLE_WAYS = ('N', 'S', 'W', 'E')
def heuristic(coord, target):
    x1, y1 = map(int, coord)
    x2, y2 = map(int, target)
    return ((x1 - x2) ** 2 + (y1 - y2) ** 2) ** 0.5
def is_coord_in_maze(maze, coord):
    len_x, len_y = len(maze), len(maze[0])
    if coord[0] < 0 or coord[0] >= len_x:
        return False
    if coord[1] < 0 or coord[1] >= len_y:
        return False
    return True
def is_path_clean(maze, coord):
    coord_tuple = tuple(coord)
    if maze[coord_tuple[0]][coord_tuple[1]] != '#':
        return True
    return False


def step(coord, direction):
    if direction == 'N':
```

```
        return [coord[0], coord[1] - 1].copy()
    elif direction == 'S':
        return [coord[0], coord[1] + 1].copy()
    elif direction == 'E':
        return [coord[0] + 1, coord[1]].copy()
    elif direction == 'W':
        return [coord[0] - 1, coord[1]].copy()
def bfs(maze, start, target):
    queue = deque()
    visited = set()
    parent = {}
    start = tuple(start)
    queue.append(start)
    visited.add(start)
    while queue:
        current = queue.popleft()
        if current == target:
            path = []
            while current != start:
                path.append(current)
                current = parent[current]
            path.append(start)
            path.reverse()
```

```
            return path
        for direction in POSSIBLE_WAYS:
            next_coord = tuple(step(list(current), direction))
            if not is_coord_in_maze(maze, next_coord) or not
is_path_clean(maze, next_coord):
                continue
            if next_coord not in visited:
                parent[next_coord] = current
                queue.append(next_coord)
                visited.add(next_coord)
    return None
def a_star(maze, start, target):
    open_list = [(0, start)]
    came_from = {}
    g_score = {start: 0}
    f_score = {start: heuristic(start, target)}

    while open_list:
        current = heapq.heappop(open_list)[1]
        if current == target:
            return reconstruct_path(came_from, current)
        for direction in POSSIBLE_WAYS:
            next_coord = step(current, direction)
```

```
            if not is_coord_in_maze(maze, next_coord) or not
is_path_clean(maze, next_coord):

                continue

            tentative_g_score = g_score[current] + 1

            next_coord = tuple(next_coord)

            if next_coord in g_score and tentative_g_score >=
g_score[next_coord]:

                continue

            came_from[next_coord] = current

            g_score[next_coord] = tentative_g_score

            f_score[next_coord] = tentative_g_score + heuristic(next_coord,
target)

            heapq.heappush(open_list, (f_score[next_coord], next_coord))

    return None

def reconstruct_path(came_from, current):

    path = [current]

    while current in came_from:

        current = came_from[current]

        path.append(current)

    path.reverse()

    return path

with open('maze-for-u.txt', 'r') as f:

    maze = [list(line.strip()) for line in f.readlines()]

start = [0, 1]  # Starting coordinate

for e in range(len(maze[0]) - 1, -1, -1):
```

```
    if maze[len(maze) - 1][e] == ' ':
        end_point = (len(maze) - 1, e)
        break
start_point = [0, 1]
while True:
    x = int(input("Координата сокровища по X: "))
    y = int(input("Координата сокровища по Y: "))
    if maze[x][y] != "#":
        treja_is_here = (x, y)
        break
    print("Стена! Выбери другие координаты")
for e in range(len(maze[0]) - 1, -1, -1):
    if maze[len(maze) - 1][e] == ' ':
        end_point = (len(maze) - 1, e)
        break
print("Путь найден!")
# BFS
path_start_to_treja = bfs(maze, start, treja_is_here)
#print(path_start_to_treja)
# A*
path_treja_to_end = a_star(maze, treja_is_here, end_point)
#print(path_treja_to_end)
if path_start_to_treja is not None:
```

```
    for coord in path_start_to_treja:
        maze[coord[0]][coord[1]] = '.'
if path_treja_to_end is not None:
    for coord in path_treja_to_end:
        maze[coord[0]][coord[1]] = ','
maze[treja_is_here[0]][treja_is_here[1]] = "*"
# Запись пути в файл
with open("maze-for-me-done.txt", "w") as txt_file:
    for line in maze:
        txt_file.write(''.join(line) + '\n')
```