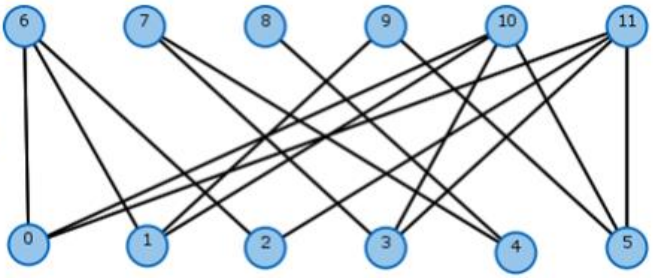
Лабораторная работа 6
Шибко Татьяна
Вариант 3

Условие

Решите две задачи. 1. Найдите максимальное паросочетание и минимальное вершинное покрытие в двудольном графе. 2. Решите задачу о назначениях:

Вариант 3.



5	6	1	6	5
6	1	1	4	2
3	2	5	4	3
5	6	6	6	4
1	3	2	2	2

Решение

Для начала выделим основные определения.

Сочетанием (matching) простого графа называется его подмножество рёбер, никакие два из которых не имеют общей вершины.

***Задача о максимальном паросочетании (matching problem*)** заключается в нахождении по данному графу сочетания максимального размера.

Назначение можно задать биекцией между двумя конечными множествами из n элементов, которая задается перестановкой $(p_1, p_2, \ldots, p_n)$.

Обоснование выбора методов

1. Поиск максимального паросочетания

Для решения задачи о максимальном паросочетании был выбран метод, основанный на преобразовании графа в потоковую сеть и применении алгоритма Форда-Фалкерсона. Этот подход позволяет эффективно найти наибольшее множество рёбер, которые не имеют общих вершин, и при этом обеспечивают максимальное покрытие графа.

- Почему этот метод? Алгоритм Форда-Фалкерсона прекрасно подходит для решения задач на двудольных графах. Основная идея состоит в преобразовании графа в потоковую сеть с фиктивными вершинами — источником и стоком. Рёбра графа получают пропускную способность, равную 11, а затем осуществляется поиск так называемых увеличивающих путей с использованием обхода в глубину (DFS). Найденные пути обновляют поток в сети, что приближает нас к конечному решению.
- Преимущества метода:
  - Этот метод гарантированно находит максимальное паросочетание в двудольном графе.
  - Применение обхода в глубину делает алгоритм интуитивно понятным и достаточно простым для реализации.

- Структура метода позволяет легко визуализировать решение, что важно при анализе работы алгоритма.

2. Задача о назначениях (Венгерский алгоритм)
Для решения задачи о назначениях был выбран Венгерский алгоритм, который является оптимальным для минимизации затрат при распределении задач между исполнителями. Этот метод работает на основе теории минимального покрытия графа, что делает его идеальным выбором для подобных задач.

- Почему этот метод? Венгерский алгоритм эффективно решает задачу о назначениях с использованием потенциалов строк и столбцов. Основная идея заключается в итеративном обновлении этих потенциалов таким образом, чтобы минимизировать общую стоимость выполнения задач. Алгоритм имеет временную сложность O(n3)O(n^3), что позволяет использовать его для матриц средних размеров. Более того, он подходит как для квадратных, так и для прямоугольных матриц.
- Преимущества метода:
  - Венгерский алгоритм гарантированно находит оптимальное решение, минимизируя затраты.
  - Метод универсален и может быть адаптирован к различным формам входных данных.
  - Простота его логики делает его понятным и удобным для реализации.

Пояснение шагов решения
1. Поиск максимального паросочетания
Алгоритм реализован в несколько шагов:

1. Граф разделяется на две доли LL и RR с помощью раскраски вершин в два цвета, что позволяет проверить его двудольность.

```
def split_graph(graph):
    if len(graph) == 0:
        raise ValueError('graph should be non-empty dict')
    colors = {key: None for key in graph.keys()}

    def set_color(node):
        cur_color = colors[node]
        neighbor_color = 'r' if cur_color == 'l' else 'l'
        for g in graph[node]:
            if colors[g] is not None:
                if colors[g] != neighbor_color:
                    raise ValueError('Graph is not bipartite')
            else:
                colors[g] = neighbor_color
                set_color(g)

    for node in graph.keys():
```

```
        if colors[node] is None:
            colors[node] = 'l'
            set_color(node)
    res = {'l': [], 'r': []}
    for key, value in colors.items():
        if value == 'l':
            res['l'].append(key)
        else:
            res['r'].append(key)
    return res
```

2. Строится потоковая сеть: добавляются фиктивные вершины — источник (ss) и сток (tt). Вершины из LL соединяются с источником, а вершины из RR — со стоком.

```
def build_net(graph, colors):
    net = {key: [] for key in graph.keys()}
    net['s'] = colors['l']
    net['t'] = []
    for u in colors['r']:
        net[u].append('t')
    for u in colors['l']:
        for v in graph[u]:
            net[u].append(v)
    return net
```

3. Используется обход в глубину (DFS) для поиска увеличивающих путей между источником и стоком.

```
def dfs(graph, start_node, visited=None, from_=None):
    if visited is None:
        visited = set()
    if from_ is None:
        from_ = {key: None for key in graph.keys()}
        from_[start_node] = start_node
    visited.add(start_node)
    for neighbor in graph[start_node]:
        if neighbor not in visited:
            from_[neighbor] = start_node
            dfs(graph, neighbor, visited, from_)
    return visited, from_
```

Полученный путь извлекается с помощью:

```
def find_dfs_path(graph, start_node, end_node):
    _, from_ = dfs(graph, start_node)
    node = end_node
    path = []
    while True:
        if from_[node] is None:
            return None
        if from_[node] != node:
            path.append(node)
            node = from_[node]
        else:
```

```
            break
    path.append(start_node)
    return list(reversed(path))
```

4. Если увеличивающий путь найден, поток в сети обновляется, а рёбра, входящие в путь, добавляются в решение. Процесс повторяется, пока существуют увеличивающие пути.

```
def find_max_matching(graph):
    colors = split_graph(graph)
    net = build_net(graph, colors)
    matching = []
    while True:
        path = find_dfs_path(net, 's', 't')
        if path is None:
            break
        net['s'].remove(path[1])
        net[path[-2]].remove('t')
        for i in range(1, len(path) - 2):
            net[path[i]].remove(path[i + 1])
            net[path[i + 1]].append(path[i])
            edge = tuple(sorted([path[i], path[i + 1]]))
            if edge in matching:
                matching.remove(edge)
            else:
                matching.append(edge)
    return matching
```

Результат: Набор рёбер, представляющих максимальное паросочетание в графе.

```
Максимальное паросочетание: [(2, 7), (0, 11), (3, 9), (4, 10), (5, 6), (1, 8)]
```

2. Задача о назначениях

Решение задачи состоит из следующих шагов:

1. Матрица затрат дополняется до квадратной, если число задач и исполнителей не совпадает, путём добавления фиктивных строк или столбцов с нулевыми значениями.

```
a = np.vstack([np.zeros((1, m), dtype=int), a])
a = np.hstack([np.zeros((n+1, 1), dtype=int), a])
```

2. Инициализируются потенциалы строк (uu) и столбцов (vv), которые помогают отслеживать минимальное покрытие матрицы.

```
u = np.zeros(n + 1, dtype=int)
v = np.zeros(m + 1, dtype=int)
p = np.zeros(m + 1, dtype=int)
way = np.zeros(m + 1, dtype=int)
```

3. Итеративно обновляются потенциалы:

```
for i in range(1, n + 1):
        p[0] = i
        j0 = 0
        minv = np.zeros(m + 1, dtype=int) + np.inf
        used = np.zeros(m + 1, dtype=bool)
        while True:
```

```
            used[j0] = True
            i0 = p[j0]
            delta = np.inf
            j1 = None
            for j in range(1, m+1):
                if not used[j]:
                    cur = a[i0][j] - u[i0] - v[j]
                    if cur < minv[j]:
                        minv[j] = cur
                        way[j] = j0
                    if minv[j] < delta:
                        delta = minv[j]
                        j1 = j
            for j in range(m + 1):
                if used[j]:
                    u[p[j]] += delta
                    v[j] -= delta
                else:
                    minv[j] -= delta
            j0 = j1
            if p[j0] == 0:
                break
```

4. Построение пути и обновление назначения

После нахождения минимального покрытия назначаются задачи:

```
    while True:
            j1 = way[j0]
            p[j0] = p[j1]
            j0 = j1
            if not j0:
                break
    cost = -v[0]
    ans = np.zeros(n + 1)
    for j in range(1, m+1):
        ans[p[j]] = j
    return cost, ans[1:]
```

Результат: Минимальная общая стоимость распределения задач и соответствие "исполнитель-задача".

```
Стоимость: 11
Назначения: [3. 2. 1. 5. 4.]
```

Листинг кода

```
import numpy as np

def find_max_matching(graph):
    colors = split_graph(graph)
    net = build_net(graph, colors)
    matching = []
    while True:
        path = find_dfs_path(net, 's', 't')
        if path is None:
            break
        net['s'].remove(path[1])
        net[path[-2]].remove('t')
        for i in range(1, len(path) - 2):
            net[path[i]].remove(path[i + 1])
            net[path[i + 1]].append(path[i])
            edge = tuple(sorted([path[i], path[i + 1]]))
            if edge in matching:
                matching.remove(edge)
            else:
                matching.append(edge)
    return matching

def dfs(graph, start_node, visited=None, from_=None):
    if visited is None:
        visited = set()
    if from_ is None:
        from_ = {key: None for key in graph.keys()}
        from_[start_node] = start_node
    visited.add(start_node)
    for neighbor in graph[start_node]:
        if neighbor not in visited:
            from_[neighbor] = start_node
            dfs(graph, neighbor, visited, from_)
    return visited, from_

def find_dfs_path(graph, start_node, end_node):
    _, from_ = dfs(graph, start_node)
    node = end_node
    path = []
    while True:
        if from_[node] is None:
            return None
        if from_[node] != node:
            path.append(node)
            node = from_[node]
        else:
            break
    path.append(start_node)
    return list(reversed(path))
```

```
def split_graph(graph):
    if len(graph) == 0:
        raise ValueError('graph should be non-empty dict')
    colors = {key: None for key in graph.keys()}

    def set_color(node):
        cur_color = colors[node]
        neighbor_color = 'r' if cur_color == 'l' else 'l'
        for g in graph[node]:
            if colors[g] is not None:
                if colors[g] != neighbor_color:
                    raise ValueError('Graph is not bipartite')
            else:
                colors[g] = neighbor_color
                set_color(g)

    for node in graph.keys():
        if colors[node] is None:
            colors[node] = 'l'
            set_color(node)
    res = {'l': [], 'r': []}
    for key, value in colors.items():
        if value == 'l':
            res['l'].append(key)
        else:
            res['r'].append(key)
    return res

def build_net(graph, colors):
    net = {key: [] for key in graph.keys()}
    net['s'] = colors['l']
    net['t'] = []
    for u in colors['r']:
        net[u].append('t')
    for u in colors['l']:
        for v in graph[u]:
            net[u].append(v)
    return net

def hungurian_assignment(a: np.ndarray):
    n, m = a.shape
    a = np.vstack([np.zeros((1, m), dtype=int), a])
    a = np.hstack([np.zeros((n+1, 1), dtype=int), a])
    u = np.zeros(n + 1, dtype=int)
    v = np.zeros(m + 1, dtype=int)
    p = np.zeros(m + 1, dtype=int)
    way = np.zeros(m + 1, dtype=int)
    for i in range(1, n + 1):
        p[0] = i
        j0 = 0
```

```
        minv = np.zeros(m + 1, dtype=int) + np.inf
        used = np.zeros(m + 1, dtype=bool)
        while True:
            used[j0] = True
            i0 = p[j0]
            delta = np.inf
            j1 = None
            for j in range(1, m+1):
                if not used[j]:
                    cur = a[i0][j] - u[i0] - v[j]
                    if cur < minv[j]:
                        minv[j] = cur
                        way[j] = j0
                    if minv[j] < delta:
                        delta = minv[j]
                        j1 = j
            for j in range(m + 1):
                if used[j]:
                    u[p[j]] += delta
                    v[j] -= delta
                else:
                    minv[j] -= delta
            j0 = j1
            if p[j0] == 0:
                break
        while True:
            j1 = way[j0]
            p[j0] = p[j1]
            j0 = j1
            if not j0:
                break
    cost = -v[0]
    ans = np.zeros(n + 1)
    for j in range(1, m+1):
        ans[p[j]] = j
    return cost, ans[1:]

# Граф для задачи о паросочетаниях
graph = {
    0: [6, 7, 11],
    1: [6, 8, 10],
    2: [7, 8, 9],
    3: [9, 10, 11],
    4: [10, 11],
    5: [6, 9],
    6: [0, 1, 5],
    7: [0, 2],
    8: [1, 2],
    9: [2, 3, 5],
    10: [1, 3, 4],
    11: [0, 3, 4]
```

```
}

max_matching = find_max_matching(graph)
print(f'Максимальное паросочетание: {max_matching}')

# Матрица затрат для задачи о назначениях
a = np.array([
    [5, 6, 1, 6, 5],
    [6, 1, 4, 4, 2],
    [3, 2, 5, 4, 3],
    [5, 6, 6, 6, 4],
    [1, 3, 2, 2, 2],
])

cost, ans = hungurian_assignment(a)
print(f'Стоимость: {cost}')
print(f'Назначения: {ans}')
```