

**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**  
**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**  
**ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ**

**Лабораторная работа №6**  
**по дисциплине «Исследование операций»**  
**Вариант 2**

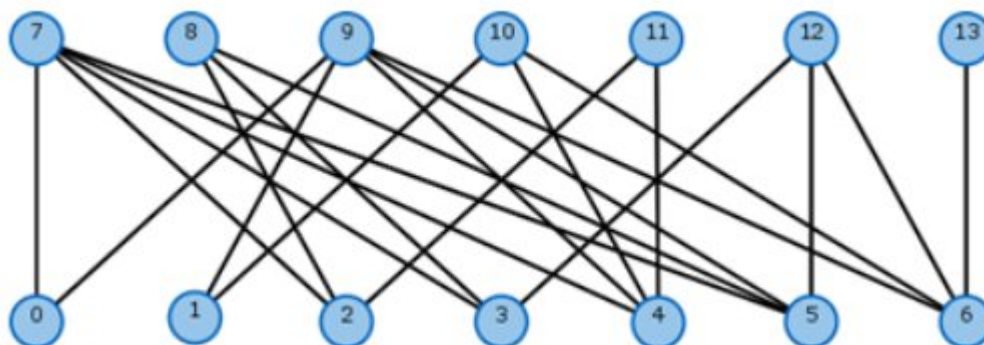
Бобовоза Владислава  
Сергеевича  
студента 3 курса, 6 группы  
специальность «прикладная  
математика»

Минск, 2024

## Постановка задачи.

Решить две задачи:

- 1) Найдите максимальное паросочетание и минимальное вершинное покрытие в двудольном графе



- 2) Решить задачу о назначениях

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

## Решение задачи 1.

Для решения задачи о максимальном паросочетании будем использовать алгоритм Форда-Фалкерсона. Тогда, зная максимальное паросочетание, можем легко получить минимальное вершинное покрытие по алгоритму:

- 1) Построить максимальное паросочетание
- 2) Ориентировать ребра:
  - а) Из паросочетания – из правой доли в левую
  - б) Не из паросочетания – из левой доли в правую
- 3) Запустить обход в глубину из всех свободных вершин левой доли, построить множества  $L^+$ ,  $L^-$ ,  $R^+$ ,  $R^-$ .
- 4) В качестве результата взять  $L^- \cup R^+$ .

## Листинг программы на языке Python:

```
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: # value == 'r' or value is None
            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 find_min_coverage(graph, max_matching):
    colors = split_graph(graph)
    help_graph = build_help_graph(graph, colors, max_matching)
    L = set(colors['l'])
    R = set(colors['r'])
    match_set = set()

    for edge in max_matching:
        match_set.add(edge[0])
        match_set.add(edge[1])
    visited = set()

    for v in (L - match_set):
        vis, _ = dfs(help_graph, v, visited=visited)
        visited = vis | visited

    return list((L - visited) | (R & visited))

def build_help_graph(graph, colors, max_matching):
    new_graph = {key: [] for key in graph.keys()}
    edges = get_all_edges_of_bipartite_graph(graph, colors)

    for edge in edges:
        start = edge[0]
        end = edge[1]
        if edge in max_matching:
            if start in colors['l']:
                new_graph[end].append(start)
            else:
                new_graph[start].append(end)
        else:
            if start in colors['l']:
                new_graph[start].append(end)
            else:
                new_graph[end].append(start)

    return new_graph

def get_all_edges_of_bipartite_graph(graph, colors):
    edges = []

    for u in colors['l']:
        for v in graph[u]:
            edges.append(tuple(sorted([u, v])))

```

```

return edges

if __name__ == '__main__':
    graph = {
        0: [7, 9],
        1: [9, 10],
        2: [7, 8, 11],
        3: [7, 8, 12],
        4: [7, 9, 10, 11],
        5: [7, 8, 9, 12],
        6: [9, 10, 12, 13],
        7: [0, 2, 3, 4, 5],
        8: [2, 3, 5],
        9: [0, 1, 4, 5, 6],
        10: [1, 4, 6],
        11: [2, 4],
        12: [3, 5, 6],
        13: [6]
    }

    max_matching = find_max_matching(graph)
    min_coverage = find_min_coverage(graph, max_matching)

    print(f'Максимальное паросочетание: {max_matching}')
    print(f'Минимальное покрытие: {min_coverage}')

```

### Результат выполнения программы:

```

Максимальное паросочетание: [(0, 9), (1, 10), (5, 7), (4, 11), (2, 8), (3, 12), (6,
13)]
Минимальное покрытие: [0, 1, 2, 3, 4, 5, 6]

```

### Решение задачи 2.

Для решения используем венгерский алгоритм.

### Листинг программы на языке Python:

```

import numpy as np

def hungarian_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: ]

if __name__ == '__main__':
    a = np.array([
        [2, 4, 1, 4, 2],
        [1, 3, 2, 2, 4],
        [3, 2, 5, 2, 3],
        [1, 3, 2, 5, 2],
        [2, 1, 3, 3, 2],
    ])

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

```

### Результат выполнения программы:

```

Стоимость: 7
Назначения: [3. 1. 4. 5. 2.]

```

В назначениях указан номер столбца (для каждой строки с первой), который мы взяли. Тогда:

```

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

```