Национальный исследовательский технологический университет «МИСиС»
Институт Информационных технологий и компьютерных наук (ИТКН)

Дисциплина «Комбинаторика и теория графов»

Отчет по теме «Алгоритм Беллмана-Форда построения кратчайших расстояний»

Выполнил:

Студент группы БИВТ 23-8
Хомушку Марк

Ссылка на репозиторий:

https://github.com/greeveus/protoss

Москва 2024

Содержание

Постановка задачи

Алгоритм Беллмана-Форда решает задачу нахождения кратчайших путей от одной исходной вершины до всех остальных вершин взвешенного ориентированного графа, который может содержать ребра с отрицательными весами. Формальная постановка задачи:

Вход:

- Ориентированный граф, где V — множество вершин, E — множество рёбер с весами для каждого ребра
- Исходная вершина s

Выход:

- Кратчайшие расстояния от начальной вершины до каждой вершины, либо сообщение о наличии отрицательного цикла

Теоретическое описание алгоритма Беллмана-Форда

Алгоритм носит имя двух американских учёных: Ричарда **Беллмана** (Richard Bellman) и Лестера **Форда** (Lester Ford). Форд фактически изобрёл этот алгоритм в 1956 г. при изучении другой математической задачи, подзадача которой свелась к поиску кратчайшего пути в графе, и Форд дал набросок решающего эту задачу алгоритма. Беллман в 1958 г. опубликовал статью, посвящённую конкретно задаче нахождения кратчайшего пути, и в этой статье он чётко сформулировал алгоритм в том виде, в котором он известен нам сейчас.

Основная идея

Алгоритм Беллмана-Форда используется для нахождения кратчайших путей в графах, которые могут содержать рёбра с отрицательными весами. Его ключевая особенность — пошаговое улучшение приближений к кратчайшим расстояниям от исходной вершины до всех остальных вершин путём последовательной релаксации рёбер.

Релаксация рёбер

Релаксация — это процесс обновления текущей оценки расстояния до вершины на основе информации о рёбрах графа. Если более короткий путь до вершины найден через ребро, расстояние обновляется. Формально, если для ребра (u,v) вес равен w, то выполняется проверка: $d[v]>d[u]+w$, где $d[u]$ и $d[v]$ — текущие расстояния до вершин u и v, а w — вес ребра. Если условие выполняется, расстояние обновляется: $d[v]=d[u]+w$

Релаксация является основным шагом алгоритма и выполняется последовательно для всех рёбер графа в течение нескольких итераций.

Принцип работы алгоритма

1. На первой итерации вычисляются кратчайшие пути, содержащие только одно ребро.
2. На второй итерации учитываются пути, содержащие два рёбра, и так далее.
3. После $|V|-1$ итерации все кратчайшие пути длиной до $|V|-1$ рёбер будут найдены. Это возможно, так как в ациклическом графе самый длинный путь между двумя вершинами состоит из $|V|-1$ рёбер.

Если после завершения $|V|-1$ итерации выполняется условие $d[v]>d[u]+w$ для какого-либо ребра, это свидетельствует о наличии отрицательного цикла.

Отрицательный цикл в графе — это цикл, сумма весов рёбер которого отрицательна. В графе с отрицательными циклами невозможно корректно определить кратчайшие расстояния до всех вершин, достижимых из этого цикла. Это связано с тем, что при каждом прохождении по циклу сумма расстояний уменьшается, теоретически стремясь к минус бесконечности.

Условия выполнения

- Граф может быть ориентированным или неориентированным.
- Допускаются отрицательные веса рёбер.
- Граф не должен содержать отрицательных циклов, достижимых из исходной вершины. Если такие циклы существуют, алгоритм выявляет их.

Формальная запись алгоритма

1. **Инициализация:**
    - Для всех вершин $v \in V$: $d[v]=\infty$

   - Установить $d[s]=0$, где s — исходная вершина.
2. **Основной цикл:** Для $|V|-1$ итераций: для каждого ребра $(u,v)\in E$: выполнить релаксацию: если для ребра (u,v) вес равен w, то выполняется проверка: $d[v]>d[u]+w$, где $d[u]$ и $d[v]$ — текущие расстояния до вершин u и v, а w — вес ребра. Если условие выполняется, расстояние обновляется: $d[v]=d[u]+w$
3. **Проверка отрицательных циклов:**
   - Для каждого ребра $(u,v)\in E$: если $d[v]>d[u]+w$, граф содержит отрицательный цикл.

Характетристика

Временная сложность

Алгоритм проходит по всем рёбрам $|E|$ для каждой из $|V|-1$ итераций.
Таким образом, временная сложность составляет: $O(|V|\cdot|E|)$

Пространственная сложность

Требуется хранить массив расстояний d размером $O(|V|)$ и список рёбер $O(|E|)$.
Общая сложность — $O(|V|+|E|)$

Сравнительный анализ с алгоритмом Дейкстры и алгоритмом Левита

1. **Алгоритм Дейкстры**

Сравнение возможностей:

- Алгоритм Дейкстры применим только для графов с неотрицательными весами рёбер. Если в графе есть рёбра с отрицательными весами, Дейкстра может возвращать некорректные результаты.
- Алгоритм Беллмана-Форда успешно работает с графами, содержащими отрицательные веса рёбер, и может обнаруживать отрицательные циклы.

Сложность:

- Дейкстра быстрее на практике для графов с неотрицательными весами:
  - Используя массив, сложность O(|V|^2)
  - Используя очередь с приоритетами, сложность O((|V|+|E|)log|V|)
- Беллман-Форд имеет более высокую временную сложность O(|V|·|E|), что делает его менее предпочтительным для больших графов с неотрицательными весами.

Применение:

- Дейкстра — выбор для графов с положительными весами, где требуется высокая производительность.
- Беллман-Форд используется, если есть подозрение на отрицательные веса рёбер или требуется проверка на отрицательные циклы.

Пример: Для графа с 100010001000 вершинами и 500050005000 рёбрами алгоритм Дейкстры с очередью с приоритетами выполнится существенно быстрее, чем Беллман-Форд.

2. **Алгоритм Левита**

Сравнение возможностей:

- Алгоритм Левита предназначен для нахождения кратчайших путей в графах с неотрицательными весами рёбер, как и Дейкстра.
- Левит использует два списка: список вершин на обработку и список обработанных вершин, что делает его подходящим для разреженных графов.

Сложность:

- В худшем случае сложность Левита составляет O(|V|+|E|), что делает его конкурентоспособным с Дейкстрой.
- В отличие от Беллмана-Форда, Левит быстрее на практике в графах с большим количеством рёбер.

Особенности:

- Левит более эффективен, чем Беллман-Форд, для разреженных графов и не работает с графами с отрицательными циклами.

- Для графов с плотной структурой или со смешанными весами рёбер Беллман-Форд предпочтительнее.

Вывод:

- Беллман-Форд обладает универсальностью, так как работает с графами, имеющими отрицательные веса рёбер, но уступает Дейкстре и Левиту по производительности.
- Дейкстра и Левит более эффективны для графов с неотрицательными весами рёбер. Левит выигрывает у Дейкстры в разреженных графах, тогда как Дейкстра лучше на плотных графах.

Перечень инструментов

1. **Язык программирования:** Python версии 3.11
2. **Среда разработки:** IDLE (Python 3.11 64 bit)
3. **Библиотеки:**
    - **sys:** Для работы с бесконечными значениями (float('inf')).

Реализация

```
class Graph:
    def __init__(self, vertices):
        """
        Инициализация графа
        vertices: Количество вершин в графе
        """
        self.V = vertices  # Число вершин
        self.edges = []    # Список рёбер (каждое ребро представляется кортежем (u, v, вес))

    def add_edge(self, u, v, weight):
        """
        Добавляет ребро в граф
        u: Начальная вершина ребра
```

```
        v: Конечная вершина ребра
        weight: Вес ребра
        """
        self.edges.append((u, v, weight))

    def bellman_ford(self, src):
        """
        Реализация алгоритма Беллмана-Форда
        src: Исходная вершина
        return: Список расстояний от src до всех вершин или сообщение о наличии
отрицательного цикла
        """
        # Шаг 1: Инициализация расстояний до всех вершин как бесконечности, кроме
начальной
        d = [float('inf')] * self.V
        d[src] = 0  # Расстояние до исходной вершины равно 0

        # Шаг 2: Релаксация всех рёбер |V| - 1 раз
        for _ in range(self.V - 1):
            for u, v, weight in self.edges:
                # Если расстояние до u не бесконечное и можно улучшить путь до v
                if d[u] != float('inf') and d[u] + weight < d[v]:
                    d[v] = d[u] + weight

        # Шаг 3: Проверка на наличие отрицательных циклов
        for u, v, weight in self.edges:
            if d[u] != float('inf') and d[u] + weight < d[v]:
                return "Граф содержит отрицательный цикл"

        return d


# Создаём граф с 5 вершинами
g = Graph(5)
```

```
# Добавляем рёбра: (u, v, вес)
g.add_edge(0, 1, 10)
g.add_edge(0, 2, 50)
g.add_edge(1, 2, 30)
g.add_edge(1, 3, 40)
g.add_edge(1, 4, 20)
g.add_edge(3, 2, 25)
g.add_edge(3, 1, 15)

# Ребро с отрицательным весом
g.add_edge(4, 3, -10)

# Ребро для отрицательного цикла
#g.add_edge(2, 0, -50)

result = g.bellman_ford(0)

if result == "Граф содержит отрицательный цикл":
    print(result)  # Сообщение о наличии отрицательного цикла
else:
    print("Кратчайшие расстояния от вершины 0:")
    for i, dist in enumerate(result):
        print(f'Вершина {i} : {dist}')
```

Описание реализации

1. **Создание графа:**
    - Для хранения графа используется список рёбер, что упрощает операции релаксации.
2. **Релаксация рёбер:**
    - Цикл по всем рёбрам $|V|-1$ раз для последовательного улучшения расстояний.
3. **Проверка отрицательных циклов:**

- После завершения |V|−1 итераций проверяется, можно ли дополнительно улучшить какое-либо расстояние. Если да — граф содержит отрицательный цикл.

Процесс тестирования

Тестовые случаи:

1. **Положительные веса:** Проверяется корректность расстояний в графах без отрицательных рёбер. Ребра графа (Рис 1) и результат (Рис 2)

```
# Добавляем рёбра: (u, v, вес)
g.add_edge(0, 1, 10)
g.add_edge(0, 2, 50)
g.add_edge(1, 2, 30)
g.add_edge(1, 3, 40)
g.add_edge(1, 4, 20)
g.add_edge(3, 2, 25)
g.add_edge(3, 1, 15)
```

Рис 1

```
Кратчайшие расстояния от вершины 0:
Вершина 0 : 0
Вершина 1 : 10
Вершина 2 : 40
Вершина 3 : 50
Вершина 4 : 30
```

Рис 2

2. **Смешанные веса:** Алгоритм корректно обрабатывает как положительные, так и отрицательные рёбра. Ребра графа (Рис 3) и результат (Рис 4)

```
# Создаём граф с 5 вершинами
g = Graph(5)

# Добавляем рёбра: (u, v, вес)
g.add_edge(0, 1, 10)
g.add_edge(0, 2, 50)
g.add_edge(1, 2, 30)
g.add_edge(1, 3, 40)
g.add_edge(1, 4, 20)
g.add_edge(3, 2, 25)
g.add_edge(3, 1, 15)

# Ребро с отрицательным весом
g.add_edge(4, 3, -10)
```

Рис 3

```
Кратчайшие расстояния от вершины 0:
Вершина 0 : 0
Вершина 1 : 10
Вершина 2 : 40
Вершина 3 : 20
Вершина 4 : 30
```

Рис 4

3. **Отрицательные циклы:** Включение рёбер с отрицательными весами, формирующих отрицательный цикл, должно корректно выявляться алгоритмом. Ребра графа (Рис 5) и результат (Рис 6)

```
# Добавляем рёбра: (u, v, вес)
g.add_edge(0, 1, 10)
g.add_edge(0, 2, 50)
g.add_edge(1, 2, 30)
g.add_edge(1, 3, 40)
g.add_edge(1, 4, 20)
g.add_edge(3, 2, 25)
g.add_edge(3, 1, 15)

# Ребро с отрицательным весом
g.add_edge(4, 3, -10)

# Ребро для отрицательного цикла
g.add_edge(2, 0, -50)
```

Рис 5

Рис 6

Ссылка на репозиторий**:** https://github.com/greeveus/protoss