Математические алгоритмы поиска кратчайшего пути в программировании

Поиск кратчайшего пути — это важная задача в области математического программирования. Она заключается в нахождении самого короткого (или наиболее выгодного) маршрута в графе, который представляет собой набор объектов (вершин), соединённых линиями (рёбрами).

Эта проблема возникает в различных сферах: от маршрутизации грузов в логистике до построения маршрутов в GPS-навигаторах. В этом докладе мы рассмотрим два популярных алгоритма, которые решают эту задачу: *алгоритм Дейкстры и алгоритм A-стар (A)**. Мы обсудим, как они работают, в каких ситуациях их лучше использовать и приведём примеры их применения.

Также стоит отметить, что вообще такое граф и для чего он нужен.

Граф — это математическая структура, которая состоит из вершин (узлов) и рёбер (связей) между ними. Рёбра могут иметь направление, а также веса — числа, которые обозначают силу связей с вершинами.

Графы используются для представления объектов, их взаимосвязей и отношений между ними. Например, в виде графа можно представить социальную сеть, где вершины — это пользователи, дружеские связи между ними — рёбра, а веса — это, скажем, частота обмена сообщениями. Или например, пример с городами.

Алгоритм Деикстры

Это один из самых известных способов нахождения кратчайшего пути в графе с неотрицательными весами рёбер. Этот алгоритм работает по следующему принципу.

Сначала мы выбираем начальную вершину и присваиваем ей значение расстояния 0, так как путь до самой себя нулевой. Остальным вершинам назначаем значение бесконечности, поскольку мы пока не знаем, насколько далеко они находятся. Затем мы начинаем проверять соседние вершины текущей вершины. Если новый найденный путь к соседу короче, чем известное расстояние, мы обновляем значение расстояния для этого соседа.

Мы продолжаем выбирать вершины с наименьшим расстоянием и обновлять соседей, пока не пройдём все вершины графа. В результате мы получим кратчайшие расстояния от начальной вершины до всех остальных.

Или

'''Алгоритм поддерживает набор посещенных вершин и набор непросмотренных вершин. Он начинается с исходной вершины и итеративно выбирает непросмотренную вершину с наименьшим ориентировочным расстоянием от источника. Затем он посещает соседей этой вершины и обновляет их ориентировочные расстояния, если найден более короткий путь. Этот процесс продолжается до тех пор, пока не будет достигнута конечная вершина или не будут посещены все достижимые вершины.'''

Особенности программирования с использованием алгоритма Деикстры

Алгоритм Дейкстры прост в реализации и очень эффективен для небольших графов. Однако, если у графа есть отрицательные веса рёбер, он не сможет корректно работать. Также алгоритм может работать медленно на больших графах, если не оптимизирован, например, с использованием приоритетной очереди.

Алгоритм A-стар

Алгоритм A-стар — это более продвинутый алгоритм, который сочетает в себе идеи из алгоритма Дейкстры и эвристические подходы. Этот алгоритм не только ищет кратчайшие пути, но и использует оценки, чтобы ускорить процесс поиска.

Эвристика в программировании - это подход, использующий **неформальные методы** для решения задач, которые не всегда могут быть решены алгоритмически или оптимально.

Как и в алгоритме Дейкстры, мы начинаем с начальной вершины и назначаем ей стоимость 0. Затем мы используем две функции для оценки: первая показывает фактическую стоимость пути от начальной вершины до текущей (g(n)), а вторая — эвристическая функция, которая оценивает стоимость пути от текущей вершины до целевой (h(n)). Сумма этих двух значений (f(n) = g(n) + h(n)) позволяет нам определить, какой маршрут исследовать дальше.

Каждый раз, когда мы выбираем следующую вершину, мы можем учесть не только расстояние, но и оценить, насколько близко мы находимся к цели. Это делает алгоритм A-стар более эффективным в поиске пути, особенно в больших графах.

Особенности программирования с использованием алгоритма A-стар

Алгоритм A-стар обладает гибкостью благодаря использованию различных эвристик, которые могут значительно ускорить поиск. Однако выбор неправильной эвристики может привести к потере эффективности или неверным результатам. Важно понимать, что для достижения наилучших результатов эвристика должна быть как можно ближе к реальной стоимости пути.

Преимущества и недостатки алгоритмов

Алгоритм Дейкстры имеет свои плюсы и минусы. К его преимуществам относятся простота реализации и точность, но он может быть неэффективен для больших графов и не поддерживает отрицательные веса. С другой стороны, алгоритм A-стар более гибкий и быстрый благодаря эвристическим функциям, однако он требует большего объёма памяти и правильного выбора эвристики.

Применение алгоритмов на предприятиях и в математическом программировании

Оба алгоритма находят широкое применение в различных отраслях. Например, в логистике они используются для оптимизации маршрутов доставки товаров, что помогает сократить время и снизить затраты. В робототехнике алгоритмы помогают роботам планировать свои движения, избегая препятствий и находя наилучший маршрут к цели. Кроме того, алгоритмы используются в видеоиграх для управления перемещением персонажей, что позволяет создавать более реалистичное поведение.

Применение алгоритмов в реальной жизни

В реальной жизни алгоритмы поиска кратчайшего пути применяются во множестве ситуаций. Например, в GPS-навигаторах они помогают пользователям находить самые быстрые маршруты до заданной точки. Алгоритмы также могут использоваться для оптимизации дорожного движения в городах, чтобы уменьшить пробки и сократить время в пути. В сетях передачи данных алгоритмы позволяют находить оптимальные маршруты для передачи информации, что улучшает скорость и эффективность.

Примеры на коде

Задача 1

Например, рассмотрим задачу на нахождение кратчайших расстояний между городами:

Для работы нам понадобится heapq — один из модулей стандартной библиотеки Python, предназначенный для работы с кучами. Куча — структура данных, где каждый родительский узел имеет значение, меньшее или равное любому из его дочерних элементов.

Внутри функции создаём словарь distances, чтобы хранить расстояния от начальной вершины до всех остальных. Мы также создаём кучу priority_queue для выбора вершин с наименьшим текущим расстоянием.

Далее алгоритм в цикле обновляет расстояния до соседних вершин до тех пор, пока не найдёт кратчайшие расстояния для всех вершин. Результат выводится в виде словаря, где ключи — вершины, а значения — расстояния до них от точки **A**.

Теперь рассчитаем для нашего графа кратчайшие расстояния от **A** до остальных вершин. Граф представим в виде словаря, где ключи — вершины, а значения — словари с соседними вершинами и весами рёбер.

```
import heapq

def dijkstra(graph, start):
    distances = {vertex: float('inf') for vertex in graph}
    distances[start], priority_queue = 0, [(0, start)]

    while priority_queue:
        current_distance, current_vertex =
heapq.heappop(priority_queue)
        if current_distance > distances[current_vertex]:
            continue

        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances

def main():
    # Пример использования:
    graph = {
        'Aksay': {'Birminghem': 4, 'Cambridge': 7},
        'Birminghem': {'Aksay': 4, 'Dover': 2, 'Edinburgh': 8},
        'Cambridge': {'Aksay': 7, 'Dover': 2, 'Edinburgh': 5},
        'Dover': {'Birminghem': 2, 'Cambridge': 2, 'Edinburgh':
1,'Folkestone': 4},
        'Edinburgh': {'Cambridge': 5, 'Dover': 1, 'Folkestone': 11},
        'Folkestone': {'Birminghem': 8, 'Dover': 4, 'Edinburgh': 11}
    }
```

```
    result = dijkstra(graph, 'Aksay')
    print("Кратчайшие расстояния до городов: ")
    for vertex, distance in result.items():
        print(f"До города {vertex}: {distance}")

if __name__ == "__main__":
    main()
```

В этом примере мы создали граф, где вершины соединены рёбрами с определёнными весами. После запуска алгоритма мы получаем кратчайшие расстояния от вершины 'A' до всех остальных вершин.

Алгоритм Дейкстры показал свою эффективность при работе с небольшими графами. Он успешно вычислил кратчайшие расстояния, и его реализация оказалась простой и понятной.

Задача 2

Необходимо вычислить кратчайший путь в графе.

````
Давайте подробно рассмотрим представленный код алгоритма A-star, который используется для поиска кратчайшего пути в графе. Код можно разделить на две основные части: основная функция `a_star` и вспомогательная функция `reconstruct_path`.

### Основная функция: `a_star`

```python
def a_star(graph, start, goal, heuristic):
    open_set = {start}
    came_from = {}
```
1. **Параметры функции**:
   - `graph`: это словарь, представляющий граф, где ключи — это узлы (вершины), а значения — это словари соседей и весов (стоимости) рёбер.
````

   - `start`: начальная вершина, с которой начинается поиск.
   - `goal`: целевая вершина, к которой нужно найти путь.
   - `heuristic`: словарь, который содержит эвристическую оценку расстояний от каждой вершины до целевой.

2. **Инициализация переменных**:
   - `open_set`: множество, которое содержит вершины, которые еще нужно исследовать. Изначально оно содержит только начальную вершину.
   - `came_from`: словарь, который отслеживает, откуда пришёл каждый узел. Это поможет восстановить путь после завершения поиска.

```python
    g_score = {vertex: float('infinity') for vertex in graph}
    g_score[start] = 0
```
3. **Инициализация оценок**:
   - `g_score`: словарь, который содержит наименьшие известные расстояния от начальной вершины до каждой вершины в графе. Изначально все расстояния устанавливаются в бесконечность, кроме начальной вершины, для которой расстояние равно 0.

```python
    f_score = {vertex: float('infinity') for vertex in graph}
    f_score[start] = heuristic[start]
```
4. **Эвристические оценки**:
   - `f_score`: словарь, который содержит оценки стоимости пути от начальной вершины до целевой через каждую вершину. Изначально все значения равны бесконечности, кроме начальной, которая устанавливается в значение её эвристики.

```python
    while open_set:
        current = min(open_set, key=lambda x: f_score[x])
```
5. **Основной цикл поиска**:

   - Цикл продолжается до тех пор, пока есть вершины в `open_set`. Внутри цикла выбирается текущая вершина (`current`) с наименьшим значением `f_score` из всех вершин, находящихся в `open_set`.

```python
        if current == goal:
            return reconstruct_path(came_from, current)
```

6. **Проверка на достижение цели**:
   - Если `current` равен `goal`, значит, мы нашли путь к целевой вершине. В этом случае вызывается функция `reconstruct_path` для восстановления и возврата найденного пути.

```python
        open_set.remove(current)
```

7. **Удаление текущей вершины**:
   - `current` удаляется из `open_set`, так как она больше не будет исследоваться.

```python
        for neighbor, weight in graph[current].items():
            tentative_g_score = g_score[current] + weight
```

8. **Обход соседей**:
   - Для каждого соседа (`neighbor`) текущей вершины `current`, вычисляется временное значение `tentative_g_score`, которое представляет собой сумму расстояния до `current` и веса (стоимости) ребра, соединяющего `current` и `neighbor`.

```python
            if tentative_g_score < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g_score
                f_score[neighbor] = g_score[neighbor] +
heuristic[neighbor]
```

```
                open_set.add(neighbor)
```
9. **Обновление оценок**:
   - Если `tentative_g_score` меньше известного `g_score` для `neighbor`, это значит, что найден более короткий путь до `neighbor`. В этом случае:
     - `came_from[neighbor]` устанавливается на `current`, чтобы отслеживать, откуда пришел `neighbor`.
     - Обновляется `g_score[neighbor]` новым значением `tentative_g_score`.
     - `f_score[neighbor]` пересчитывается как сумма нового `g_score[neighbor]` и эвристической оценки для `neighbor`.
     - `neighbor` добавляется в `open_set` для дальнейшего исследования.

```python
    return []
```
10. **Отсутствие пути**:
    - Если `open_set` опустеет, значит, путь не найден, и функция возвращает пустой список.

Вспомогательная функция: `reconstruct_path`

```python
def reconstruct_path(came_from, current):
    total_path = [current]
    while current in came_from:
        current = came_from[current]
        total_path.append(current)
    return total_path[::-1]
```
1. **Восстановление пути**:
   - Эта функция принимает словарь `came_from` и текущую вершину `current`. Она создает список `total_path`, который начинается с целевой вершины.

   - В цикле `while` восстанавливается путь, поднимаясь по дереву предшествующих вершин, добавляя каждую вершину в `total_path`.
   - Наконец, список разворачивается и возвращается в правильном порядке.

Пример использования

```python
# Пример использования
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5},
    'C': {'A': 4, 'B': 2, 'D': 1},
    'D': {'B': 5, 'C': 1}
}

heuristic = {
    'A': 7,
    'B': 6,
    'C': 2,
    'D': 0
}

print(a_star(graph, 'A', 'D', heuristic))
```

- **Граф**: здесь представлен граф с вершинами и весами рёбер.
- **Эвристика**: каждая вершина имеет эвристическую оценку, которая помогает алгоритму быстрее находить путь.
- **Вызов функции**: `print(a_star(graph, 'A', 'D', heuristic))` запускает алгоритм A-стар для поиска пути от вершины 'A' до вершины 'D' и выводит результат.

Выводы

Этот код демонстрирует, как реализовать алгоритм A-стар для поиска кратчайшего пути в графе с использованием эвристик, что делает его эффективным для решения таких задач. Если есть еще вопросы или требуется уточнение, не стесняйтесь спрашивать!

```
def a_star(graph, start, goal, heuristic):
    open_set = {start}
    came_from = {}

    g_score = {vertex: float('infinity') for vertex in graph}
    g_score[start] = 0

    f_score = {vertex: float('infinity') for vertex in graph}
    f_score[start] = heuristic[start]

    while open_set:
        current = min(open_set, key=lambda x: f_score[x])

        if current == goal:
            return reconstruct_path(came_from, current)

        open_set.remove(current)

        for neighbor, weight in graph[current].items():
            tentative_g_score = g_score[current] + weight

            if tentative_g_score < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g_score
                f_score[neighbor] = g_score[neighbor] +
heuristic[neighbor]
                open_set.add(neighbor)

    return []
```

```
    def reconstruct_path(came_from, current):
        total_path = [current]
        while current in came_from:
            current = came_from[current]
            total_path.append(current)
        return total_path[::-1]

    # Пример использования
    graph = {
        'A': {'B': 1, 'C': 4},
        'B': {'A': 1, 'C': 2, 'D': 5},
        'C': {'A': 4, 'B': 2, 'D': 1},
        'D': {'B': 5, 'C': 1}
    }

    heuristic = {
        'A': 7,
        'B': 6,
        'C': 2,
        'D': 0
    }

    print(a_star(graph, 'A', 'D', heuristic))
```

Задача 3

Необходимо вычислить наиболее короткий путь от одного здания, до другого.

Манхэттенское расстояние (также известное как расстояние городских кварталов) — это расстояние, которое определяется наименьшим числом кварталов, которые придётся пройти, двигаясь из одной точки (перекрёстка) в другую.

Или

Есть одна популярная эвристика **— манхэттенское расстояние.** Её название происходит от системы улиц на острове Манхэттен в Нью-Йорке. Это сеть улиц, которые пересекаются под прямым углом, что делает её похожей на таблицу. Чтобы добраться от одной точки до другой, приходится двигаться вдоль улиц. Если представить эти улицы в виде координатной сетки, мы можем двигаться только вдоль осей X и Y, но не по диагонали.

Если у нас есть две точки с координатами (**x1**, **y1**) и (**x2**, **y2**), то манхэттенское расстояние между ними (M) вычисляется по следующей формуле: M = |4–2| + |3–8| = 2 + 5 = 7.

На каждом шаге алгоритм A* выбирает вершину с наименьшей суммой **g(n) + h(n)**и исследует её соседей. Это продолжается до тех пор, пока алгоритм не достигнет конечной цели.

Попробуем написать A* на Python для графа с эвристикой в виде манхэттенского расстояния. Здесь функция manhattan_distance вычисляет манхэттенское расстояние между двумя точками, а astar реализует алгоритм A* с использованием этого расстояния.

```
import heapq

# Функция для вычисления манхэттенского расстояния между двумя точками
def manhattan_distance(point1, point2):
    return abs(point1[0] - point2[0]) + abs(point1[1] - point2[1])

def astar(grid, start, goal):
    cost = {start: 0}
    came_from = {start: None}
    priority_queue = [(0, start)] # Приоритетная очередь

    while priority_queue:
        current_cost, current_point = heapq.heappop(priority_queue)
        if current_point == goal:
            break
```

```
        for neighbor in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
            x, y = current_point[0] + neighbor[0], current_point[1] +
neighbor[1]

            new_cost = cost[current_point] + 1
            if 0 <= x < len(grid) and 0 <= y < len(grid[0]) and
grid[x][y] == 0:
                if (x, y) not in cost or new_cost < cost[(x, y)]:
                    cost[(x, y)] = new_cost
                    priority = new_cost + manhattan_distance((x, y),
goal)
                    heapq.heappush(priority_queue, (priority, (x, y)))
                    came_from[(x, y)] = current_point
    path, current = [], goal
    while current:
        path.append(current)
        current = came_from[current]
    path.reverse()
    return path

def main():
    grid = [
        [0, 0, 0, 0, 0],
        [0, 1, 1, 0, 0],
        [0, 0, 0, 1, 0],
        [0, 1, 0, 0, 0],
        [0, 0, 0, 0, 0]
    ]

    start, goal = (0, 0), (4, 4)
    result_path = astar(grid, start, goal)

    print("Путь от A до B:")
    for point in result_path:
        print(point)
```

```
if __name__ == '__main__':
    main()
```

Сравнения алгоритмов

- Алгоритм Дейкстры используется для нахождения кратчайшего пути от начальной вершины графа ко всем остальным. Алгоритм A* используется для нахождения кратчайшего пути от начальной вершины к заданной, учитывая эвристическую оценку расстояния.
- Алгоритм Дейкстры рассматривает все вершины равнозначно и всегда выбирает вершину с наименьшим расстоянием до неё. Алгоритм A*, кроме этого, дополнительно использует эвристику. Это делает его более эффективным для больших графов или когда есть информация о приблизительной величине дистанции до цели.
- Алгоритм Дейкстры может быть медленным для больших графов, так как исследует все вершины. Алгоритм A* в больших графах работает быстрее благодаря эвристике, которая позволяет сократить количество рассматриваемых вершин.

Таким образом, алгоритм Дейкстры будет эффективнее там, где нужно найти кратчайшие пути от одной вершины ко всем остальным и нет информации о примерной дистанции до цели. Если есть эвристическая информация, которая может помочь ускорить поиск, то лучшим выбором будет A*.
