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: «Реализация алгоритмов поиска пути в лабиринте»

Выполнил студент 1 курса гр. ИС-34 Амерханов Л.А.

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

Содержание

Введение 3
Задачи 3
Глава 1. Теоретическая часть 4
1.1 Алгоритм поиска в ширину 4
1.2 Алгоритм поиска в глубину 4
1.3 Алгоритм Дейкстры 5
1.4 Алгоритм A* 6
2. Сравнение алгоритмов 7
3. Реализация алгоритмов 9
3.1 Реализация алгоритма Дейкстры 9
3.2 Реализация алгоритма A* 9
Пример работы 11
Заключение 12
Список литературы 13
Приложение 1 14
Листинг программы 14
Приложение 2 19
Визуализация работы программы 19
Приложение 3 20
Представление лабиринта 20

Введение

Цель данной курсовой работы - реализация алгоритма поиска пути в лабиринте. Поиск пути в лабиринте является одной из ключевых задач в области искусственного интеллекта и робототехники, а также находит широкое применение в различных областях, таких как игровая индустрия, автоматизация производства и транспортировка грузов.

В данной работе мы рассмотрим 4 популярных алгоритма поиска пути в лабиринте: алгоритм Дейкстры, алгоритм A*, алгоритм поиска в ширину и в глубину. Будет проведен анализ преимуществ и недостатков алгоритмов, а также оценка их производительности и сложности.

Реализация алгоритма поиска пути в лабиринте будет выполнена на языке программирования Python.

Цель работы: реализовать алгоритмы обхода графа: Дейкстры и A* для задачи поиска маршрута в лабиринте.

Задачи

- Изучить различные 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.

Глава 1. Теоретическая часть

1.1 Алгоритм поиска в ширину

Алгоритм поиска в ширину (BFS) является одним из наиболее распространенных алгоритмов поиска пути в графах. Он использует очередь для хранения узлов, которые нужно посетить, и поэтому является алгоритмом поиска по уровням.

Алгоритм поиска в ширину начинается с выбора начального узла и добавления его в очередь. Затем алгоритм извлекает узел из очереди и добавляет в очередь все его не посещённые соседние узлы. Когда узел помещен в очередь, он помечается как посещенный. Этот процесс продолжается до тех пор, пока очередь не станет пустой.

Каждый раз, когда алгоритм извлекает узел из очереди, он проверяет, является ли этот узел целевым. Если это так, алгоритм завершается, и путь найден. Если же целевой узел не найден, алгоритм продолжает работу, извлекая следующий узел из очереди и продолжая добавлять его соседей.

Алгоритм поиска в ширину гарантирует нахождение кратчайшего пути от начального узла до любого другого узла, если такой путь существует. Он также гарантирует, что, когда узел помещается в очередь, кратчайший путь до этого узла уже был найден [3].

1.2 Алгоритм поиска в глубину

Алгоритм поиска в глубину (DFS) является еще одним распространенным алгоритмом поиска пути в графах. В отличие от алгоритма поиска в ширину, который использует очередь для хранения узлов, алгоритм поиска в глубину использует стек.

Алгоритм поиска в глубину начинается с выбора начального узла и помещения его в стек. Затем алгоритм извлекает узел из стека и добавляет в

стек все его не посещённые соседние узлы. Когда узел помещен в стек, он помечается как посещенный. Этот процесс продолжается до тех пор, пока стек не станет пустым.

Каждый раз, когда алгоритм извлекает узел из стека, он проверяет, является ли этот узел целевым. Если это так, алгоритм завершается, и путь найден. Если же целевой узел не найден, алгоритм продолжает работу, извлекая следующий узел из стека и продолжая добавлять его соседей.

Алгоритм поиска в глубину не гарантирует нахождения кратчайшего пути от начального узла до целевого узла. Это связано с тем, что алгоритм может застрять в локальном максимуме и не искать путь в других направлениях [4].

1.3 Алгоритм Дейкстры

Алгоритм Дейкстры — это алгоритм поиска кратчайшего пути в графе с неотрицательными весами ребер. Он был разработан нидерландским ученым Эдсгером Дейкстрой в 1959 году и на сегодняшний день является одним из наиболее популярных алгоритмов в области теории графов и компьютерной науки [5].

Суть алгоритма заключается в построении дерева кратчайших путей из заданной вершины графа до всех остальных вершин. Алгоритм работает пошагово, на каждом шаге выбирая вершину с минимальным расстоянием от исходной вершины и обновляя расстояния до соседних вершин.

Алгоритм Дейкстры эффективен в случае, когда веса ребер графа неотрицательны, но может давать неверные результаты, если в графе есть ребра отрицательного веса. Если в графе есть ребра отрицательного веса, следует использовать другие алгоритмы [7].

Работа алгоритма продемонстрирована синим цветом. (приложение 2)

1.4 Алгоритм A*

Алгоритм A* — это эвристический алгоритм поиска кратчайшего пути в графе с весами ребер. Он использует оценку расстояния от текущей вершины до целевой вершины для выбора следующей вершины в пути и достигает более высокой производительности, чем алгоритм Дейкстры, за счет использования эвристической оценки [6].

Алгоритм A* работает похожим образом на алгоритм Дейкстры, но добавляет эвристическую функцию, которая оценивает расстояние от текущей вершины до целевой вершины. Эта функция, известная как функция оценки стоимости, помогает алгоритму ориентироваться в нужном направлении и выбирать более перспективные пути.

Одной из самых распространенных функций оценки стоимости является евклидово расстояние между текущей вершиной и целевой вершиной. Однако, в зависимости от конкретной задачи, могут быть использованы и другие функции оценки.

Алгоритм A* является оптимальным, если используется эвристическая функция, удовлетворяющая условию "эвристика никогда не переоценивает действительную стоимость пути". Это означает, что эвристическая функция должна быть допустимой.

Алгоритм A* широко используется в робототехнике, видеоиграх, планировании маршрутов и других областях, где требуется нахождение оптимальных путей в графах [1].

Работа алгоритма выделена желтым цветом. (приложение 2)

2. Сравнение алгоритмов

Алгоритм Дейкстры и алгоритм A* используются для нахождения кратчайшего пути от начальной точки до целевой точки в графе с неотрицательными весами ребер. Однако, алгоритм A* может использовать эвристику для ускорения поиска, учитывая информацию о целевой точке. В случае, когда эвристика хорошо выбрана, A* может быть значительно более эффективным, чем Дейкстры. Алгоритмы поиска в ширину (BFS) и поиска в глубину (DFS) не находят обязательно кратчайший путь, но они могут быть использованы для поиска любого пути в графе. BFS гарантирует нахождение кратчайшего пути в случае, если все ребра имеют одинаковый вес и граф не имеет циклов отрицательного веса. DFS также может быть использован для поиска пути в графе, но за счет использования стека он склонен к поиску вглубь, что может привести к неоптимальному пути, особенно если граф имеет большое количество путей [8].

Кратко сравнивая алгоритмы, можно сказать, что:

Алгоритм A* и Дейкстры находят кратчайший путь в графе с неотрицательными весами ребер. Но A* более эффективен, если выбрана хорошая эвристика.

Алгоритмы BFS и DFS не гарантируют нахождение кратчайшего пути, но могут быть использованы для поиска любого пути в графе. BFS более эффективен для нахождения кратчайшего пути в графе с одинаковыми весами ребер.

A* и Дейкстры работают на основе очереди с приоритетами, BFS работает на основе очереди, а DFS работает на основе стека.

Для более наглядного сравнения, можно составить следующую таблицу:

Характеристика	A*	Дейкстра	BFS	DFS
Гарантия нахождения кратчайшего пути (при условии отсутствия ребер отрицательного веса)	Да	Да	Да	Нет
Временная сложность (в худшем случае)	$O(b^d)$	$O(b^d)$	$O(b^d)$	$O(b^m)$
Память (в худшем случае)	$O(b^d)$	$O(b^d)$	$O(b^d)$	O(bm)
Применимость к различным типам графов	Да	Да	Да	Да
Эвристика	Да	Нет	Нет	Нет
Структура данных	Очередь с приоритетами	Очередь с приоритетами	Очередь	Стек

Здесь b - фактор ветвления графа, d - глубина наименьшего пути до цели, m - общее число ребер в графе. Временная сложность и память в худшем случае зависят от характеристик графа и используемых структур данных.

Из таблицы видно, что алгоритмы A* и Дейкстры обеспечивают гарантию нахождения кратчайшего пути и могут использоваться для графов с разными характеристиками. Однако A* может быть более эффективным, если есть хорошая эвристика. Алгоритмы BFS и DFS не гарантируют нахождения кратчайшего пути, но могут использоваться для поиска любого пути. BFS более эффективен для графов с одинаковыми весами ребер, а DFS склонен к поиску в глубину и может быть менее оптимальным для больших графов.

В зависимости от задачи и характеристик графа, каждый из этих алгоритмов может быть наиболее подходящим выбором.

3. Реализация алгоритмов

3.1 Реализация алгоритма Дейкстры

1. Создайте пустой словарь для хранения расстояний от начальной вершины до всех остальных вершин и инициализируйте его бесконечными значениями. Установите расстояние от начальной вершины до нее самой равным 0.
2. Создайте пустое множество для хранения посещенных вершин.
3. Начните цикл, который будет выполняться до тех пор, пока все вершины не будут посещены.
4. Найдите вершину с минимальным расстоянием до начальной вершины из еще не посещенных вершин. Это можно сделать путем перебора всех вершин и поиска минимального значения.
5. Добавьте найденную вершину в множество посещенных вершин.
6. Обновите расстояния до всех соседних вершин, которые еще не были посещены. Для этого пройдитесь по всем соседним вершинам и обновите их расстояние, если оно больше, чем текущее расстояние до текущей вершины плюс вес ребра между текущей вершиной и соседней вершиной.
7. Повторяйте шаги 4-6, пока все вершины не будут посещены.
8. Верните словарь с расстояниями от начальной вершины до всех остальных вершин [7].

3.2 Реализация алгоритма A*

Алгоритм A* не сильно отличается от алгоритма Дейкстры, зачастую он работает более эффективнее, ведь работает направленно на конечную цель, но это может так же дать обратный результат.

1. Создайте пустой словарь для хранения рассчитанных расстояний от начальной вершины до всех остальных вершин и инициализируйте его

бесконечными значениями. Установите расстояние от начальной вершины до нее самой равным 0.

2. Создайте пустой словарь для хранения родительских вершин.
3. Создайте пустое множество для хранения посещенных вершин.
4. Создайте список открытых вершин и добавьте в него начальную вершину.
5. Начните цикл, который будет выполняться до тех пор, пока все вершины не будут посещены или не будет найден путь до конечной вершины.
6. Из открытых вершин выберите вершину с наименьшей стоимостью, которая равна сумме расстояния от начальной вершины до текущей вершины и эвристической оценки расстояния от текущей вершины до конечной вершины.
7. Если текущая вершина является конечной, то путь найден. В противном случае добавьте текущую вершину в множество посещенных.
8. Обойдите все соседние вершины текущей вершины и обновите их расстояние, если оно больше, чем расстояние от начальной вершины до текущей вершины плюс вес ребра между текущей вершиной и соседней вершиной. Если расстояние было обновлено, то добавьте соседнюю вершину в список открытых вершин и установите ее родительскую вершину.
9. Повторяйте шаги 6-8, пока все вершины не будут посещены или не будет найден путь до конечной вершины.
10. Если путь был найден, то по родительским вершинам можно восстановить путь от начальной вершины до конечной. Верните список вершин, составляющих путь [2].

Пример работы

Можно увидеть визуализацию работы алгоритмов, где желтым цветом помечен путь от входа до ключа, в нем реализован алгоритм Дейкстры, розовым – пути, которые пересекаются: от входа до ключа и от ключа до входа, синим от ключа до входа, в котором реализован алгоритм A*. (приложение 2)

Можно увидеть двумерный массив, состоящих из различных символов, где “#” - стена, “.” – путь от входа до ключа, “;” – места, где путь от входа до ключа и от ключа до входа пересекаются, “,” – путь от ключа до выхода, все остальное – элементы, куда можно двигаться. (приложение 3)

Заключение

В ходе проделанной работы была создана программа, которая при помощи двух алгоритмов пути возвращает результат пути по координатам от точки начала до ключа и от ключа до конца лабиринта.

В результате было разработана эффективная реализация алгоритмов, которые могут применяться не только в лабиринте, а еще и в графо подобных системах, где нужно найти оптимальный путь от точки А до точки Б.

Тем не менее, следует дать оценку для алгоритмов. Алгоритм Дейкстры и A* являются двумя распространенными алгоритмами поиска кратчайшего пути в графах.

Алгоритм Дейкстры работает, обходя все вершины графа, и вычисляет кратчайшее расстояние от начальной вершины до всех остальных вершин. Это делается путем поддержания множества вершин, которые еще не были посещены, и выбора на каждом шаге ближайшей вершины из этого множества.

A* алгоритм является разновидностью алгоритма Дейкстры, который использует эвристику (оценку) расстояния до целевой вершины для оптимизации процесса поиска. Таким образом, A* может находить более оптимальные пути, чем алгоритм Дейкстры, за более короткое время.

Таким образом, если у вас есть информация о местоположении целевой вершины и оценка расстояния до нее, то A* алгоритм будет эффективнее, чем алгоритм Дейкстры. Однако, если такой информации нет, то алгоритм Дейкстры будет более подходящим выбором.

Список литературы

1. A* search algorithm[Электронный ресурс]: Википедия. Свободная энциклопедия. – Режим доступа: https://en.wikipedia.org/wiki/A*_search_algorithm(дата обращения: 25.04.2023).
2. Amit P. Introduction to the A* Algorithm / P. Amit. – Текст: электронный // Red Blob Games: интернет-портал. – URL: https://www.redblobgames.com/pathfinding/a-star/introduction.html(дата обращения 26.04.2023)
3. Breadth-first search [Электронный ресурс]: Википедия. Свободная энциклопедия. – Режим доступа: https://en.wikipedia.org/wiki/Breadth-first_search(дата обращения 28.04.2023)
4. Depth-first search Электронный ресурс]: Википедия. Свободная энциклопедия. – Режим доступа: https://en.wikipedia.org/wiki/Depth-first_search(дата обращения 28.04.2023)
5. Dijkstra's algorithm [Электронный ресурс]: Википедия. Свободная энциклопедия. – Режим доступа: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm(дата обращения: 27.04.2023).
6. Алгоритмы: построение и анализ: учебник / Т. Х. Кормен, Ч. И. Лейзерсон, Р. Л. Ривест, К. Штайн. — 2-е изд. — М.: «Вильямс», 2006. — 1296 с.
7. **Дольников, В. Л.** Основные алгоритмы на графах: текст лекций / В. Л. Дольников, О. П. Якимова. - Ярославль: ЯрГУ, 2011. – 80 с. – URL: http://www.lib.uniyar.ac.ru/edocs/iuni/20110210.pdf(дата обращения 26.04.2023) – Текст: электронный
8. Стаут Б. Алгоритмы поиска пути: Статья. пер: Каменский М., 2000. URL: http://www.oim.ru/reader.asp?nomer=366(дата обращения: 29.04.2023)

Приложение 1
Листинг программы

```
def available_paths(coordsXY, maze):
    LenMazeY = len(maze[0])
    LenMazeX = len(maze)
    coordsX = coordsXY[0]
    coordsY = coordsXY[1]
    possibleWays = []

    if (coordsX - 1) >= 0 and maze[coordsX - 1][coordsY] == " ": #Север
        coord_for_append = (coordsX - 1, coordsY)
        possibleWays.append(coord_for_append)

    if (coordsY + 1) < LenMazeY and maze[coordsX][coordsY + 1] == " ":  #Восток
        coord_for_append = (coordsX, coordsY + 1)
        possibleWays.append(coord_for_append)

    if (coordsX + 1) < LenMazeX and maze[coordsX + 1][coordsY] == " ": #Юг
        coord_for_append = (coordsX + 1, coordsY)
        possibleWays.append(coord_for_append)

    if (coordsY - 1) >= 0 and maze[coordsX][coordsY - 1] == " ": #Запад
        coord_for_append = (coordsX, coordsY - 1)
        possibleWays.append(coord_for_append)
    return possibleWays



def a_star(maze, start, end):
    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 available_paths(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)

    # Если A* не находит путь, функция возвращает попытку построить путь
    return open_list
def dijkstra(maze, start, end):
    open_list = [start]
    closed_list = []
```

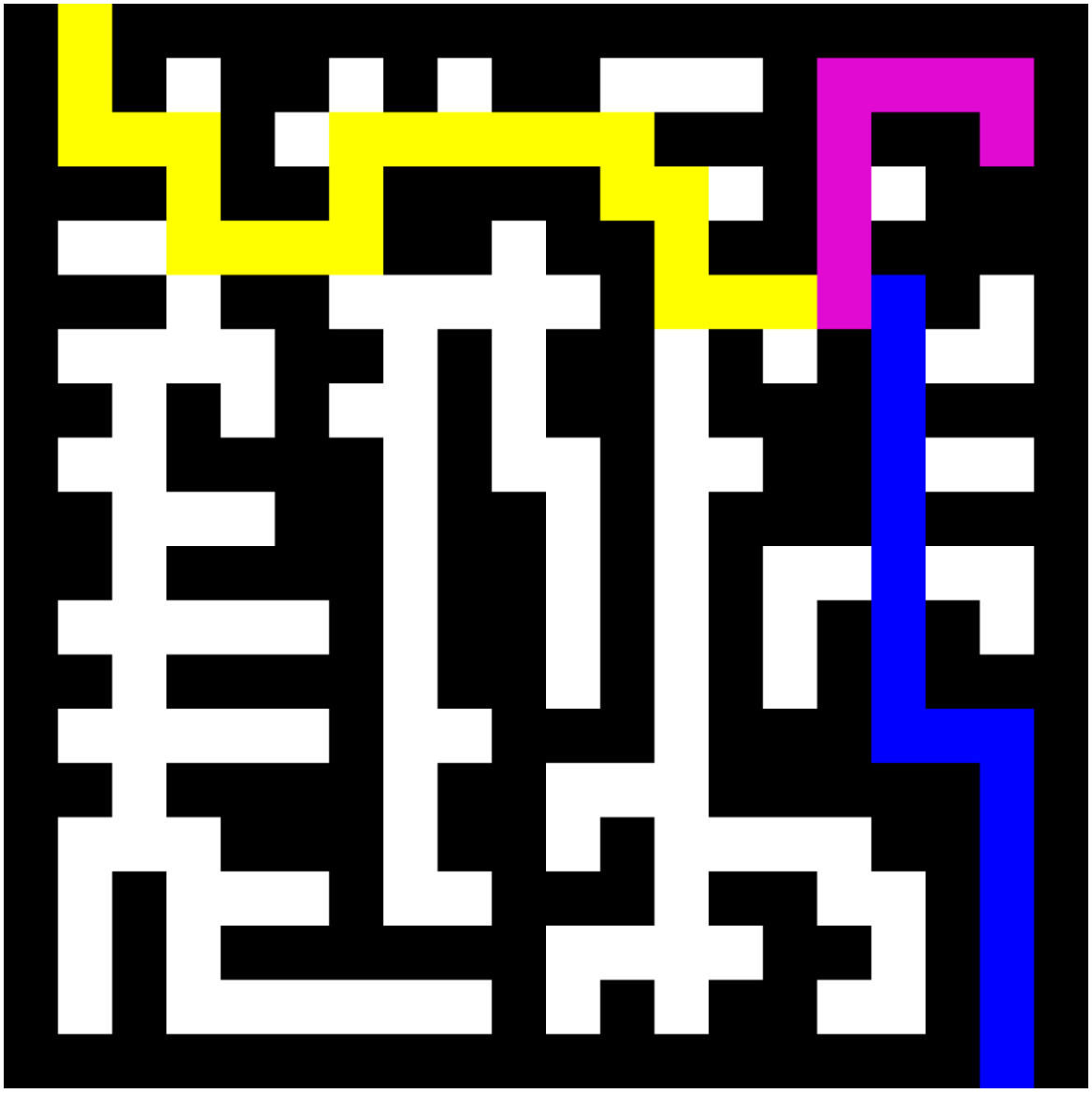
```
    came_from = {}
    g_score = {start: 0}

    while open_list:
        current = min(open_list, key=lambda x: g_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 available_paths(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
                if neighbor not in open_list:
                    open_list.append(neighbor)

    # Если не находит путь, функция возвращает попытку построить путь
    return open_list



def heuristic(current, end):
    return abs(current[0] - end[0]) + abs(current[1] - end[1])
```

```
with open('maze-for-u.txt', 'r') as f:
    maze = [list(line.strip()) for line in f.readlines()]


for Y in range(len(maze[0])):
    if maze[0][Y] == " ":
        start = (0, Y)
        break

for Y in range(len(maze[0])):
    if maze[len(maze) - 1][Y] == " ":
        end = (len(maze) - 1, Y)
        break


for i in range(len(maze)):
    for j in range(len(maze[0])):
        if maze[i][j] == "*":
            key = (i, j)
            maze[i][j] = " "
            break


pathToKey = dijkstra(maze, start, key)
pathToExit = a_star(maze, key, end)

#от старта до ключа  "."
for coords in pathToKey:
    x, y = coords
    maze[x][y] = "."
```

```
#от ключа до выхода  ","
for coords in pathToExit:
    x, y = coords
    if maze[x][y] == ".":
        maze[x][y] = ";"
    else:
        maze[x][y] = ","


# Записываем измененный лабиринт в файл
with open('maze-for-me-done.txt', 'w') as f:
    for line in maze:
        f.write("".join(line) + "\n")
```

Приложение 2

Визуализация работы программы



Приложение 3

Представление лабиринта