Лабораторная работа №18
Библиотека NetworkX

1 Цель работы

Изучить библиотеку NetworkX и научиться применять полученные знания на практике.

2 Краткая теория

В данной лабораторной работе рассмотрим библиотеку NetworkX, предназначенную для создания, манипуляции и изучения структуры, динамики и функционирования сложных сетевых структур.

Будем использовать библиотеки NetworkX и Matplotlib:

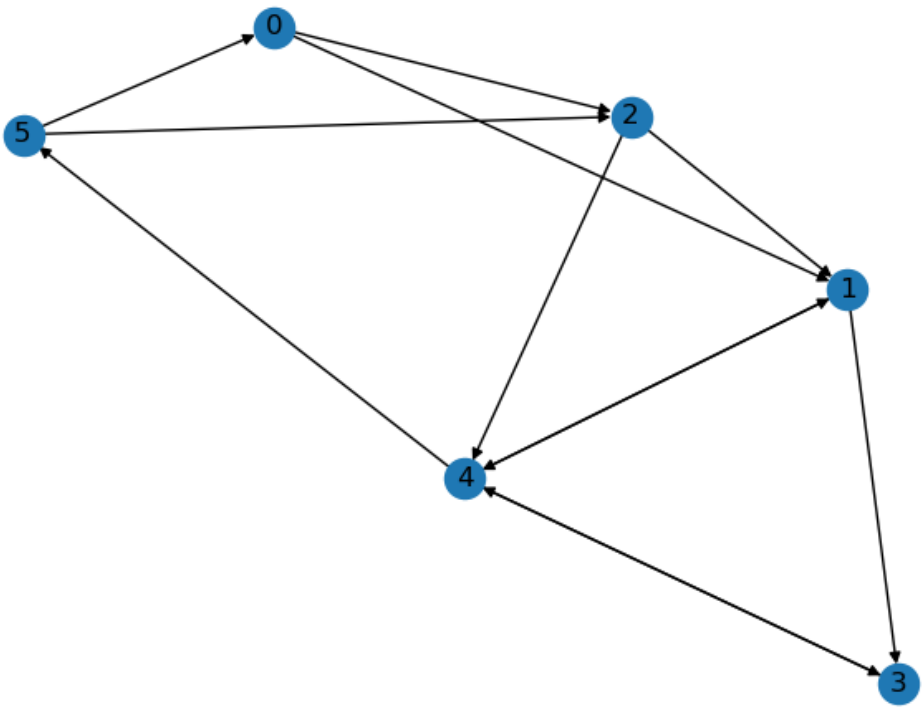
```
import networkx as nx
import matplotlib.pyplot as plt
```

Пусть задан граф множеством смежности:

```
pos = {0: {1, 2},
       1: {3, 4},
       2: {1, 4},
       3: {4},
       4: {1, 3, 5},
       5: {0, 2}}
```

Создадим соответствующий направленный граф:

```
N = len(pos)
G = nx.DiGraph()
a = [(i, j) for i in range(N) for j in pos[i]]  # генерация списка рёбер

G.add_nodes_from(range(N))
G.add_edges_from(a)

nx.draw(G, with_labels=True)

plt.show()
```



2.1 Алгоритмы обхода графа

Во многих приложениях нужно уметь выписывать все вершины графа по одному разу, начиная с некоторой. Это делается с помощью обходов в глубину или в ширину.

Основная идея обходов:

1) на каждом шаге рассмотреть очередную необработанную вершину;

2) пометить эту вершину некоторым образом;

3) до/после обработки данной вершины осуществить обход из всех нерассмотренных соседей.

Для упорядочивания вершин используется очередь (обход в ширину) или стек (обход в глубину).

2.1.1 Поиск в ширину

Код программы, реализующей поиск в ширину (с записью предшественников):

```
def bfs(graph, s, out=0):
    parents = {v: None for v in graph}
    level = {v: None for v in graph}
    level[s] = 0  # уровень начальной вершины
    queue = [s]  # добавляем начальную вершину в очередь
    while queue:  # пока там что-то есть
        v = queue.pop(0)  # извлекаем вершину
        for w in graph[v]:  # запускаем обход из вершины v
            if level[w] is None:  # проверка на посещенность
                queue.append(w)  # добавление вершины в очередь
                parents[w] = v
                level[w] = level[v] + 1  # подсчитываем уровень вершины
        if out:
            print(level[w], level, queue)
    return level, parents
```

И программы, восстанавливающей маршрут:

```
def path(end, parents):
    path = [end]
    parent = parents[end]
    while parent is not None:
        path.append(parent)
        parent = parents[parent]
    return path[::-1]
```

2.1.2 Поиск в глубину

Код программы, реализующей поиск в глубину (с записью предшественников):

```
def dfs(graph, s, out=0):
    level = {v: None for v in graph}
    level[s] = 0  # уровень начальной вершины
    queue = [s]  # добавляем начальную вершину в очередь
    while queue:  # пока там что-то есть
        v = queue.pop(-1)  # извлекаем вершину
        for w in graph[v]:  # запускаем обход из вершины v
            if level[w] is None:  # проверка на посещенность
                queue.append(w)  # добавление вершины в очередь
                level[w] = level[v] + 1  # подсчитываем уровень вершины
        if out:
            print(level[w], level, queue)
    return level
```

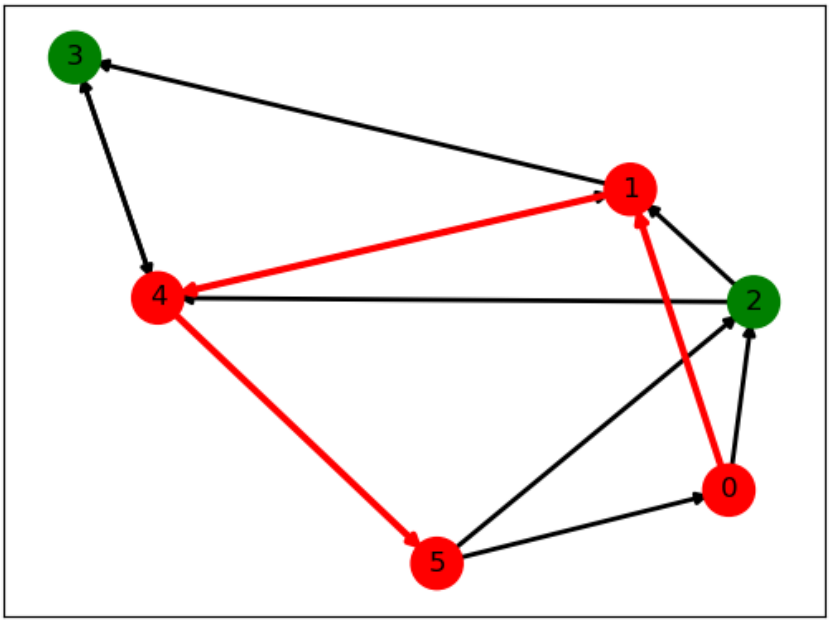
```
print(dfs(pos, 0, 1))
1 {0: 0, 1: 1, 2: 1, 3: None, 4: None, 5: None} [1, 2]
2 {0: 0, 1: 1, 2: 1, 3: None, 4: 2, 5: None} [1, 4]
3 {0: 0, 1: 1, 2: 1, 3: 3, 4: 2, 5: 3} [1, 3, 5]
1 {0: 0, 1: 1, 2: 1, 3: 3, 4: 2, 5: 3} [1, 3]
2 {0: 0, 1: 1, 2: 1, 3: 3, 4: 2, 5: 3} [1]
2 {0: 0, 1: 1, 2: 1, 3: 3, 4: 2, 5: 3} []
{0: 0, 1: 1, 2: 1, 3: 3, 4: 2, 5: 3}
print(pos[2])
{1, 4}
```

Определим с помощью поиска в ширину кратчайший маршрут:

```
level, parents = bfs(pos, 0, out=0)
print(level)
{0: 0, 1: 1, 2: 1, 3: 2, 4: 2, 5: 3}
print(parents)
{0: None, 1: 0, 2: 0, 3: 1, 4: 1, 5: 4}

PATH = path(5, parents)
print(PATH)
[0, 1, 4, 5]
```

Визуализируем этот маршрут:

```
red_node = set(PATH)  # вершины маршрута
red_edges = [(PATH[i], PATH[i + 1]) for i in range(len(PATH) - 1)]  # рёбра
маршрута

# разделение по цветам вершин и рёбер
node_colours = ['g' if node not in red_node else 'red' for node in G.nodes()]
black_edges = [edge for edge in G.edges() if edge not in red_edges]

# построение графа
# p = nx.spring_layout(G)

p = {0: [0.38144628, -0.66882419],
     1: [0.23970166, 0.49135202],
     2: [0.41724407, 0.05678197],
     3: [-0.55966794, 1.],
     4: [-0.44016179, 0.07245783],
     5: [-0.03856228, -0.95176763]}

nx.draw_networkx_nodes(G, p, cmap=plt.get_cmap('jet'),
node_color=node_colours, node_size=500)
nx.draw_networkx_labels(G, p)
nx.draw_networkx_edges(G, p, edgelist=black_edges, width=2.0, edge_color='k',
arrows=True)
nx.draw_networkx_edges(G, p, edgelist=red_edges, width=3.0, edge_color='r',
arrows=True)

plt.show()
```




```
# координаты вершин на рисунке
print(p)
```

```
{0: [0.38144628, -0.66882419], 1: [0.23970166, 0.49135202], 2: [0.41724407,
0.05678197], 3: [-0.55966794, 1.0], 4: [-0.44016179, 0.07245783], 5: [-
0.03856228, -0.95176763]}
```

Пример 1.

Две вершины (v и u) ориентированного графа называют сильно связными, если существует путь из v в u и существует путь из u в v. Ориентированный граф называется сильно связным, если любые две его вершины сильно связны.

Напишите функцию, использующую модифицированый алгоритм поиска в глубину (алгоритм Косарайю) для определения компонент сильной связности.

Алгоритм:

1. Инвертируем дуги исходного ориентированного графа.

2. Запускаем поиск в глубину на этом обращённом графе, запоминая, в каком порядке выходили из вершин.

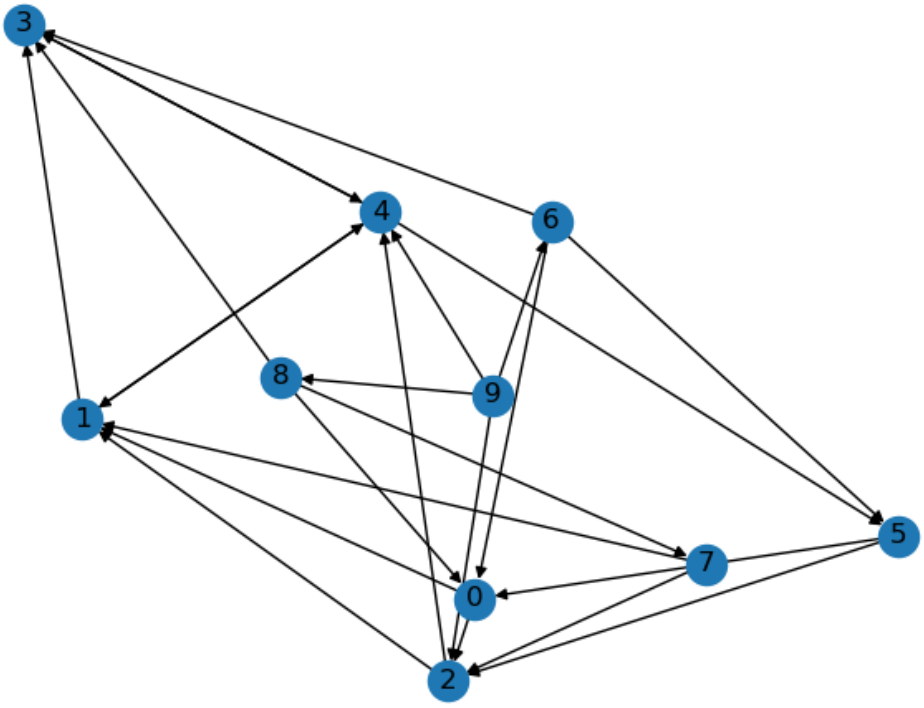
3. Запускаем поиск в глубину на исходном графе, в очередной раз выбирая непосещённую вершину с максимальным номером в векторе, полученном в п. 2.

Полученные из п. 3 деревья и являются сильно связными компонентами.

Находим и строим графически с помощью этой функции компоненты сильной связности графа:

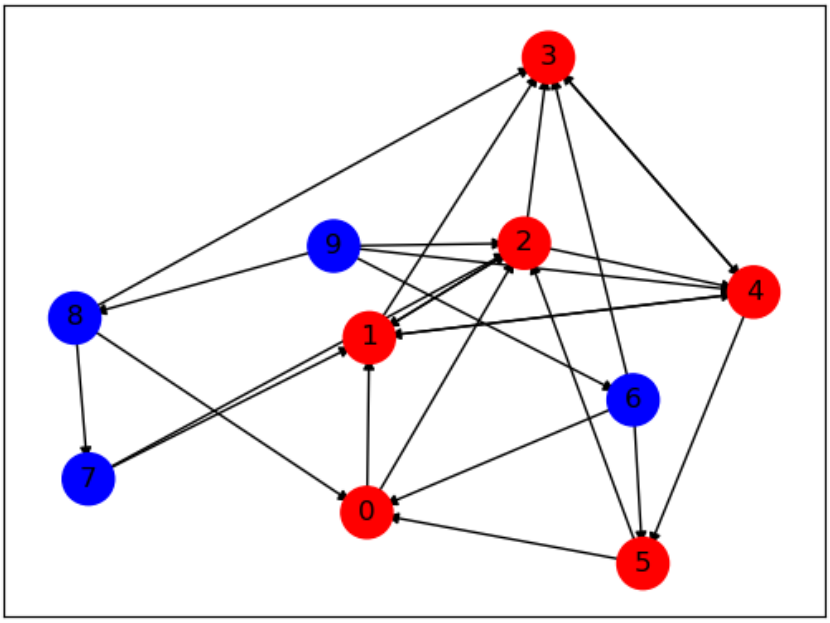
```
pos2 = {0: {1, 2},
        1: {3, 4},
        2: {1, 4},
        3: {4},
        4: {1, 3, 5},
        5: {0, 2},
```

```
        6: {3, 0, 5},
        7: {2, 1},
        8: {0, 7, 3},
        9: {2, 4, 6, 8}}

N = 10
G = nx.DiGraph()
a = [(i, j) for i in range(N) for j in pos2[i]]  # генерация списка рёбер

G.add_nodes_from(range(N))
G.add_edges_from(a)

nx.draw(G, with_labels=True)

plt.show()
```




```
def kosaraju(g):
    size = len(g)

    vis = [False] * size
    l = [0] * size
    x = size
    t = [[]] * size

    def visit(g, vis, u, x, l, t):
        if not vis[u]:
            vis[u] = True
            for v in g[u]:
                vis, x, l, t = visit(g, vis, v, x, l, t)
                t[v] = t[v] + [u]
            x -= 1
            l[x] = u
        return vis, x, l, t

    for u in range(len(g)):
        vis, x, l, t = visit(g, vis, u, x, l, t)
    c = [0] * size
```

```
    def assign(vis, c, t, u, root):
        if vis[u]:
            vis[u] = False
            c[u] = root
            for v in t[u]:
                vis, c, t = assign(vis, c, t, v, root)
        return vis, c, t

    for u in l:
        vis, c, t = assign(vis, c, t, u, u)

    print('Компоненты сильной связности:')
    dup = {i: 0 for i in range(len(c))}
    for i in c:
        dup[i] += 1
    css = []
    for i in dup.keys():
        if dup[i] > 1:
            for j in range(len(c)):
                if c[j] == i:
                    print(j, ', ', sep='', end='')
                    css.append(j)
            print()
    return css


css = kosaraju(pos2)
Компоненты сильной связности:
0, 1, 2, 3, 4, 5,

N = len(pos2)
G = nx.DiGraph()
a = [(i, j) for i in range(N) for j in pos2[i]]

G.add_nodes_from(range(N))
G.add_edges_from(a)

p = nx.spring_layout(G)
red_node = set(css)
red_edges = [(css[i], css[i + 1]) for i in range(len(css) - 1)]

node_colours = ['b' if node not in red_node else 'red' for node in G.nodes()]
black_edges = [edge for edge in G.edges() if edge not in red_edges]

nx.draw_networkx_nodes(G, p, cmap=plt.get_cmap('jet'),
node_color=node_colours, node_size=500)
nx.draw_networkx_labels(G, p)
nx.draw_networkx_edges(G, p, edgelist=black_edges, width=1.0, edge_color='k',
arrows=True)
nx.draw_networkx_edges(G, p, edgelist=red_edges, width=1.0, edge_color='k',
arrows=True)

plt.show()
```



Пример 2.

Создадим две строки, комбинация которых даст нам обозначения всех клеток шахматного поля:

```
letters = 'abcdefgh'
numbers = '12345678'
```

Создадим структуру типа словарь для хранения графа в формате множества смежности:

```
graph = dict()
print(graph)
{}
```

Заполним имена вершин графа:

```
for l in letters:
    for n in numbers:
        graph[l + n] = set()
```

Заполним множества смежности:

```
def add_edge(graph, v1, v2):
    graph[v1].add(v2)
    graph[v2].add(v1)


for i in range(8):
    for j in range(8):
        v1 = letters[i] + numbers[j]
        v2 = ''
        if 0 <= i + 2 < 8 and 0 <= j + 1 < 8:
            v2 = letters[i + 2] + numbers[j + 1]
            add_edge(graph, v1, v2)

        if 0 <= i - 2 < 8 and 0 <= j + 1 < 8:
            v2 = letters[i - 2] + numbers[j + 1]
            add_edge(graph, v1, v2)
```

```
            if 0 <= i + 1 < 8 and 0 <= j + 2 < 8:
                v2 = letters[i + 1] + numbers[j + 2]
                add_edge(graph, v1, v2)

            if 0 <= i - 1 < 8 and 0 <= j + 2 < 8:
                v2 = letters[i - 1] + numbers[j + 2]
                add_edge(graph, v1, v2)
```

Проведём сканирование графа в ширину:

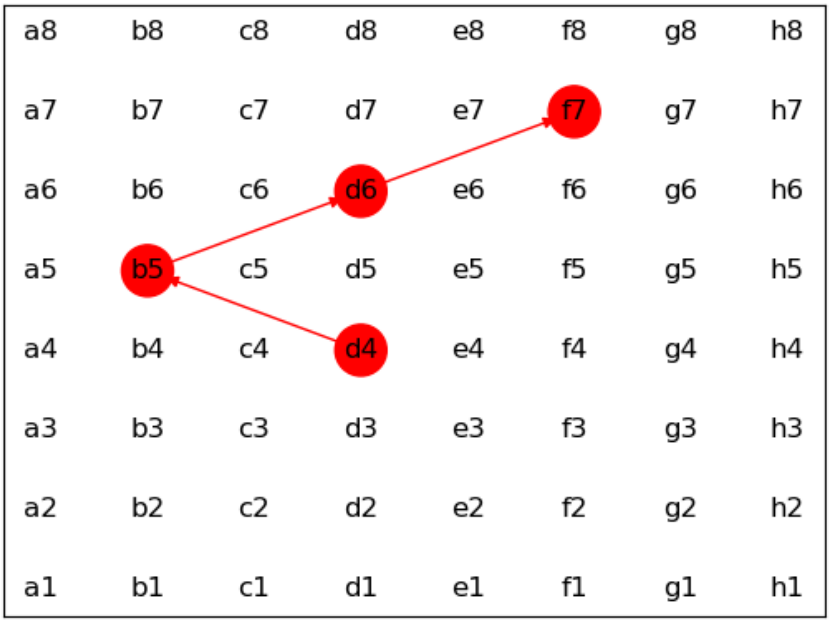
```
start = 'd4'
end = 'f7'

level, parents = bfs(graph, start)
```

И получим маршрут коня:

```
Kpath = path(end, parents)
print(Kpath)
['d4', 'b5', 'd6', 'f7']
```

Нарисуем граф, соответствующий маршрутам коня по шахматной доске, и отметим на нём найденный маршрут.

```
p = {}
for l in range(len(letters)):
    for n in range(len(numbers)):
        p[letters[l] + numbers[n]] = [-1 + l * 0.25, -1 + n * 0.25]

G = nx.DiGraph()
a = [(i, j) for i in graph for j in graph[i]]

G.add_edges_from(a)

# p = nx.spring_layout(G)
red_node = set(Kpath)
red_edges = [(Kpath[i], Kpath[i + 1]) for i in range(len(Kpath) - 1)]

node_colours = ['w' if not node in red_node else 'r' for node in G.nodes()]
black_edges = [edge for edge in G.edges() if edge not in red_edges]

nx.draw_networkx_nodes(G, p, cmap=plt.get_cmap('jet'),
node_color=node_colours, node_size=500)
nx.draw_networkx_labels(G, p)
nx.draw_networkx_edges(G, p, edgelist=red_edges, width=1.0, edge_color='r',
arrows=True)

plt.show()
```



Пример 3.

Рассмотрим граф G(V, E), имеющий V вершин и E ребер. Раскраской графа G называется окрашивание вершин графа G такое, что никакие две смежные вершины не имеют одинаковый цвет. Хроматическое число графа X(G) – это наименьшее число цветов, которое используется для раскраски графа. Известен жадный алгоритм раскраски графа.

Жадный алгоритм последовательного раскрашивания:

Входные данные: граф G(V, E).

Выходные данные: массив c[v] раскрашенных вершин

Для всех вершин определить множество A = {1, 2, 3, …, n} всех цветов.

Выбрать стартовую вершину (с которой начинаем алгоритм). Раскрасить вершину в цвет color. Вычеркнуть этот цвет из множества цветов всех вершин, смежных со стартовой.

Выбрать нераскрашенную вершину v.

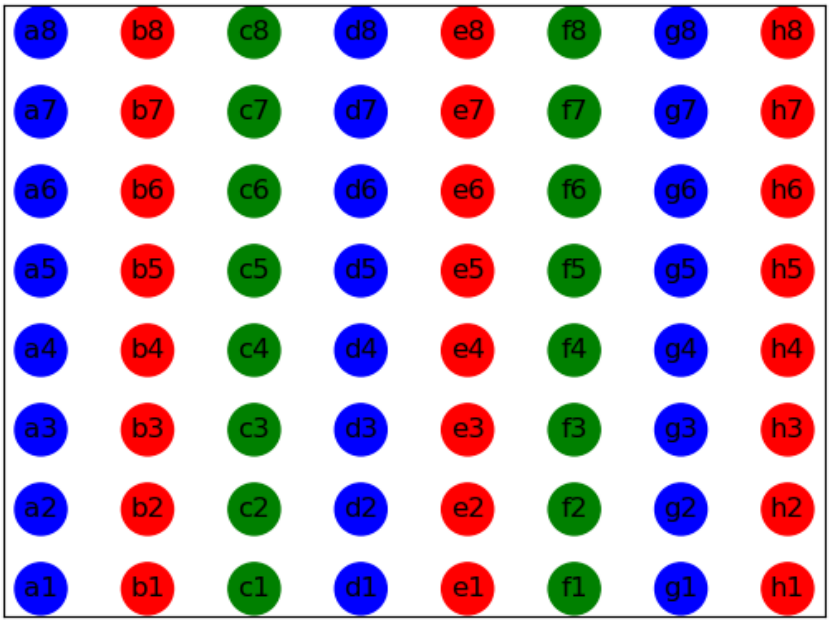
Раскрасить выбранную вершину в минимально возможный цвет из множества A. Вычеркнуть этот цвет из множества цветов всех вершин, смежных с вершиной v.

Проделать шаг 3, шаг 4 для всех нераскрашенных вершин графа.

На основе этого алгоритма раскрасьте граф из задачи про коня.

```
def ggca(graph):
    colors = {i: None for i in graph.keys()}
    for i in graph.keys():
        minimal_set = []
        for s in graph[i]:
            if colors[s] is not None:
                minimal_set.append(colors[s])
```

```
        for c in range(len(graph)):
            if c not in minimal_set:
                colors[i] = c
                break
    return colors


r = ggca(graph)
print(r)
{'a1': 0, 'a2': 0, 'a3': 0, 'a4': 0, 'a5': 0, 'a6': 0, 'a7': 0, 'a8': 0,
'b1': 1, 'b2': 1, 'b3': 1, 'b4': 1, 'b5': 1, 'b6': 1, 'b7': 1, 'b8': 1, 'c1':
2, 'c2': 2, 'c3': 2, 'c4': 2, 'c5': 2, 'c6': 2, 'c7': 2, 'c8': 2, 'd1': 0,
'd2': 0, 'd3': 0, 'd4': 0, 'd5': 0, 'd6': 0, 'd7': 0, 'd8': 0, 'e1': 1, 'e2':
1, 'e3': 1, 'e4': 1, 'e5': 1, 'e6': 1, 'e7': 1, 'e8': 1, 'f1': 2, 'f2': 2,
'f3': 2, 'f4': 2, 'f5': 2, 'f6': 2, 'f7': 2, 'f8': 2, 'g1': 0, 'g2': 0, 'g3':
0, 'g4': 0, 'g5': 0, 'g6': 0, 'g7': 0, 'g8': 0, 'h1': 1, 'h2': 1, 'h3': 1,
'h4': 1, 'h5': 1, 'h6': 1, 'h7': 1, 'h8': 1}

G = nx.DiGraph()
a = [(i, j) for i in graph for j in graph[i]]

G.add_edges_from(a)

# p = nx.spring_layout(G)

node_colours = ['b' if r[node] == 0 else 'r' if r[node] == 1 else 'g' for
node in G.nodes()]

nx.draw_networkx_nodes(G, p, cmap=plt.get_cmap('jet'),
node_color=node_colours, node_size=500)
nx.draw_networkx_labels(G, p)

plt.show()
```



3 Задания для выполнения работы

Задание 1.

Напишите функцию, реализующую алгоритм Дейкстры.

Рис. 4.8. Алгоритм Дейкстры для нахождения кратчайших путей.

```
процедура DIJKSTRA(G, l, s)
{Вход: граф G(V, E) (ориентированный или нет) с неотрицательными
  длинами рёбер {l_e : e ∈ E}; вершина s ∈ V.}
{Выход: для всех вершин u, достижимых из s,
  dist[u] будет равно расстоянию от s до u (и ∞ для недостижимых).}
для всех вершин u ∈ V:
  dist[u] ← ∞
  prev[u] ← nil
dist[s] ← 0
H ← MAKEQUEUE(V) {в качестве ключей используются значения dist}
пока H не пусто:
  u ← DELETEMIN(H)
  для всех рёбер (u, v) ∈ E:
    если dist[v] > dist[u] + l(u, v):
      dist[v] ← dist[u] + l(u, v)
      prev[v] ← u
      DECREASEKEY(H, v, dist[v])
```

Задание 2.

Сгенерируйте случайный взвешенный граф. И определите на нём маршрут минимальной длины с помощью алгоритма Дейкстры.

Задание 3.

Проиллюстрируйте работу одного из алгоритмов (поиска в ширину или глубину, Дейкстры) с помощью визуализации действий с графом на каждой итерации с помощью библиотек NetworkX и Matplotlib, аналогично примеру 1.

Задание 4* (НЕОБЯЗАТЕЛЬНО).

Используйте какой-нибудь интересный алгоритм из библиотеки https://networkx.org/documentation/stable/reference/algorithms/index.html.

Задание 5* (НЕОБЯЗАТЕЛЬНО).

Напишите функцию, реализующую алгоритм Форда-Беллмана https://ru.wikipedia.org/wiki/Алгоритм_Беллмана_—_Форда.

Задание 6* (НЕОБЯЗАТЕЛЬНО).

Найдите выход из лабиринта с помощью различных алгоритмов и сравните их. Вывести рёбра пропорционально их длине.

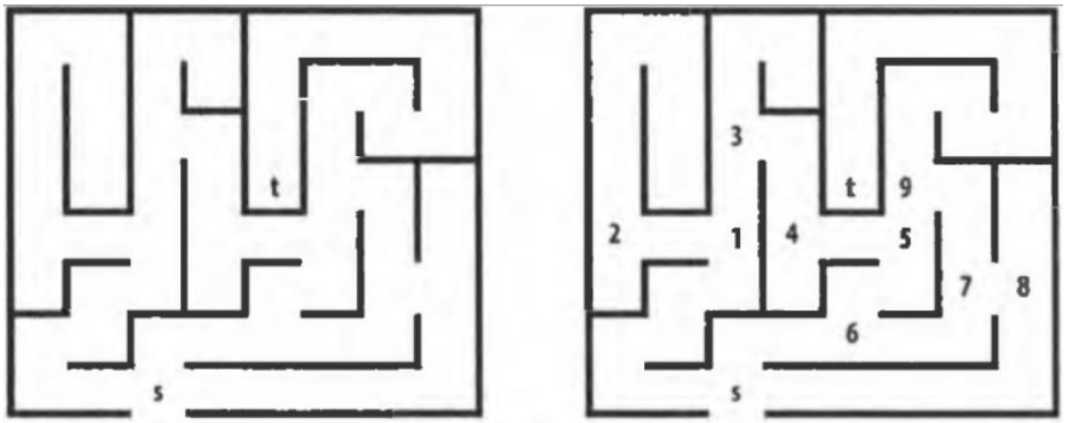


Рис. 6.5. Лабиринт, в котором надо пройти из точки s в точку t

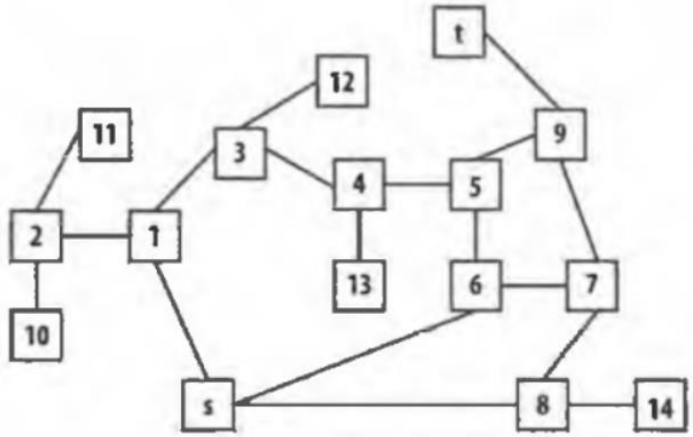


Рис. 6.6. Представление лабиринта на рис. 6.5 в виде графа

Задание 7* (НЕОБЯЗАТЕЛЬНО).

У Еремея есть электросамокат, и он хочет доехать от дома до института, затратив как можно меньше энергии. Весь город расположен на холмистой местности и разделён на квадраты. Для каждого перекрёстка известна его высота в метрах над уровнем моря. Если ехать от перекрёстка с большей высотой до смежного с ним перекрёстка с меньшей высотой, то электроэнергию можно аккумулировать (заряжая скутер), а если наоборот, то расход энергии равен разнице высот между перекрёстками. Помогите Еремею спланировать маршрут, чтобы он затратил наименьшее возможное количество энергии от дома до института и определите это количество.