Пути минимального веса

1 Постановка задачи

Дан взвешенный граф и задана стартовая вершина s. Необходимо найти кратчайшее расстояние из этой вершины до всех остальных. В поставленной задаче не указаны никакие ограничения, однако они появятся для конкретных алгоритмов. Для удобства неизвестные расстояния будем помечать float("inf"). Для нее определены все арифметические операции и не понадобятся дополнительные проверки. Но использование -1 и None, возможно, как это было в поиске в ширину.

2 Алгоритм Дейкстры

2.1 Описание алгоритма

Алгоритм Дейкстры позволяет решить данную задачу при условии, что все ребра имеют неотрицательный вес. Изначально расстояния до всех вершин, кроме стартовой, не известны. До стартовой расстояние 0. Алгоритм можно описать следующими шагами:

1. на каждой итерации алгоритм среди непомеченных вершин выбирает вершину с наименьшим до нее расстоянием;
2. помечает вершину как посещенную;
3. пытается улучшить расстояние до смежных с ней вершин (релаксация).

На каждой итерации поддерживается инвариант, что расстояния до помеченных вершин уже являются кратчайшими и более меняться не будут. Именно по этой причине граф не должен содержать ребра с отрицательным весом, что станет понятно после доказательства данного утверждения. Асимптотика алгоритма сильно зависит от его реализации и будет рассмотрена позже. Сразу стоит упомянуть про восстановление пути. Как и в случае BFS, в алгоритме Дейкстры можно использовать восстановление пути для решения задачи о поиске кратчайшего пути, в не только его длины. Этот подход ничем не отличается от рассмотренного подхода в BFS и подробно рассматриваться не будет.

2.2 Доказательство корректности

Теорема 1. *После того как какая-либо вершина u становится помеченной, текущее расстояние до неё $d[u]$ уже является кратчайшим, и, соответственно, больше меняться не будет.*

Доказательство. Докажем по индукции.

База: на первой итерации вершина s имеет расстояние $d[s] = 0$, что и является кратчайшем для нее расстоянием.

Переход: пусть мы выбрали вершину u. Докажем, что расстояние $d[u]$ является кратчайшим (обозначим через $l[u]$).

Рассмотрим кратчайший путь P до вершины u. Понятно, этот путь можно разбить на два пути: P_1, состоящий только из помеченных вершин (как минимум стартовая вершина s будет в этом пути), и остальная часть пути P_2 (она тоже может включать помеченные вершины, но начинается обязательно с непомеченной). Обозначим через p первую вершину пути P_2, а через q — последнюю вершины пути P_1.

Докажем сначала наше утверждение для вершины p, т.е. докажем равенство $d[p] = l[p]$. Однако это практически очевидно: ведь на одной из предыдущих итераций мы выбирали вершину q и выполняли релаксацию из неё. Поскольку (в силу самого выбора вершины p) кратчайший путь до p равен кратчайшему пути до q плюс ребро (p, q), то при выполнении релаксации из q величина $d[p]$ действительно установится в требуемое значение.

Вследствие неотрицательности стоимостей рёбер длина кратчайшего пути $l[p]$ (а она по только что доказанному равна $d[p]$) не превосходит длины $l[u]$ кратчайшего пути до вершины u. Учитывая, что $l[u] \le d[u]$ (ведь алгоритм Дейкстры не мог найти более короткого пути, чем это вообще возможно), в итоге получаем соотношения: $d[p] = l[p] \le l[u] \le d[u]$

С другой стороны, поскольку и p, и u — вершины непомеченные, то так как на текущей итерации была выбрана именно вершина u, а не вершина p, то получаем другое неравенство: $d[p] \ge d[u]$

Из этих двух неравенств заключаем равенство $d[p] = d[u]$, а тогда из найденных до этого соотношений получаем и: $d[u] = l[u]$, ч.т.д. □

2.3 Базовая реализация

```
if __name__ == "__main__":
    # a - adjacency list
    # each edge is a tuple (target vertex, weight)
    d = [float("inf")] * n
    d[s] = 0
    used == ["False"] * n
    while True:
        u = -1
        for i in range(n):
```

```
            if not used[i] and (u == -1 or d[i] < d[u]):
                u = i
        if u == -1 or d[u] == float("inf)":
            break
        used[u] = True
        for v, w in a[u]:
            d[v] = min(d[v], d[u] + w)
    print(d)
```

Оценим время работы. Внешний цикл выполнится не более, чем N раз, ведь на каждой итерации мы помечаем одну вершину. Так как мы можем выбирать минимум только среди непомеченных вершин, то алгоритм рано или поздно завершится. Поиск минимума выполняется за $O(N)$. Не более, чем M (количество ребер) раз мы обновляем расстояние до вершины. Итого, в приведенном выше варианте время работы $O(N^2 + M) = O(N^2)$.

2.4 Реализация через кучу

Для ускорения алгоритма применяют кучу либо дерево отрезков (мы его в курсе не рассматриваем, но знайте, что куча не единственный способ оптимизации алгоритма). Для обоих структур справедливо, что их высота порядка $O(\log N)$, а значит и операции с ними работают за это время. Благодаря им операция извлечения минимума работает за $O(\log N)$ вместо $O(N)$. Но при этом возрастает время работы операции обновления расстояния до вершины: с $O(1)$ до $O(\log N)$. Получается, что мы не более, чем N раз выполняем операцию извлечения минимума и не более, чем M (количество ребер) раз мы обновляем расстояние до вершины. Таким образом исходная асимптотика $O(N^2 + M) = O(N^2)$ меняется на $O(N \log N + M \log N) = O((N + M) \log N)$ или $O(M \log N)$.

Ниже приведен пример реализации алгоритма Дейкстры с кучей. Заметим, что операция добавления элемента в кучу нам не нужна, ее мы можем построить просто при помощи list comprehension. Но у нас появилась новая операция, изменение приоритета у элемента, которую в общем случае куча не поддерживает. Для того, чтобы добавить поддержку операции, необходимо знать местоположение элемента в куче. Выполнять поиск элемента за линию просто ухудшит изначальную асимптотику. Поэтому заведем массив v2h, в котором будет поддерживаться актуальное положение каждого элемента в куче. И при изменении кучи меняется и он.

```
def sift_up(heap, i):
    while i > 0 and d[heap[(i - 1) // 2]] > d[heap[i]]:
        v2h[heap[i]], v2h[heap[(i - 1) // 2]] = (i - 1) // 2, i
        heap[i], heap[(i - 1) // 2] = heap[(i - 1) // 2], heap[i]
        i = (i - 1) // 2
```

```
def sift_down(heap, i):
    n = len(heap)
    while i * 2 + 1 < n:
        j = i
        if d[heap[i]] > d[heap[i * 2 + 1]]:
            j = i * 2 + 1
        if i * 2 + 2 < n and d[heap[j]] > d[heap[i * 2 + 2]]:
            j = i * 2 + 2
        if i == j:
            break
        v2h[heap[i]], v2h[heap[j]] = j, i
        heap[i], heap[j] = heap[j], heap[i]
        i = j


def extract_min(heap):
    x = heap[0]
    heap[0] = heap.pop()
    v2h[heap[0]] = 0
    sift_down(heap, 0)
    return x


if __name__ == "__main__":
    # a - adjacency list
    # each edge is a tuple (target vertex, weight)
    d = [float("inf")] * n
    d[s] = 0
    heap = [s] + [i for i in range(n) if i != s]
    v2h = [0] * n
    for i, j in enumerate(heap):
        v2h[j] = i
    while heap:
        u = extract_min(heap)
        if d[u] == float("inf"):
            break
        for v, w in a[u]:
            if d[v] > d[u] + w:
                d[v] = d[u] + w
                sift_up(heap, v2h[v])
    print(d)
```

3 Алгоритм Форда-Беллмана

3.1 Описание алгоритма

Алгоритм Форда-Беллмана, как и алгоритм Дейкстры, используется для поиска кратчайшего расстояния от одной вершины до остальных. Он является типичным алгоритмом ДП. Состояния описываются двумя параметрами (i, j) и означают "длину кратчайшего пути, проходящего не более, чем по i ребрам, и заканчивающегося в вершине j". Алгоритм реализуется двумя вложенными циклами. Первый цикл перебирает значения первого параметра, т.е. количество ребер в кратчайших расстояниях. Одну итерацию этого цикла назовем фазой алгоритма. i-ая фаза алгоритма выглядит следующим образом:

1. перебираем все ребра (u, v) в графе (считаем, что ребра ориентированы);
2. пытаемся улучшить расстояние $d[i][v]$ до вершины v, сравнив с $d[i-1][u] + w(u, v)$.

Неизвестным остается количество фаз алгоритма, что мы оценим позже. Сам алгоритм имеет несколько особенностей:

1. алгоритм работает корректно даже при наличии ребер отрицательного веса, -1 – валидное значение для расстояний, поэтому массив d инициализировать -1 нельзя;
2. алгоритм работает корректно при наличии циклов отрицательного веса (сумма весов ребер цикла отрицательна) в и позволяет искать их.

3.2 Оценка количества фаз

Для доказательства следующих теорем мы будем считать, что веса ребер могут быть только положительными.

Теорема 2. *Любой кратчайший путь в графе с ребрами положительного веса не содержит циклов.*

Доказательство. Докажем от противного. Пусть мы нашли такой кратчайший путь, который содержит цикл. Для конкретности обозначим за s и t начало и конец пути соотвественно. Не теряя общности, считаем, что в нашем пути содержится только один цикл, и мы прошли по нему ровно один раз. Тогда наш путь выглядит как $(p_0 = s, p_1, \dots p_i = u, \dots p_j = u, \dots p_k = t)$. Так как этот путь кратчайший, то путь от s до $p_i = u$ является кратчайшим и путь s до $p_j = u$ является кратчайшим. Тогда кратчайший путь из u в u проходит по циклу, но это не возможно, так как все веса ребер положительные и длина пути > 0. Тогда кратчайший путь из u в u не

содержит ребер, следовательно $i = j$ и путь из s в t не содержит циклов. Пришли к противоречию. □

Следствие 2.1. *Кратчайший путь между двумя вершинами содержит не более $N - 1$ ребер, где N – количество вершин в графе.*

Наличие ребер нулевого веса слегка меняет ситуацию. У нас может быть цикл из u в u, состоящий из ребер нулевого веса. Но тогда проход по этому циклу ничего не меняет и его удаление из кратчайшего пути не увиличивает его. Поэтому циклы нулевого веса никогда не будут присутствовать в найденных алгоритмом кратчайших путях. Теперь оценим количество фаз алгоритма.

Теорема 3. *После выполнения i фаз алгоритм Форда-Беллмана корректно находит все кратчайшие пути, длина которых (по числу рёбер) не превосходит i.*

Иными словами, для любой вершины u обозначим через k число рёбер в кратчайшем пути до неё (если таких путей несколько, можно взять любой). Тогда это утверждение говорит о том, что после k фаз этот кратчайший путь будет найден гарантированно.

Доказательство. Рассмотрим произвольную вершину u, до которой существует путь из стартовой вершины s, и рассмотрим кратчайший путь до неё: $(p_0 = s, p_1, \ldots, p_k = u)$. Перед первой фазой кратчайший путь до вершины $p_0 = s$ найден корректно. Во время первой фазы ребро (p_0, p_1) было просмотрено алгоритмом Форда-Беллмана, следовательно, расстояние до вершины p_1 было корректно посчитано после первой фазы. Повторяя эти утверждения k раз, получаем, что после k-й фазы расстояние до вершины $p_k = u$ посчитано корректно, ч.т.д. □

Из следствия 2.1 и теоремы 3 следует, что алгоритм Форда-Беллмана требует не более $N - 1$ фаз для вычисления кратчайших расстояний до всех вершин.

3.3 Реализация алгоритма

При реализации алгоритма Форда-Беллмана удобней всего использовать список ребер. Базовая реализация алгоритма выглядит следущим образом:

```
if __name__ == "__main__":
    # edges - `edge list` for directed edges (arcs)
    d = [[float("inf")] * n for _ in range(n)]
    d[0][s] = 0
    for i in range(1, n):
        for u, v, w in edges:
            d[i][v] = min(d[i][v], d[i-1][u] + w)
    print(d[n-1])
```

Отсюда видно, что асимптотика алгоритма $O(NM)$. Приведенный алгоритм можно немного оптимизировать. Первая оптимизация позволяет улучшить время работы алгоритма в среднем. Из доказанных теорем понятно, что выполнять лишние фазы алгоритма бессмысленно. Тогда мы можем проверять, произошли ли изменения в расстояниях до вершин на каждой фазе. Если на какой-то фазе изменений не было, значит мы нашли все расстояния и можно завершать алгоритм. Вторая оптимизация позволяет сэкономить память. На самом деле нам не надо хранить веса для всех фаз алгоритма, достаточно для последней фазы. В таком случае мы вообще схлопнем массив d до одномерного массива d', и динамика будет $d'[v] = \min(d'[v], d'[u] + w(u, v))$.

```
if __name__ == "__main__":
    # edges - `edge list` for directed edges (arcs)
    d = [float("inf")] * n
    d[s] = 0
    for _ in range(1, n):
        changed = False
        for u, v, w in edges:
            if d[v] > d[u] + w:
                d[v] = d[u] + w
                changed = True
        if not changed:
            break
    print(d)
```

Также, как в алгоритмах Дейкстры и BFS, в алгоритме Форда-Беллмана можно реализовать восстановление пути. Реализация аналогична упомянутым алгоритмам.

3.4 Поиск цикла отрицательного веса

Ранее мы доказали, что кратчайший путь не содержит циклов положительного веса, а циклы нулевого веса роли не играют. Но циклы отрицательного веса все меняют. В таком цикле растояния до вершин будут каждый раз уменьшаться, если мы будем по нему гулять. Таким образом нам вообще не выгодно его заканчивать, а значит мы можем считать, что кратчайшие расстояния до этих вершин будут $-\infty$. В связи с этим $N - 1$ фаз не хватит чтобы посчитать кратчайшие расстояния. Получаем критерий наличия цикла отрицательного веса: если на N-ой фазе происходит обновление расстояний, в графе существует цикл отрицательного веса.

Модифицируем алгоритм, используя выведенный критерий наличия искомого цикла. Для начала мы увеличим количество итераций до N. Далее необходимо запомнить номер любой вершины, которая обновилась на последней итерации. Так как, эта вершина обновилась, то она либо лежит в цикле, либо достижима из него. Первым шагом необходимо найти

вершину, которая гарантированно лежит на цикле. Для этого используем массив p, необходимый для восстановления пути. Используя массив, откатимся ровно N раз назад. Таким образом мы гарантировано окажемся в цикле. Пусть текущая вершина u. После этого пройдемся по циклу, также откатываясь назад, пока не вернемся в u. Полученный цикл - цикл отрицательного веса.

```
if __name__ == "__main__":
    #edges - `edge list` for directed edges (arcs)
    d = [float("inf")] * n
    d[s] = 0
    p = [-1] * n
    for _ in range(n):
        # now it stores number of vertex u,
        # for which d[u] has changed
        changed = -1
        for u, v, w in edges:
            if d[v] > d[u] + w:
                d[v] = d[u] + w
                changed = v
                p[v] = u
            if changed == -1:
                break

    if changed == -1:
        print("No negative cycle")
        exit(0)

    for _ in range(n):
        changed = p[changed]

    path = [changed]
    cur = changed
    while changed != p[cur]:
        cur = p[cur]
        path.append(cur)
    print(*path[::-1])
```

4 Алгоритм Флойда-Уоршелла

4.1 Описание алгоритма

Данный алгоритм отличается от рассмотренных ранее алгоритмов в постановке задачи. Он позволяет решать задачу поиска кратчайших расстояний между всеми парами вершин. Ребра в графе могут быть отрица-

тельными, однако циклов отрицательного веса быть не должно.

Сам по себе алгоритм достаточно простой. Он перебирает все возможные тройки вершин i, j, k. Если существуют пути из i в k и из k в j, то алгоритм пытается улучшить путь из i в j, проложив его через k, т.е. $d(i,j) = \min(d(i,j), d(i,k) + d(k,j))$. Уже из описания алгоритма понятно, что его асимптотика $O(N^3)$.

Перебор k происходит во внешнем цикле. После каждой итерации этого цикла известны кратчашие расстояния между всеми парами вершин, проходящие только через вершины из множества $\{1 \dots k\}$ (не считая начальную и конечную вершины). Тогда после выполнения всех итераций внешнего цикла будут звестны кратчашие расстояния между всеми парами вершин, проходящие только через вершины из множества $V = \{1 \dots N\}$.

4.2 Реализация алгоритма

Здесь будет приведена сразу оптимизированная реализация. При реализации в лоб массив d должен хранить ответы для всех итераций внешнего цикла, массив d будет трехмерным. Однако можно обойтись без этой размерности. В самом деле, если мы улучшили (уменьшили) какое-то значение в матрице расстояний, мы не могли ухудшить тем самым длину кратчайшего пути для каких-то других пар вершин, обработанных позднее.

Изначально массив d выглядит следующим образом:

- на главной диагонали стоят нули;
- $d[i][j] = w[i][j]$ если есть ребро (если есть кратные ребра, выбирайте наименьшее);
- $d[i][j] = +\infty$, иначе.

```
if __name__ == "__main__":
    # d initialized from input
    for k in range(n):
        for i in range(n):
            for j in range(n):
                d[i][j] = min(d[i][j], d[i][k] + d[k][j])
    print(d)
```

4.3 Циклы отрицательного веса

Если в графе есть циклы отрицательного веса, то формально алгоритм Флойда-Уоршелла неприменим к такому графу. На самом же деле, для тех пар вершин i и j, между которыми нельзя зайти в цикл отрицательного вес, алгоритм отработает корректно. Для тех же пар вершин, ответа для которых не существует (по причине наличия отрицательного

цикла на пути между ними), алгоритм Флойда найдёт в качестве ответа какое-то число (возможно, сильно отрицательное, но не обязательно). Тем не менее, можно улучшить алгоритм Флойда, чтобы он аккуратно обрабатывал такие пары вершин и выводил для них, например, $-\infty$. Для этого можно сделать, например, следующий критерий "не существования пути". Итак, пусть на данном графе отработал обычный алгоритм Флойда. Тогда между вершинами i и j не существует кратчайшего пути тогда и только тогда, когда найдётся такая вершина k, достижимая из i и из которой достижима j, для которой выполняется $d[k][k] < 0$.

```
if __name__ == "__main__":
    # d initialized from input
    for k in range(n):
        for i in range(n):
            for j in range(n):
                d[i][j] = min(d[i][j], d[i][k] + d[k][j])

    for k in range(n):
        for i in range(n):
            for j in range(n):
                if d[i][k] < float("inf") and d[k][k] < 0 and \
                        d[k][j] < float("inf"):
                    d[i][j] = float("-inf");
    print(d)
```