

## Лекция 16. Компоненты связности. Задача 2-выполнимости (2-SAT).

### 1. Введение

В этой лекции мы углубимся в тему **связности в ориентированных графах** и рассмотрим её применение к решению задачи **2-выполнимости (2-SAT)**.

**Компоненты сильной связности (КСС)** – это множества вершин в ориентированном графе, где каждая вершина достижима из любой другой.

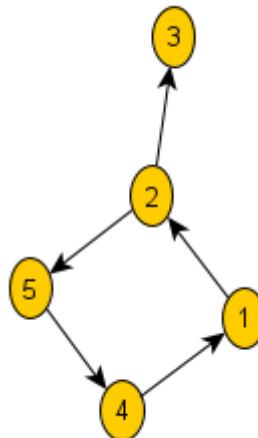
**2-SAT** – это частный случай задачи выполнимости булевых формул, где каждая скобка содержит ровно два литерала. Мы научимся решать её за линейное время, используя свойства КСС.

### 2. Компоненты сильной связности (КСС)

#### 2.1. Определение

В **ориентированном графе** компонента сильной связности – это максимальное множество вершин, в котором для любых двух вершин  $u$  и  $v$  существует путь из  $u$  в  $v$  и из  $v$  в  $u$ .

**Пример:**



- КСС 1: {1, 2, 4, 5} (все достижимы друг из друга)
- КСС 2: {3} (достижима только сама из себя)

#### 2.2. Алгоритм Косарайю

Алгоритм состоит из двух обходов в глубину (DFS):

1. **Первый DFS** на исходном графе для определения порядка выхода вершин.

2. **Второй DFS** на транспонированном графе (ребра развернуты) в порядке убывания времени выхода.

**Транспонированный граф** – граф, где все ребра развернуты в обратную сторону.

## 2.3. Реализация алгоритма Косарайю

```
from collections import defaultdict
from typing import List

class Kosaraju:
    def __init__(self, n: int):
        # Инициализация класса
        self.n = n # Количество вершин в графе
        self.graph = defaultdict(list) # Исходный ориентированный граф
        self.transposed = defaultdict(list) # Транспонированный граф (ребра
        в обратную сторону)
        self.visited = [False] * n # Массив посещённых вершин
        self.order = [] # Порядок выхода вершин из DFS
        self.components = [] # Список найденных компонент связности

    def add_edge(self, u: int, v: int):
        """Добавляет ориентированное ребро  $u \rightarrow v$  в оба графа"""
        self.graph[u].append(v) # Прямое ребро в исходном графе
        self.transposed[v].append(u) # Обратное ребро в транспонированном
        графе

    def dfs1(self, u: int):
        """Первый DFS: обход исходного графа для определения порядка
        выхода"""
        self.visited[u] = True # Помечаем вершину как посещённую
        # Рекурсивно обходим всех соседей
        for v in self.graph[u]:
            if not self.visited[v]:
                self.dfs1(v)
        # После обработки всех потомков добавляем вершину в порядок выхода
        self.order.append(u)

    def dfs2(self, u: int, component: List[int]):
        """Второй DFS: обход транспонированного графа для поиска
        компоненты"""
        self.visited[u] = True # Помечаем вершину как посещённую
        component.append(u) # Добавляем вершину в текущую компоненту
        # Обходим соседей в транспонированном графе (обратные ребра)
        for v in self.transposed[u]:
            if not self.visited[v]:
                self.dfs2(v, component)

    def find_scc(self) -> List[List[int]]:
        """Основной метод: находит все компоненты сильной связности"""
        # Шаг 1: Первый DFS на исходном графе
        self.visited = [False] * self.n # Сбрасываем массив посещений
        for i in range(self.n):
            if not self.visited[i]:
                self.dfs1(i) # Запускаем DFS из непосещённых вершин
```

```

# Шаг 2: Второй DFS на транспонированном графе
self.visited = [False] * self.n # Снова сбрасываем посещения
# Обходим вершины в порядке УБЫВАНИЯ времени выхода (обратный
порядок)
for u in reversed(self.order):
    if not self.visited[u]:
        component = [] # Создаём новую компоненту
        self.dfs2(u, component) # Находим все вершины компоненты
        self.components.append(component) # Добавляем компоненту в
результат

return self.components

# Пример использования
g = Kosaraju(6)
edges = [(0, 1), (1, 2), (2, 0), (1, 3), (3, 4), (4, 5), (5, 3)]
for u, v in edges:
    g.add_edge(u, v)

print("Компоненты сильной связности:", g.find_scc())
# Вывод: [[0, 2, 1], [3, 5, 4]]

```

### 3. Конденсация графа

#### 3.1. Определение

**Конденсация** – это граф, полученный сжатием каждой КСС в одну вершину. Рёбра между КСС сохраняются, если есть хотя бы одно ребро между вершинами исходных КСС.

**Свойства:**

- Конденсация – **ациклический ориентированный граф** (DAG).
- Любой путь в конденсации соответствует пути в исходном графе.

#### 3.2. Построение конденсации

Алгоритм:

1. Найти все КСС (алгоритмом Косарайю).
2. Для каждой КСС создать новую вершину.
3. Добавить рёбра между вершинами конденсации, если в исходном графе есть рёбра между соответствующими КСС.

```

from collections import defaultdict
from typing import List, Tuple, Optional

class Kosaraju:...

```

```

def build_condensation(n: int, edges: List[Tuple[int, int]]) -> Tuple[int, List[List[int]]):

    """Строит конденсацию графа - сжатие КСС в отдельные вершины"""

    # Шаг 1: Находим все компоненты сильной связности
    kosaraju = Kosaraju(n)
    for u, v in edges:
        kosaraju.add_edge(u, v)
    scc = kosaraju.find_scc() # Получаем список компонент

    # Шаг 2: Создаём массив, сопоставляющий каждой вершине её компоненту
    comp_id = [-1] * n # comp_id[i] = номер компоненты вершины i
    for idx, component in enumerate(scc):
        for node in component:
            comp_id[node] = idx # Присваиваем всем вершинам компоненты один
ID

    # Шаг 3: Строим рёбра между компонентами
    condensation = defaultdict(set) # Используем set чтобы избежать
дубликатов
    for u, v in edges:
        # Если ребро соединяет разные компоненты
        if comp_id[u] != comp_id[v]:
            # Добавляем ребро между компонентами
            condensation[comp_id[u]].add(comp_id[v])

    # Преобразуем в списки для удобства
    cond_graph = [[] for _ in range(len(scc))]
    for u in condensation:
        cond_graph[u] = list(condensation[u])

    return len(scc), cond_graph # Возвращаем количество компонент и граф
конденсации

# Пример
n = 6
edges = [(0, 1), (1, 2), (2, 0), (1, 3), (3, 4), (4, 5), (5, 3)]
comp_count, cond_graph = build_condensation(n, edges)
print("Конденсация (вершины, рёбра):", comp_count,
dict(enumerate(cond_graph)))
# Вывод: Конденсация (вершины, рёбра): 2 {0: [1], 1: []}

```

## 4. Задача 2-выполнимости (2-SAT)

### 4.1. Определение

Дана булева формула в **конъюнктивной нормальной форме (КНФ)**, где каждая скобка содержит ровно два литерала.

**Пример:**

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (x_2 \vee \neg x_3)$$

Требуется определить, существует ли такой набор значений переменных, при котором формула выполняется.

## 4.2. Граф импликаций

Каждую скобку  $(a \vee b)$  можно представить как две импликации:

$$\neg a \Rightarrow b, \neg b \Rightarrow a$$

Строим ориентированный граф с  $2n$  вершинами (для каждой переменной и её отрицания).

**Пример для скобки  $(x_1 \vee \neg x_2)$ :**

- $\neg x_1 \Rightarrow \neg x_2$
- $x_2 \Rightarrow x_1$

## 4.3. Алгоритм решения 2-SAT

1. Построить граф импликаций из всех скобок.
2. Найти КСС в графе импликаций.
  - Если для какой-либо переменной  $x_i$  и  $\neg x_i$  находятся в одной КСС  $\rightarrow$  формула невыполнима.
  - Иначе можно построить решение:
3. В конденсации графа импликаций вершины уже топологически отсортированы.
4. Присваиваем значения переменным в порядке убывания топологической сортировки:
  - Если вершина  $x_i$  встречается раньше  $\neg x_i$ , присваиваем  $x_i = False$ .
  - Иначе  $x_i = True$ .

## 4.4. Реализация 2-SAT

```
from collections import defaultdict
from typing import List, Optional

class TwoSAT:
    def __init__(self, n: int):
        # n - количество переменных (x1, x2, ..., xn)
        self.n = n

        # Граф импликаций: каждая вершина - литерал (xi или -xi)
        self.graph = defaultdict(list)

        # Транспонированный граф для алгоритма Косарайю
        self.transposed = defaultdict(list)

        # Массив посещений для DFS (2n вершин: n переменных и n их отрицаний)
        self.visited = [False] * (2 * n)
```

```

# Порядок выхода вершин из первого DFS
self.order = []

# ID компоненты сильной связности для каждой вершины
self.comp_id = [-1] * (2 * n)

def add_clause(self, a: int, b: int):
    """Добавляет дизъюнкцию (a ∨ b) в виде импликаций в граф"""

    def get_index(var):
        """
        Преобразует литерал в индекс вершины графа.
        Схема нумерации:
        - Положительный литерал  $x_i \rightarrow$  индекс  $2*(i-1)$ 
        - Отрицательный литерал  $\neg x_i \rightarrow$  индекс  $2*(i-1) + 1$ 

        Пример для  $n=3$ :
         $x_1 \rightarrow 0, \neg x_1 \rightarrow 1$ 
         $x_2 \rightarrow 2, \neg x_2 \rightarrow 3$ 
         $x_3 \rightarrow 4, \neg x_3 \rightarrow 5$ 
        """
        if var > 0:
            # Положительный литерал ( $x_i$ )
            return 2 * (var - 1)
        else:
            # Отрицательный литерал ( $\neg x_i$ )
            return 2 * (-var - 1) + 1

    # Преобразуем дизъюнкцию (a ∨ b) в две импликации:
    # ( $\neg a \Rightarrow b$ ) и ( $\neg b \Rightarrow a$ )

    # Получаем индексы для литералов
    not_a = get_index(-a) # Вершина для  $\neg a$ 
    not_b = get_index(-b) # Вершина для  $\neg b$ 
    b_idx = get_index(b) # Вершина для  $b$ 
    a_idx = get_index(a) # Вершина для  $a$ 

    # Добавляем импликации в граф:
    # Если  $\neg a$  истинно, то  $b$  должно быть истинно
    self.graph[not_a].append(b_idx)
    # Если  $\neg b$  истинно, то  $a$  должно быть истинно
    self.graph[not_b].append(a_idx)

def dfs1(self, u: int):
    """Первый DFS: обход в глубину для построения порядка выхода"""
    self.visited[u] = True

    # Рекурсивно обходим всех соседей
    for v in self.graph[u]:
        if not self.visited[v]:
            self.dfs1(v)

    # После обработки всех потомков добавляем вершину в порядок выхода
    # (вершины добавляются в порядке завершения их обработки)
    self.order.append(u)

```

```

    def dfs2(self, u: int, cid: int):
        """Второй DFS: обход транспонированного графа для поиска компонент
связности"""
        # Присваиваем вершине ID текущей компоненты
        self.comp_id[u] = cid

        # Обходим соседей в транспонированном графе
        for v in self.transposed[u]:
            # Если вершина ещё не посещена, рекурсивно обрабатываем её
            if self.comp_id[v] == -1:
                self.dfs2(v, cid)

    def solve(self) -> Optional[List[bool]]:
        """Решает задачу 2-SAT, возвращает назначение переменных или None
если невыполнимо"""

        # ШАГ 1: Строим транспонированный граф
        # Для каждого ребра  $u \rightarrow v$  в исходном графе добавляем ребро  $v \rightarrow u$  в
транспонированный
        for u in self.graph:
            for v in self.graph[u]:
                self.transposed[v].append(u)

        # ШАГ 2: Первый DFS - определяем порядок выхода вершин
        # Сбрасываем массив посещений
        self.visited = [False] * (2 * self.n)

        # Запускаем DFS из всех непосещённых вершин
        for i in range(2 * self.n):
            if not self.visited[i]:
                self.dfs1(i)

        # ШАГ 3: Второй DFS - находим компоненты сильной связности
        # Обходим вершины в ОБРАТНОМ порядке выхода (из первого DFS)
        cid = 0 # Счётчик ID компонент
        for u in reversed(self.order):
            # Если вершина ещё не назначена компоненте, начинаем новую
компоненту
            if self.comp_id[u] == -1:
                self.dfs2(u, cid)
                cid += 1

        # ШАГ 4: Проверяем на противоречия
        # Для каждой переменной  $x_i$  проверяем, не лежат ли  $x_i$  и  $\neg x_i$  в одной
компоненте
        for i in range(self.n):
            #  $x_i$  имеет индекс  $2*i$ ,  $\neg x_i$  имеет индекс  $2*i + 1$ 
            if self.comp_id[2 * i] == self.comp_id[2 * i + 1]:
                # Если  $x_i$  и  $\neg x_i$  в одной компоненте - противоречие, формула
невыполнима
                return None

        # ШАГ 5: Строим решение
        assignment = [False] * self.n # Начальное назначение (все False)

```

```

    for i in range(self.n):
        # Логика выбора значения:
        # Если компонента  $x_i$  ( $2*i$ ) имеет БОЛЬШИЙ номер, чем компонента
         $\neg x_i$  ( $2*i + 1$ ),
            # то это означает, что в топологической сортировке конденсации
            # компонента  $x_i$  находится ПОСЛЕ компоненты  $\neg x_i$ 
            # В этом случае присваиваем  $x_i = \text{True}$ 
        if self.comp_id[2 * i] > self.comp_id[2 * i + 1]:
            assignment[i] = True
        else:
            assignment[i] = False

    return assignment

# Пример использования
ts = TwoSAT(3)
# Добавляем клаузы для формулы:  $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (x_2 \vee \neg x_3)$ 
ts.add_clause(1, -2) #  $(x_1 \vee \neg x_2)$ 
ts.add_clause(-1, 3) #  $(\neg x_1 \vee x_3)$ 
ts.add_clause(2, -3) #  $(x_2 \vee \neg x_3)$ 

solution = ts.solve()
print("Решение 2-SAT:", solution) # Ожидаем [True, True, True] или другое
                                    # допустимое решение

```

## 5. Сравнение методов

Метод	Время	Память	Применение
Косарайю (КСС)	$O(n + m)$	$O(n + m)$	Конденсация, анализ зависимостей
2-SAT (граф + КСС)	$O(n + m)$	$O(n + m)$	Проверка выполнимости формул