Сириус
IT-Колледж

Лекция 7. Кратчайшие пути в графах

Нахождение кратчайших путей из одной вершины в невзвешенных графах, поиск в ширину. Нахождение кратчайших путей из одной вершины в графах с положительными весами, алгоритм Дейкстры.

Кратчайшие пути в графах

Путь в графе между парой вершин V и U - это последовательность вершин $L_i(0 \leq i \leq k)$, удовлетворяющих условиям:

1. $L_0 = V$
2. $L_k = U$
3. для любого i $(0 \leq i \leq k-1)$ вершины L_i и $L_i + 1$ соединены ребром.

Задача о кратчайшем пути — задача поиска самого короткого пути (цепи) между двумя точками (вершинами) на графе, в которой минимизируется сумма весов рёбер, составляющих путь.

В различных постановках задачи, роль длины ребра могут играть не только сами длины, но и время, стоимость, расходы, объём затрачиваемых ресурсов или другие характеристики, связанные с прохождением каждого ребра. Таким образом, задача находит практическое применение в большом количестве областей (информатика, экономика, география и др.).

Кратчайшие пути в графах

Задача о кратчайшем пути в заданный пункт назначения. Требуется найти кратчайший путь в заданную вершину назначения t, который начинается в каждой из вершин графа (кроме t). Поменяв направление каждого принадлежащего графу ребра, эту задачу можно свести к задаче о единой исходной вершине (в которой осуществляется поиск кратчайшего пути из заданной вершины во все остальные).

Задача о кратчайшем пути между заданной парой вершин. Требуется найти кратчайший путь из заданной вершины u в заданную вершину v.

Задача о кратчайшем пути между всеми парами вершин. Требуется найти кратчайший путь из каждой вершины u в каждую вершину v. Эту задачу тоже можно решить с помощью алгоритма, предназначенного для решения задачи об одной исходной вершине, однако обычно она решается быстрее.

Нахождение кратчайших путей из одной вершины в невзвешенных графах, поиск в ширину

Кратчайшие пути в невзвешенных графах

Дан невзвешенный ориентированный граф $G = (V, E)$, а также вершина s. Найти длину кратчайшего пути от s до каждой из вершин графа. Длина пути — количество рёбер в нём.

Кратчайшие пути в невзвешенных графах. Поиск в ширину

Обход в ширину (Поиск в ширину, BFS, Breadth-first search) — один из простейших алгоритмов обхода графа, являющийся основой для многих важных алгоритмов для работы с графами.

Алгоритм работает следующим образом.

1. Создадим массив $dist$ расстояний. Изначально $dist[s] = 0$ (поскольку расстояний от вершины до самой себя равно 0) и $dist[v] = \infty$ для $v \neq s$.
2. Создадим очередь q. Изначально в q добавим вершину s.
3. Пока очередь q непуста, делаем следующее:
   i. Извлекаем вершину v из очереди.
   ii. Рассматриваем все рёбра (v,u)∈E. Для каждого такого ребра пытаемся сделать релаксацию: если dist[v]+1<dist[u], то мы делаем присвоение dist[u]=dist[v]+1 и добавляем вершину u в очередь.

Кратчайшие пути в невзвешенных графах. Поиск в ширину

```
def BFS(self, s):
    visited = [False] * (max(self.graph) + 1)
    queue = []
    queue.append(s)
    visited[s] = True

    while queue:
        s = queue.pop(0)
        print(s, end=" ")

        for i in self.graph[s]:
            if visited[i] == False:
                queue.append(i)
                visited[i] = True
```

Кратчайшие пути в невзвешенных графах. Волновой алгоритм

Алгоритм волновой трассировки (волновой алгоритм, алгоритм Ли) — алгоритм поиска пути, алгоритм поиска кратчайшего пути на планарном графе. Принадлежит к алгоритмам, основанным на методах поиска в ширину.

В основном используется при компьютерной трассировке (разводке) печатных плат, соединительных проводников на поверхности микросхем. Другое применение волнового алгоритма — поиск кратчайшего расстояния на карте в компьютерных стратегических играх.

Волновой алгоритм в контексте поиска пути в лабиринте был предложен Э. Ф. Муром. Ли независимо открыл этот же алгоритм при формализации алгоритмов трассировки печатных плат в 1961 году.

Кратчайшие пути в невзвешенных графах. Волновой алгоритм

Работа алгоритма включает в себя три этапа: **инициализацию**, **распространение волны** и **восстановление пути**.

Во время **инициализации** строится образ множества ячеек обрабатываемого поля, каждой ячейке приписываются атрибуты проходимости/непроходимости, запоминаются стартовая и финишная ячейки.

Далее, от стартовой ячейки порождается шаг в соседнюю ячейку, при этом проверяется, проходима ли она, и не принадлежит ли ранее меченной в пути ячейке.

При выполнении условий проходимости и непринадлежности её к ранее помеченным в пути ячейкам, в атрибут ячейки записывается число, равное количеству шагов от стартовой ячейки, на первом шаге это будет 1. Каждая ячейка, меченная числом шагов от стартовой ячейки, становится стартовой и из неё порождаются очередные шаги в соседние ячейки.

Кратчайшие пути в невзвешенных графах. Волновой алгоритм

```
from collections import deque

def lee_algorithm(matrix, start, end):
    queue = deque()
    visited = set()
    distance = {start: 0}
    prev = {}

    queue.append(start)
    visited.add(start)

    while queue:
        node = queue.popleft()

        for neighbor in get_neighbors(matrix, node):
            if neighbor not in visited:
                visited.add(neighbor)
                distance[neighbor] = distance[node] + 1
                prev[neighbor] = node
                queue.append(neighbor)

            if neighbor == end:
                return get_shortest_path(prev, start, end)

    return None
```

```
def get_neighbors(matrix, node):
    neighbors = []
    row, col = node

    if row > 0 and matrix[row - 1][col] != 1:
        neighbors.append((row - 1, col))

    if row < len(matrix) - 1 and \
            matrix[row + 1][col] != 1:
        neighbors.append((row + 1, col))

    if col > 0 and matrix[row][col - 1] != 1:
        neighbors.append((row, col - 1))

    if col < len(matrix[0]) - 1 and \
            matrix[row][col + 1] != 1:
        neighbors.append((row, col + 1))

    return neighbors

def get_shortest_path(prev, start, end):
    path = []
    node = end

    while node != start:
        path.append(node)
        node = prev[node]

    path.append(start)
    path.reverse()

    return path
```

Кратчайшие пути в невзвешенных графах. Восстановление пути

Пусть теперь заданы 2 вершины s и t, и необходимо не только найти длину кратчайшего пути из s в t, но и восстановить какой-нибудь из кратчайших путей между ними. Всё ещё можно воспользоваться алгоритмом BFS, но необходимо ещё и поддерживать массив предков p, в котором для каждой вершины будет храниться предыдущая вершина на кратчайшем пути.

Кратчайшие пути в невзвешенных графах. Проверка принадлежности вершины кратчайшему пути

Дан ориентированный граф G, найти все вершины, которые принадлежат хотя бы одному кратчайшему пути из s в t.

Запустим из вершины s в графе G BFS — найдём расстояния d_1. Построим транспонированный граф G^T — граф, в котором каждое ребро заменено на противоположное. Запустим из вершины t в графе G^T BFS — найдём расстояния d_2.

Теперь очевидно, что v принадлежит хотя бы одному кратчайшему пути из s в t тогда и только тогда, когда $d_1(v) + d_2(v) = d_1(t)$ — это значит, что есть путь из s в v длины $d_1(v)$, а затем есть путь из v в t длины $d_2(v)$, и их суммарная длина совпадает с длиной кратчайшего пути из s в t.

Кратчайшие пути в невзвешенных графах. Кратчайший цикл в ориентированном графе

Задача: Найти цикл минимальной длины в ориентированном графе.

Попытаемся из каждой вершины найти кратчайший цикл, проходящий через неё, с помощью BFS. Это делается аналогично обычному BFS: мы должны найти расстояний от вершины до самой себя, при этом не считая, что оно равно 0.

Итого, у нас $|V|$ запусков BFS, и каждый запуск работает за $O(|V| + |E|)$. Тогда общее время работы составляет $O(|V|^2 + |V||E|)$. Если инициализировать массив $dist$ единожды, а после каждого запуска BFS возвращать исходные значения только для достижимых вершин, решение будет работать за $O(|V||E|)$.

Сириус
IT-Колледж

Нахождение кратчайших путей из одной вершины в графах с положительными весами, алгоритм Дейкстры

Кратчайшие пути во взвешенных графах. Алгоритм Дейкстры

Дан взвешенный ориентированный граф $G = (V, E)$, а также вершина s. Длина ребра (u, v) равна $w(u, v)$. Длины всех рёбер неотрицательные.
Найти длину кратчайшего пути от s до каждой из вершин графа. Длина пути — сумма длин рёбер в нём.

Алгоритм Дейкстры (Dijkstra’s algorithm) — алгоритм на графах, изобретённый нидерландским учёным Эдсгером Дейкстрой в 1959 году. Находит кратчайшие пути от одной из вершин графа до всех остальных. Алгоритм работает только для графов без рёбер отрицательного веса. Алгоритм широко применяется в программировании, например, его используют протоколы маршрутизации OSPF и IS-IS.

Кратчайшие пути во взвешенных графах. Алгоритм Дейкстры

1. Создать массив $dist$ расстояний. Изначально $dist[s] = 0$ и $dist[v] = \infty$ для $v \neq s$.
2. Создать булёв массив $used$, $used[v] = 0$ для всех вершин v — в нём мы будем отмечать, совершалась ли релаксация из вершины.
3. Пока существует вершина v такая, что $used[v] = 0$ и $dist[v] \neq \infty$, притом, если таких вершин несколько, то v — вершина с минимальным $dist[v]$, делать следующее:
   i. Пометить, что мы совершали релаксацию из вершины v, то есть присвоить $used[v] = 1$.
   ii. Рассматриваем все рёбра $(v, u) \in E$. Для каждого ребра пытаемся сделать релаксацию: если $dist[v] + w(v, u) < dist[u]$, присвоить $dist[u] = dist[v] + w(v, u)$.

Иными словами, алгоритм на каждом шаге находит вершину, до которой расстояние сейчас минимально и из которой ещё не была произведена релаксация, и делает её.

Кратчайшие пути во взвешенных графах. Алгоритм Дейкстры

```
class Graph():
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0] * vertices
                      for _ in range(vertices)]

    def printSolution(self, dist):
        print("Vertex \t Distance from Source")
        for node in range(self.V):
            print(node, "\t\t", dist[node])

    def minDistance(self, dist, sptSet):
        min = 1e7

        for v in range(self.V):
            if dist[v] < min and \
                    sptSet[v] == False:
                min = dist[v]
                min_index = v

        return min_index
```

```
    def dijkstra(self, src):

        dist = [1e7] * self.V
        dist[src] = 0
        sptSet = [False] * self.V

        for cout in range(self.V):

            u = self.minDistance(dist, sptSet)

            sptSet[u] = True

            for v in range(self.V):
                if (self.graph[u][v] > 0 and
                sptSet[v] == False and
                dist[v] > dist[u] + self.graph[u][v]):
                    dist[v] = dist[u] + self.graph[u][v]

        self.printSolution(dist)
```

Кратчайшие пути во взвешенных графах. Алгоритм Дейкстры

Вычислим время работы алгоритма. Мы V раз ищем вершину минимальным $dist$, поиск минимума линейный за $O(V)$, отсюда $O(V2)$. Обработка ребер происходит суммарно за $O(E)$, потому что на каждое ребро мы тратим $O(1)$ действий. Таким образом, финальная асимптотика: $O(V^2 + E)$.

Кратчайшие пути во взвешенных графах. Алгоритм Дейкстры

Искать вершину с минимальным $dist$ можно гораздо быстрее, используя такую структуру данных как очередь с приоритетом. Нам нужно хранить пары $(dist, index)$ и уметь делать такие операции:

- Извлечь минимум (чтобы обработать новую вершину)
- Удалить вершину по индексу (чтобы уменьшить $dist$ до какого-то соседа)
- Добавить новую вершину (чтобы уменьшить $dist$ до какого-то соседа)

Для этого используют, например, кучу или сет. Удобно помимо сета хранить сам массив dist, который его дублирует, но хранит элементы по порядку. Тогда, чтобы заменить значение $(dist1, u)$ на $(dist2, u)$, нужно удалить из сета значение $(dist[u], u)$, сделать $dist[u] = dist2$; и добавить в сет $(dist[u], u)$.

Сириус IT-Колледж

Кратчайшие пути во взвешенных графах. Алгоритм Дейкстры

```python
import heapq

def calculate_distances(graph, starting_vertex):
    distances = {vertex: float('inf') for vertex in graph}
    distances[starting_vertex] = 0

    pq = [(0, starting_vertex)]
    while len(pq) > 0:
        current_distance, current_vertex = heapq.heappop(pq)

        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(pq, (distance, neighbor))

    return distances
```

Кратчайшие пути во взвешенных графах. Алгоритм Дейкстры

Данный алгоритм будет работать за $VO(\log V)$ извлечений минимума и $O(E\log V)$ операций уменьшения расстояния до вершины. Поэтому алгоритм работает за $O(E\log V)$.

Заметьте, что этот алгоритм не лучше и не хуже алгоритма без сета, который работает за $O(V^2+E)$. Ведь если $E=O(V^2)$ (граф почти полный), то Дейкстра без сета работает быстрее, а если, наример, $E=O(V)$, то Дейкстра на сете работает быстрее.