Муниципальное автономное образовательное учреждение
лицей № 180
г. Нижнего Новгорода

Научно-исследовательская работа
«Алгоритмы нахождения кратчайшего пути»

Выполнил: Гаврин Алексей,
Ученик 7 м класса
Научный руководитель:
Сухова Маргарита Александровна

Нижний Новгород
2024

Содержание

Введение

В современном обществе, где технологические инновации играют все более значимую роль в разнообразных сферах деятельности, оптимизация процессов и увеличение продуктивности работы становятся приоритетными направлениями. Алгоритмы поиска кратчайшего пути представляют собой фундаментальные инструменты для реализации подобных задач, поскольку они позволяют находить наиболее эффективные и быстрые маршруты, сокращая затраты времени и ресурсов.

Эти алгоритмы находят широкое применение в разнообразных областях, включая логистику, навигацию, планирование сетевых маршрутов и организацию мероприятий. В области логистики, алгоритмы нахождения кратчайшего пути позволяют оптимизировать транспортные маршруты, снижая выбросы углекислого газа и затраты на топливо. В сфере навигации, эти алгоритмы помогают прокладывать маршруты для автомобилей, велосипедов и пешеходов, учитывая различные факторы, такие как дорожные условия, ограничения скорости и время суток. В области маршрутизации сетей, алгоритмы кратчайшего пути используются для оптимизации передачи данных между узлами сети, обеспечивая наиболее эффективный и быстрый обмен информацией. Наконец, в планировании мероприятий, алгоритмы нахождения кратчайших путей могут быть использованы для оптимизации маршрутов передвижения участников мероприятия, что позволяет сократить общее время мероприятия и улучшить его эффективность.

Таким образом, алгоритмы нахождения кратчайшего пути играют ключевую роль в современном мире, позволяя оптимизировать процессы и повышать эффективность работы в различных сферах деятельности.

Цель

Цель моего проекта изучить алгоритмы нахождения кратчайшего пути и разработка приложения для упрощения навигации по складу.

Алгоритмы поиска кратчайшего пути

Алгоритмы кратчайшего пути используются для определения наиболее эффективного или краткого маршрута между двумя точками в графе. Существует множество различных алгоритмов, которые могут быть классифицированы по различным категориям, включая:

1. Алгоритм Дейкстры — это один из самых популярных алгоритмов кратчайшего пути. Он используется для определения кратчайшего пути между двумя вершинами графа, учитывая, что все веса ребер неотрицательны.
2. Алгоритм Беллмана-Форда позволяет найти кратчайшие пути в графе с отрицательными весами ребер. Однако, если граф содержит цикл с отрицательным весом, алгоритм может выдать неправильный результат.
3. Алгоритм A* (или алгоритм поиска по восхождению к вершине) — это эвристический алгоритм, который находит кратчайший путь между двумя вершинами, учитывая заданную эвристическую функцию. Время работы алгоритма зависит от выбранной эвристической функции и может быть как полиномиальным, так и экспоненциальным.
4. Алгоритм Флойда – Уоршелла используется для нахождения кратчайших путей между всеми парами вершин в графе.
5. Алгоритм «Поиск в ширину» – это один из основных алгоритмов на графах, который позволяет найти все вершины на заданном расстоянии от начальной вершины. Алгоритм «Поиск в ширину» можно использовать для решения различных задач, таких как нахождение кратчайшего пути в графе, нахождение компонент связности графа и т.д.
6. Поиск в глубину – это метод обхода графа, который проходит через все вершины насколько возможно. Алгоритм перебирает все исходящие ребра из текущей вершины, идет в нерассмотренную вершину, возвращается и продолжает перебор ребер. Если после алгоритма остались не рассмотренные вершины, то алгоритм запускается снова от одной из них.

Поиск в ширину

Поиск в ширину — это алгоритм на графах, используемый для нахождения всех кратчайших путей от одной вершины графа до всех остальных вершин. Он имеет линейную временную сложность O(n + m), где n - количество вершин, а m - количество ребер в графе. Поиск в ширину может быть использован в различных приложениях, таких как:

- Определение кратчайшего маршрута в транспортной сети: поиск в ширину можно использовать для нахождения кратчайшего маршрута между двумя вершинами графа, представляющими города, с учетом стоимости проезда между каждой парой городов.
- Вычисление минимального расстояния между городами: поиск в ширину может использоваться для нахождения минимального расстояния между любой парой городов в транспортной сети.
- Нахождение кратчайших путей в лабиринтах: поиск в ширину также можно использовать для решения задачи нахождения кратчайших путей в лабиринтах. Лабиринт представляется в виде графа, где вершины соответствуют пересечениям лабиринта, а ребра - проходам между пересечениями.
- Компоненты связности и циклы: поиск в ширину используется для нахождения компонент связности графа и определения наличия циклов в графе.

Рассмотрим этот алгоритм на представленном ниже графе (рис. 1):

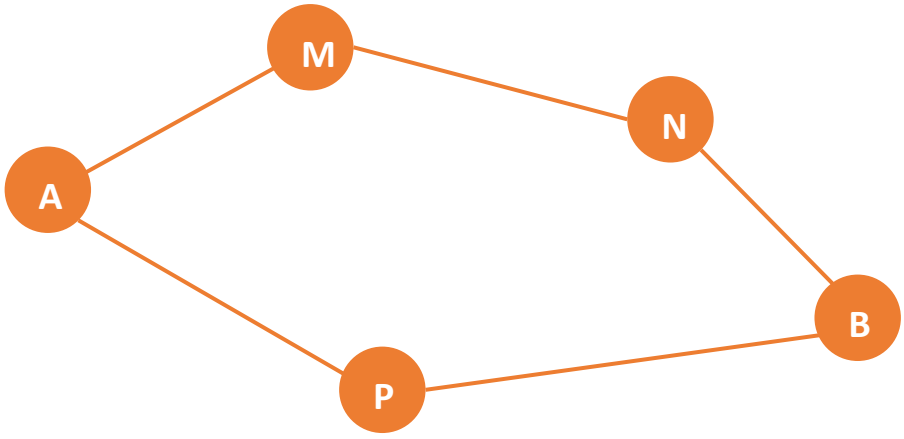

рис. 1

Программа см. Приложение 1.

Этот алгоритм реализует поиск в ширину для нахождения кратчайшего пути в графе. Входные данные представляют собой граф в виде словаря, где ключи — это вершины графа, а значения - списки смежных вершин. Функция **bfs** принимает стартовую вершину (**start**), конечную вершину (**goal**) и **граф**, и возвращает **словарь**, где ключами являются вершины, а значениями -

предшествующие вершины на кратчайшем пути от стартовой вершины к данной.

Алгоритм начинается с добавления стартовой вершины в очередь (**queue**) и добавления ее в набор посещенных вершин (**visited**). Затем на каждой итерации цикла выбираются вершины из очереди, находятся смежные вершины для каждой из них и добавляются в очередь, если они еще не были посещены. Если текущая вершина равна конечной, алгоритм прерывается и возвращается найденный кратчайший путь.

В конце работы алгоритма выводится кратчайший путь от стартовой до конечной вершины.

Поиск в глубину

Поиск в глубину – один из методов обхода графа. Стратегия поиска в глубину, как и следует из названия, состоит в том, чтобы идти «вглубь» графа, насколько это возможно. Алгоритм поиска описывается рекурсивно: перебираем все исходящие из рассматриваемой вершины рёбра. Если ребро ведёт в вершину, которая не была рассмотрена ранее, то запускаем алгоритм от этой нерассмотренной вершины, а после возвращаемся и продолжаем перебирать рёбра. Возврат происходит в том случае, если в рассматриваемой вершине не осталось рёбер, которые ведут в нерассмотренную вершину. Если после завершения алгоритма не все вершины были рассмотрены, то необходимо запустить алгоритм от одной из нерассмотренных вершин.

Программа см. Приложение 2

Функция **dfs** – это алгоритм обхода графа в глубину. Она принимает на вход **граф** (словарь с ключами-вершинами и значениями-списками смежных вершин), **стартовую вершину** и (опционально) начальный набор посещённых вершин. Если начальный набор посещённых вершин не был передан, функция создаёт **пустой набор**. Затем функция добавляет **стартовую вершину** в набор посещённых и выводит её на экран. Далее она перебирает все смежные вершины, которые ещё не были посещены, и рекурсивно вызывает функцию **dfs** с очередной вершиной в качестве аргумента. После завершения рекурсивных вызовов функция возвращает итоговый набор посещённых вершин. В примере выше создаётся простой граф с вершинами от **«0»** до **«4»** и несколькими рёбрами между ними. Затем выполняется обход этого графа в глубину с начальной вершиной **«0»**.

Создание приложения

В практической части работы разработано программное обеспечение (ПО) для оптимизации процессов логистики на складе. В качестве алгоритма поиска оптимального маршрута используется алгоритм поиска в ширину. Разработанное ПО обеспечивает возможность ввода начальной точки (A) и конечной точки (B) маршрута, а также предоставляет пользователю информацию о кратчайшем пути между указанными точками в процессе выполнения. Результатом работы программы будет являться отчет о наиболее эффективном маршруте (рис. 3 – рис. 4).

Программа будет использовать алгоритм «поиска в ширину» для определения оптимального маршрута. Файл location.lc (рис. 2) представляет собой двухмерный массив, хранящий информацию о расположении стеллажей и стен на складе. Нулевое значение соответствует пустому пространству, а единичное - стене или стеллажу. Этот файл будет использоваться программой для определения препятствий на пути и поиска оптимального маршрута движения.

```
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0],
[0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0],
[0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0],
[0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0],
[0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0],
[0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0]]
# 0 - empty cell
# 1 - wall
```

рис. 2. Содержание файла "location.lc"

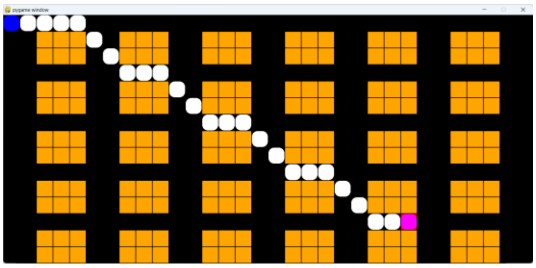

рис. 3. Демонстрация работы программы

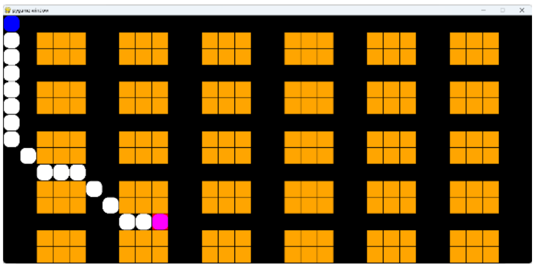

рис. 4. Демонстрация работы программы

Приложение 1

```
from collections import deque

graph = {'A': ['M', 'P'],
        'M': ['A', 'N'],
        'N': ['M', 'B'],
        'P': ['A', 'B'],
        'B': ['P', 'N']}

def bfs(start, goal, graph):
    queue = deque([start])
    visited = {start: None}

    while queue:
        cur_node = queue.popleft()
        if cur_node == goal:
            break

        next_nodes = graph[cur_node]
        for next_node in next_nodes:
            if next_node not in visited:
                queue.append(next_node)
                visited[next_node] = cur_node
    return visited

start = 'A'
goal = 'B'
visited = bfs(start, goal, graph)

cur_node = goal
print(f'\npath from {goal} to {start}: \n {goal} ', end='')
while cur_node != start:
    cur_node = visited[cur_node]
    print(f'---> {cur_node} ', end='')
```

Приложение 2

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)

    print(start)

    for next in graph[start] - visited:
        dfs(graph, next, visited)
    return visited


graph = {'0': set(['1', '2']),
        '1': set(['0', '3', '4']),
        '2': set(['0']),
        '3': set(['1']),
```

```
        '4': set(['2', '3'])}

dfs(graph, '0')
```

Приложение 3

Код программы app.pyw (Код написан на языке Python v3.7.3)

```
import pygame as pg
from collections import deque


def get_rect(x, y):
    return x * TILE + 1, y * TILE + 1, TILE - 2, TILE - 2

def get_next_nodes(x, y):
    check_next_node = lambda x, y: True if 0 <= x < cols and 0 <= y < rows and not grid[y][x] else
False
    ways = [-1, 0], [0, -1], [1, 0], [0, 1], [-1, -1], [1, -1], [1, 1], [-1, 1]
    return [(x + dx, y + dy) for dx, dy in ways if check_next_node(x + dx, y + dy)]

def get_click_mouse_pos():
    x, y = pg.mouse.get_pos()
    grid_x, grid_y = x // TILE, y // TILE
    pg.draw.rect(sc, pg.Color('red'), get_rect(grid_x, grid_y))
    click = pg.mouse.get_pressed()
    return (grid_x, grid_y) if click[0] else False

def bfs(start, goal, graph):
    queue = deque([start])
    visited = {start: None}

    while queue:
        cur_node = queue.popleft()
        if cur_node == goal:
            break

        next_nodes = graph[cur_node]
        for next_node in next_nodes:
            if next_node not in visited:
                queue.append(next_node)
                visited[next_node] = cur_node
    return queue, visited

grid = [[]]

exec(f"grid = {open('location.lc', 'r').read()}")

cols, rows = len(grid[0]), len(grid)
TILE = 50

pg.init()
sc = pg.display.set_mode([cols * TILE, rows * TILE])
```

```
pg.display.set_caption("App Navigator")
clock = pg.time.Clock()

graph = {}
for y, row in enumerate(grid):
    for x, col in enumerate(row):
        if not col:
            graph[(x, y)] = graph.get((x, y), []) + get_next_nodes(x, y)

start = (0, 0)
goal = start
queue = deque([start])
visited = {start: None}

running = True

while running:
    sc.fill(pg.Color('black'))

    [[pg.draw.rect(sc, pg.Color('orange'), get_rect(x, y))
      for x, col in enumerate(row) if col] for y, row in enumerate(grid)]

    mouse_pos = get_click_mouse_pos()
    if mouse_pos and not grid[mouse_pos[1]][mouse_pos[0]]:
        queue, visited = bfs(start, mouse_pos, graph)
        goal = mouse_pos

    path_head, path_segment = goal, goal
    while path_segment and path_segment in visited:
        pg.draw.rect(sc, pg.Color('white'), get_rect(*path_segment), TILE, border_radius=TILE // 3)
        path_segment = visited[path_segment]
    pg.draw.rect(sc, pg.Color('blue'), get_rect(*start), border_radius=TILE // 3)
    pg.draw.rect(sc, pg.Color('magenta'), get_rect(*path_head), border_radius=TILE // 3)

    for event in pg.event.get():
        if event.type == pg.QUIT:
            running = False
    pg.display.flip()
    clock.tick(30)
pg.quit()
```

Список используемых источников

1. https://ya.ru/alisa_davay_pridumaem (19.02.2024 - 20.02.2024)
2. https://ru.wikipedia.org/wiki/Поиск_в_ширину (20.02.2024)
3. https://ru.wikipedia.org/wiki/Поиск_в_глубину (20.02.2024)