

Алгоритмы обхода графов

1 Поиск в глубину

Одним из алгоритмов обхода является поиск в глубину (depth-first search, dfs). Он заключается в последовательности следующих действий:

1. Заходит в вершину u и помечает ее, как посещенную;
2. Перебирает все исходящие ребра из вершины u ;
3. Если вершина, в которую ведет ребро, еще не посещена, dfs переходит в нее, иначе пропускает;
4. Если ребер больше нет, dfs выходит из вершины u .

По факту, dfs идет по какому-то одному пути, пока не упрется в тупик. Далее он возвращается на шаг назад и пытается пойти по другому пути. Так будет продолжаться, пока все вершины, доступные из какой-то стартовой вершины, не будут посещены. Наиболее часто dfs пишется рекурсивно, хотя и возможны итеративные реализации. Базовая реализация алгоритма с использованием матрицы смежности выглядит следующим образом:

```
def dfs(u):
    used[u] = True
    for v in range(n):
        if a[u][v] == 1 and not used[v]:
            dfs(v)

if __name__ == "__main__":
    # Read graph:
    # a - adj. matrix
    # n - number of vertices
    used = [False] * n
    for i in range(n):
        if not used[i]:
            dfs(i)
```

Примечание: здесь и далее для удобства и сокращения кода я использую глобальные переменные (массивы a и $used$) внутри функции, что не является хорошей практикой со стороны дизайна кода.

Асимптотика алгоритма $O(N + M)$. Легко заметить, что dfs заходит в каждую вершину не более одного раза. Кроме того, в процессе работы алгоритма просматриваются все ребра. Отсюда и получается такая

асимптотика. Хотя в приведенной выше реализации время работы будет $O(N^2)$, так как при использовании матрицы смежности перебор всех ребер для одной вершины работает за $O(N)$. Такая же оценка будет справедлива для графов, близких к полным. В полном графе количество ребер $M = \frac{N(N-1)}{2} = O(N^2)$, не учитывая петли и кратные ребра.

В чистом виде данный алгоритм просто выполняет обход графа. Обычно данный код применяется с некоторым набором модификаций для решения той или иной задачи. На самом деле этот алгоритм уже встречался при решении задач по динамике при помощи рекурсии с мемоизацией. Ведь состояния – вершины графа, а переходы – ребра. Кроме этого, dfs используют для решения следующих задач:

- проверка ацикличности графа и поиск циклов,
- подсчет кол-ва компонент связности,
- поиск компонент сильной связности,
- поиск мостов и точек сочленения,
- топологическая сортировка графа,
- проверка графа на двудольность,
- выполнение этапа препроцессинга для других алгоритмов.

Рассмотрим подробнее некоторые применения. Отсюда и далее предполагаем, что граф мы храним в виде матрицы смежности.

1.1 Количество компонент связности

Обычно dfs запускается из какой-нибудь вершины. Как только алгоритм посетит все вершины, обход заканчивается. Но, если граф не является связным, то dfs не сможет посетить все вершины. А именно, он посетит только те вершины, до которых можно добраться из стартовой вершины. Таким образом за один обход dfs полностью пройдет по одной компоненте связности. Отсюда, количество компонент связности – количество запусков обхода в глубину.

```
def dfs(u):
    used[u] = True
    for v in range(n):
        if a[u][v] == 1 and not used[v]:
            dfs(v)

if __name__ == "__main__":
    # Read graph:
```

```

# a - adj. matrix
# n - number of vertices
used = [False] * n
components = 0
for i in range(n):
    if not used[i]:
        components += 1
        dfs(i)
print(components)

```

1.2 Поиск циклов в неориентированном графе

Тут все просто. Будем считать, что все вершины покрашены в белый цвет. Когда dfs заходит в вершину, он ее красит в черный цвет. Соответственно, когда мы из черной вершины найдем ребро в черную вершину, то мы найдем цикл. Единственное, что надо исключить, это тривиальные циклы, которые в неориентированном графе всегда присутствуют.

```

def dfs(u, prev):
    # 'prev' variable shows previous vertex
    # we need it to avoid trivial cycles
    used[u] = True
    for v in range(n):
        if a[u][v] == 1:
            if not used[v]:
                dfs(v, u)
            elif v != prev:
                # we found a cycle

if __name__ == "__main__":
    # Read graph:
    # a - adj. matrix
    # n - number of vertices
    used = [False] * n
    for i in range(n):
        if not used[i]:
            dfs(i, -1)

```

1.3 Поиск циклов в ориентированном графе

Для ориентированного графа чуть посложнее, двухцветная раскраска тут не поможет (предлагается самостоятельно найти пример, когда вышеприведенный код будет работать неверно). Будем использовать трехцветную раскраску:

- 0 – белая вершина, еще не посещалась,
- 1 – серая вершина, dfs прошел через вершину, но не вышел,
- 2 – черная вершина, dfs закончил ее обработку.

Приведенный ниже пример проверяет граф на ацикличность. Если граф ацикличен, то программа выводит YES, иначе – NO.

```
def dfs(u):
    color[u] = 1
    for v in range(n):
        if a[u][v] == 1:
            if color[v] == 0:
                dfs(v)
            elif color[v] == 1:
                # we found a cycle
                print("NO")
                exit(0)
    color[u] = 2

if __name__ == "__main__":
    # Read graph:
    # a - adj. matrix
    # n - number of vertices
    color = [0] * n
    for i in range(n):
        if color[i] == 0:
            dfs(i)
    print("YES")
```

1.4 Топологическая сортировка

Пусть есть ориентированный ациклический граф. Необходимо упорядочить вершины так, что если из вершины u в вершину v есть путь, то вершина v стоит позже, чем вершина u . Из постановки задачи понятно, что если в графе есть цикл, то упорядочить вершины, образующие цикл, не возможно.

Вариант решения этой задачи при помощи dfs предложил Тарьян. Отсюда этот алгоритм называют алгоритмом Тарьяна, однако это название относится еще к двум алгоритмам, поэтому стоит уточнять.

```
def dfs(u):
    used[u] = True
    for v in range(n):
        if a[u][v] == 1 and not used[v]:
```

```

        dfs(v)
    order.append(u)

if __name__ == "__main__":
    # Read graph:
    # a - adj. matrix
    # n - number of vertices
    used = [False] * n
    order = [] # Vertex order
    for i in range(n):
        if not used[i]:
            dfs(i)
    print(order[::-1]) # We need to reverse it

```

1.5 Поиск компонент сильной связности

С поиском компонент сильной связности немного сложнее. Для решения данной задачи воспользуемся алгоритмом Косараю, который представляет собой две серии поисков в глубину. Отсюда асимптотика алгоритма $O(N + M)$.

Введем пару обозначений. **Время выхода** $\text{tout}[C]$ из компоненты C сильной связности есть максимум из значений $\text{tout}[v], \forall v \in C$. **Время входа** $\text{tin}[C]$ в компоненту C сильной связности есть минимум из значений $\text{tin}[v], \forall v \in C$. Понятно, что компоненты сильной связности для графа не пересекаются, т.е. фактически это разбиение всех вершин графа. Отсюда логично определение **графа конденсации** G^{SCC} как графа, получаемого из исходного графа сжатием каждой компоненты сильной связности в одну вершину. Каждой вершине графа конденсации соответствует компонента сильной связности графа G , а ориентированное ребро между двумя вершинами C_i и C_j графа конденсации проводится, если найдётся пара вершин $u \in C_i, v \in C_j$, между которыми существовало ребро в исходном графе, т.е. $(u, v) \in E$. Важнейшим свойством графа конденсации является то, что он **ацикличесен** (попробуйте сами доказать данное свойство).

Теорема. Пусть C и C' – две различные компоненты сильной связности, и пусть в графе конденсации между ними есть ребро (C, C') . Тогда $\text{tout}[C] > \text{tout}[C']$.

При доказательстве возникает два принципиально различных случая в зависимости от того, в какую из компонент первой зайдёт обход в глубину, т.е. в зависимости от соотношения между $\text{tin}[C]$ и $\text{tin}[C']$:

- Первой была достигнута компонента C . Это означает, что в какой-то момент времени обход в глубину заходит в некоторую вершину v компоненты C , при этом все остальные вершины компонент C и C' ещё не посещены. Но, т.к. по условию в графе конденсаций есть

ребро (C, C') , то из вершины v будет достижима не только вся компонента C , но и вся компонента C' . Это означает, что при запуске из вершины v обход в глубину пройдёт по всем вершинам компонент C и C' , а, значит, они станут потомками по отношению к v в дереве обхода в глубину, т.е. для любой вершины $u \in C \cup C', u \neq v$ будет выполнено $\text{tout}[v] > \text{tout}[u]$, ч.т.д.

- Первой была достигнута компонента C' . Опять же, в какой-то момент времени обход в глубину заходит в некоторую вершину $v \in C'$, причём все остальные вершины компонент C и C' не посещены. Поскольку по условию в графе конденсаций существовало ребро (C, C') , то, вследствие ацикличности графа конденсаций, не существует обратного пути C' в C , т.е. обход в глубину из вершины v не достигнет вершин C . Это означает, что они будут посещены обходом в глубину позже, откуда и следует $\text{tout}[C] > \text{tout}[C']$, ч.т.д.

Доказанная теорема является основой алгоритма поиска компонент сильной связности. Из неё следует, что любое ребро (C, C') в графе конденсаций идёт из компоненты с большей величиной tout в компоненту с меньшей величиной.

На первом шаге алгоритма мы для каждой вершины будем считать время выхода $\text{tout}[v]$. Если мы отсортируем все вершины $v \in V$ в порядке убывания времени выхода $\text{tout}[v]$, то первой окажется некоторая вершина u , принадлежащая "корневой" компоненте сильной связности, т.е. в которую не входит ни одно ребро в графе конденсаций. Теперь нам хотелось бы запустить такой обход из этой вершины u , который бы посетил только эту компоненту сильной связности и не зашёл ни в какую другую; научившись это делать, мы сможем постепенно выделить все компоненты сильной связности: удалив из графа вершины первой выделенной компоненты, мы снова найдём среди оставшихся вершину с наибольшей величиной tout , снова запустим из неё этот обход, и т.д.

Чтобы научиться делать второй шаг алгоритма, рассмотрим транспонированный граф G^T , т.е. граф, полученный из G изменением направления каждого ребра на противоположное. Нетрудно понять, что в этом графе будут те же компоненты сильной связности, что и в исходном графе. Более того, граф конденсации $(G^T)^{\text{SCC}}$ для него будет равен транспонированному графу конденсации исходного графа G^{SCC} . Это означает, что теперь из рассматриваемой нами "корневой" компоненты уже не будут выходить рёбра в другие компоненты. Таким образом, чтобы обойти всю "корневую" компоненту сильной связности, содержащую некоторую вершину u , достаточно запустить обход из вершины u в графе G^T . Этот обход посетит все вершины этой компоненты сильной связности и только их. Как уже говорилось, дальше мы можем мысленно удалить эти вершины из графа (позначить их посещёнными), находить очередную вершину с максимальным значением $\text{tout}[v]$ и запускать обход на транспонированном графе из неё, и т.д.

Стоит отметить, что сортировка вершин по времени выхода по факту решает задачу топологической сортировки графа. Отсюда, первым шагом алгоритма будет применение алгоритма Тарьяна для решения задачи топологической сортировки.

В приведенной реализации:

- `dfs1()` – dfs для первого шага, алгоритм Тарьяна;
- `dfs2()` – dfs для второго шага;
- `a` – матрица смежности графа G ;
- `ar` – матрица смежности графа G^T .

```
def dfs1(u):
    used[u] = True
    for v in range(n):
        if a[u][v] == 1 and not used[v]:
            dfs1(v)
    order.append(u)

def dfs2(u):
    used[u] = True
    component.append(u)
    for v in range(n):
        if ar[u][v] == 1 and not used[v]:
            dfs2(v)

if __name__ == "__main__":
    # Read graph:
    # a - adj. matrix
    # n - number of vertices

    # First phase
    used = [False] * n
    order = []
    for i in range(n):
        if not used[i]:
            dfs1(i)
    order = order[::-1]

    # Second phase
    used = [False] * n
    component = []
    for u in order:
```

```

if not used[u]:
    dfs2(u)
    # print to component
    component = []

```

1.6 Проверка на двудольность

Двудольный граф – это граф, вершины которого можно разбить на два множества таким образом, что все ребра графа соединяют вершины только из разных множеств, т.е. две вершины, соединенные ребром, не могут находиться в одном множестве. научимся решать задачу: является ли граф двудольным?

Перед тем, как приступить к решению, попробуем немного переформулировать условие. Необходимо все вершины графа раскрасить в два цвета таким образом, чтобы никакие две вершины одного цвета не были соединены ребром. Очевидно, что раскраска в два цвета эквивалентна разбиению на два множества. Данная задача не имеет решения, только если в графе присутствует цикл нечетной длины. Попробуйте сами доказать, что любой цикл нечетной длины не может быть раскрашен таким образом.

Само решение задачи будет представлять из себя серию обходов в глубину, где каждый обход будет красить свою компоненту связности. Изначально все вершины бесцветные (-1). Запустим обход из любой вершины, покрасив ее в белый цвет (0). Из нее мы перейдем в следующую вершину, покрасим ее в черный цвет (1) и т.д. Если мы в какой-то момент найдем ребро, которое ведет в уже покрашенную вершину, то нам надо проверить цвет этой вершины. Если он противоположен цвету текущей вершины, то все хорошо (нашли цикл четной длины). Иначе, мы нашли цикл нечетной длины и задача не имеет решения.

```

def dfs(u, c):
    color[u] = c
    for v in range(n):
        if a[u][v] == 1:
            if color[v] == -1:
                dfs(v, c ^ 1)
            elif color[v] == c:
                print("NO")
                exit(0)

if __name__ == "__main__":
    n = int(input())
    a = []
    for i in range(n):
        a.append(list(map(int, input().split())))

```



```
color = [-1] * n
for i in range(n):
    if color[i] == -1:
        dfs(i, 0)
print("YES")
```

2 Обход в ширину