Задание

Изучение алгоритмов поиска

Цель задания

Исследование алгоритмов решения задач методом поиска

Описание предметной области

Имеется транспортная сеть, связывающая города СНГ. Сеть представлена в виде таблицы связей между городами. Связи являются двусторонними, т.е. допускают движение в обоих направлениях. Необходимо проложить маршрут из одной заданной точки в другую.

Формулировка задания

Этап 1

Неинформированный поиск

На этом этапе известна только топология связей между городами.

Выполнить:

1. поиск в ширину;
2. поиск глубину;
3. поиск с ограничением глубины;
4. поиск с итеративным углублением;
5. двунаправленный поиск.

Отобразить движение по дереву на его графе с указанием сложности каждого вида поиска. Сделать выводы.

Этап 2

Информированный поиск

Воспользовавшись информацией о протяженности связей от текущего узла, выполнить:

1. жадный поиск по первому наилучшему соответствию;
2. затем, использую информацию о расстоянии до цели по прямой от каждого узла, выполнить поиск методом минимизации суммарной оценки A*.

Отобразить на графе выбранный маршрут и сравнить его сложность с неинформированным поиском. Сделать выводы.

Таблица связей между городами

city1	city2	r
Вильнюс	Брест	531
Витебск	Брест	638
Витебск	Вильюс	360
Воронеж	Витебск	869
Воронеж	Волгоград	581
Волгоград	Витебск	1455
Витебск	Ниж.Новгород	911
Вильнюс	Даугавпилс	211
Калининград	Брест	699
Калиниград	Вильнюс	333
Каунас	Вильнюс	102
Киев	Вильнюс	734
Киев	Житомир	131
Житомир	Донецк	863
Житомир	Волгоград	1493
Кишинев	Киев	467
Кишинев	Донецк	812
С.Петербург	Витебск	602
С.Петербург	Калининград	739
С.Петербург	Рига	641
Москва	Казань	815
Москва	Ниж.Новгород	411
Москва	Минск	690
Москва	Донецк	1084
Москва	С.Петербург	664

city1	city2	r
Мурманск	С.Петербург	1412
Мурманск	Минск	2238
Орел	Витебск	522
Орел	Донецк	709
Орел	Москва	368
Одесса	Киев	487
Рига	Каунас	267
Таллинн	Рига	308
Харьков	Киев	471
Харьков	Симферополь	639
Ярославль	Воронеж	739
Ярославль	Минск	940
Уфа	Казань	525
Уфа	Самара	461

> Задание по варианту №9 — SOURCE:"**Брест**" DESTINATION:"**Казань**"

Выполнение работы

1. Построение графа

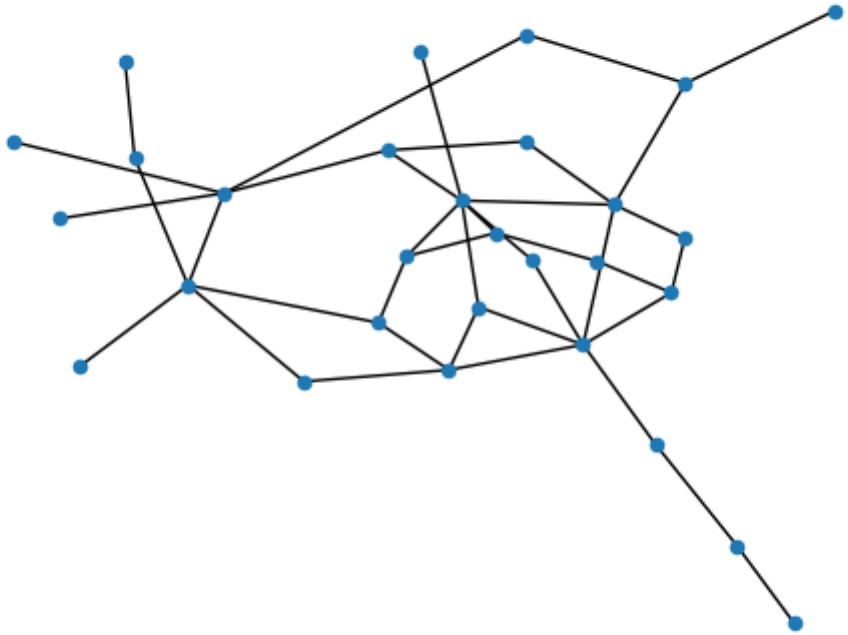
Построим граф с помощью pyplot, networkx и spring_layout:

Определяем граф из файла data.csv

```
In [2]: import matplotlib.pyplot as plt
   ...: import networkx as nx
   ...:
   ...: SOURCE_NODE = "Брест"
   ...: DESTINATION_NODE = "Казань"
   ...:
   ...: G = list()
   ...: with open("data.csv", "r") as dataset:
   ...:     for line in dataset.readlines():
   ...:         e = line.split(";")
   ...:         G.append([
   ...:             e[0],
   ...:             e[1],
   ...:             int(e[2].replace("\n", ""))
   ...:         ])
   ...: print(G)
   ...:
   ...: G_ = nx.Graph()
   ...: for e_ in G:
   ...:     G_.add_edge(e_[0], e_[1], weight=e_[2])
   ...:
   ...: pos = nx.spring_layout(G_, seed=23)
   ...:
   ...: nx.draw_networkx_edges(G_, pos, edgelist=G_.edges)
   ...: nx.draw_networkx_nodes(G_, pos, node_size=20)
   ...: plt.axis("off")
   ...: plt.show()
   ...:
```

Результат:

```
[
    ['Вильнюс', 'Брест', 531],
    ['Витебск', 'Брест', 638],
    ['Витебск', 'Вильюс', 360],
    ['Воронеж', 'Витебск', 869],
    ['Воронеж', 'Волгоград', 581],
    ['Волгоград', 'Витебск', 1455],
    ['Витебск', 'Ниж.Новгород', 911],
    ['Вильнюс', 'Даугавпилс', 211],
    ['Калининград', 'Брест', 699],
    ['Калиниград', 'Вильнюс', 333],
    ['Каунас', 'Вильнюс', 102],
    ['Киев', 'Вильнюс', 734],
    ['Киев', 'Житомир', 131],
    ['Житомир', 'Донецк', 863],
    ['Житомир', 'Волгоград', 1493],
    ['Кишинев', 'Киев', 467],
    ['Кишинев', 'Донецк', 812],
    ['С.Петербург', 'Витебск', 602],
    ['С.Петербург', 'Калининград', 739],
    ['С.Петербург', 'Рига', 641],
    ['Москва', 'Казань', 815],
    ['Москва', 'Ниж.Новгород', 411],
    ['Москва', 'Минск', 690],
    ['Москва', 'Донецк', 1084],
    ['Москва', 'С.Петербург', 664],
    ['Мурманск', 'С.Петербург', 1412],
    ['Мурманск', 'Минск', 2238],
    ['Орел', 'Витебск', 522],
    ['Орел', 'Донецк', 709],
    ['Орел', 'Москва', 368],
    ['Одесса', 'Киев', 487],
    ['Рига', 'Каунас', 267],
    ['Таллинн', 'Рига', 308],
    ['Харьков', 'Киев', 471],
    ['Харьков', 'Симферополь', 639],
    ['Ярославль', 'Воронеж', 739],
    ['Ярославль', 'Минск', 940],
    ['Уфа', 'Казань', 525],
    ['Уфа', 'Самара', 461]
 ]
```



2. Неинформированный поиск

2.1 Поиск в ширину

2.1.1 Список ребер

Воспользуемся nx.bfs_edges, получим список ребер, которые обойдет алгоритм поиска в ширину. Сохраним список ребер до содержащего DESTINATION_NODE, для построения пути

```
bfs_edges = list()
for e in nx.bfs_edges(G_, source=SOURCE_NODE):
    bfs_edges.append(e)
    if e.__contains__(DESTINATION_NODE):
        break
print("all bfs edges:")
for e in nx.bfs_edges(G_, source=SOURCE_NODE):
    print(e)
print("bfs edges till DESTINATION_NODE:")
for e in bfs_edges:
    print(e)
```

Результат:

```
all bfs edges:
('Брест', 'Вильнюс')
('Брест', 'Витебск')
('Брест', 'Калининград')
('Вильнюс', 'Даугавпилс')
('Вильнюс', 'Калиниград')
('Вильнюс', 'Каунас')
('Вильнюс', 'Киев')
('Витебск', 'Вильюс')
('Витебск', 'Воронеж')
('Витебск', 'Волгоград')
('Витебск', 'Ниж.Новгород')
('Витебск', 'С.Петербург')
('Витебск', 'Орел')
('Каунас', 'Рига')
('Киев', 'Житомир')
('Киев', 'Кишинев')
('Киев', 'Одесса')
('Киев', 'Харьков')
('Воронеж', 'Ярославль')
('Ниж.Новгород', 'Москва')
('С.Петербург', 'Мурманск')
('Орел', 'Донецк')
('Рига', 'Таллинн')
('Харьков', 'Симферополь')
('Ярославль', 'Минск')
('Москва', 'Казань')
('Казань', 'Уфа')
('Уфа', 'Самара')
bfs edges till DESTINATION_NODE:
('Брест', 'Вильнюс')
('Брест', 'Витебск')
('Брест', 'Калининград')
('Вильнюс', 'Даугавпилс')
('Вильнюс', 'Калиниград')
('Вильнюс', 'Каунас')
('Вильнюс', 'Киев')
('Витебск', 'Вильюс')
('Витебск', 'Воронеж')
('Витебск', 'Волгоград')
('Витебск', 'Ниж.Новгород')
('Витебск', 'С.Петербург')
('Витебск', 'Орел')
('Каунас', 'Рига')
('Киев', 'Житомир')
('Киев', 'Кишинев')
('Киев', 'Одесса')
('Киев', 'Харьков')
('Воронеж', 'Ярославль')
('Ниж.Новгород', 'Москва')
```

```
('С.Петербург', 'Мурманск')
('Орел', 'Донецк')
('Рига', 'Таллинн')
('Харьков', 'Симферополь')
('Ярославль', 'Минск')
('Москва', 'Казань')
```

2.1.2 Путь

Найдем путь в графе bfs_edges

```
G__ = nx.Graph()
G__.add_edges_from(bfs_edges)
bfs_shortest_path_verts = nx.shortest_path(G__, SOURCE_NODE, DESTINATION_NODE)
bfs_path = list()
for i in range(len(bfs_shortest_path_verts) - 1):
    bfs_path.append((bfs_shortest_path_verts[i], bfs_shortest_path_verts[i +
1]))
print("bfs path:")
print(bfs_path)
```

Результат:

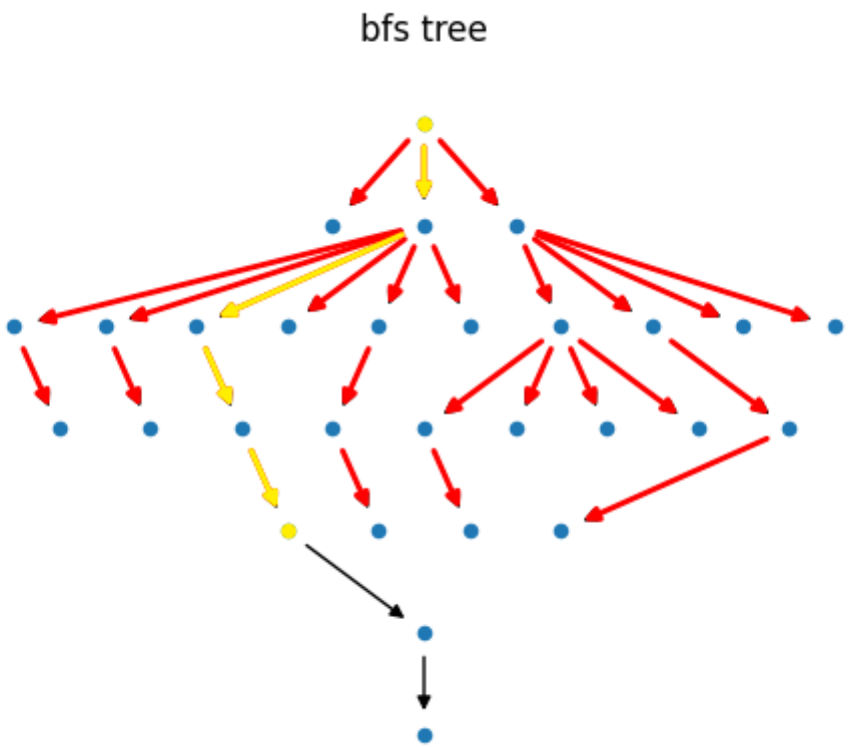
```
bfs path:
[
    ('Брест', 'Витебск'),
    ('Витебск', 'Ниж.Новгород'),
    ('Ниж.Новгород', 'Москва'),
    ('Москва', 'Казань')
]
```

2.1.3 Дерево

Построим дерево, получаещееся в процессе обхода графа алгоритмом BFS. Красным обозначены ребра, которые обходит алгоритм, но не относящиеся непосредственно к основному пути (обозначен желтым)

```
bfs_tree_ = nx.bfs_tree(G_, source=SOURCE_NODE)
for i, layer in enumerate(nx.topological_generations(bfs_tree_)):
    for n in layer:
        bfs_tree_.nodes[n]["layer"] = i
pos_bfs_tree = nx.multipartite_layout(bfs_tree_, subset_key="layer",
align="horizontal")
for k in pos_bfs_tree:
    pos_bfs_tree[k][-1] *= -1
nx.draw_networkx_edges(bfs_tree_, pos_bfs_tree, edgelist=bfs_tree_.edges)
nx.draw_networkx_edges(bfs_tree_, pos_bfs_tree, edgelist=bfs_edges, edge_color=
(1, 0, 0), width=2)
nx.draw_networkx_edges(bfs_tree_, pos_bfs_tree, edgelist=bfs_path,
edge_color='#ffee00', width=2)
nx.draw_networkx_nodes(bfs_tree_, pos_bfs_tree, node_size=20)
nx.draw_networkx_nodes(bfs_tree_, pos_bfs_tree, nodelist=[SOURCE_NODE,
DESTINATION_NODE], node_size=20,
                       node_color='#ffee00')
plt.title("bfs tree")
plt.axis("off")
plt.show()
```

Результат:

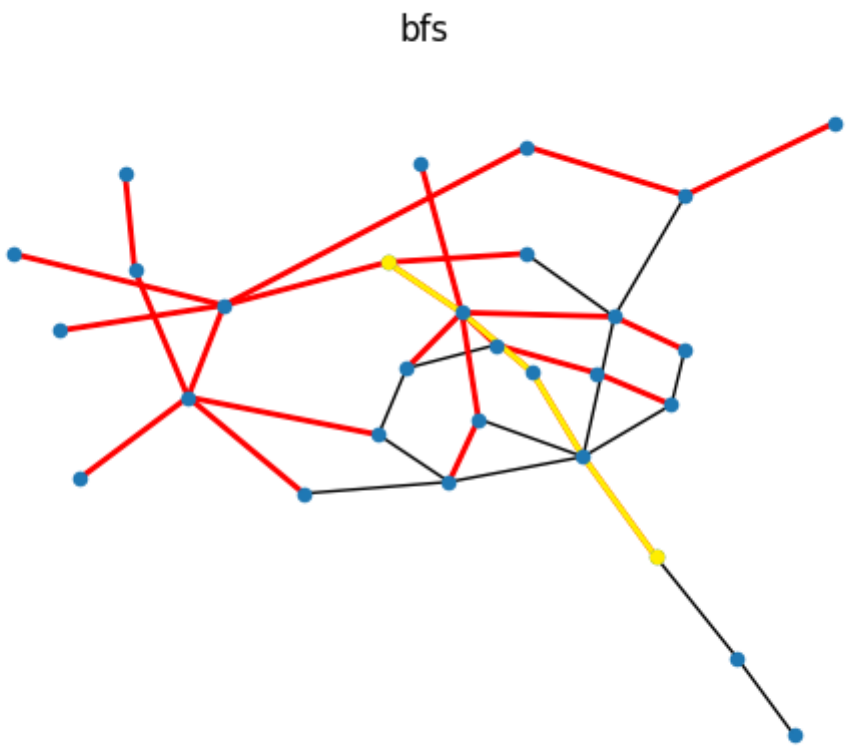


2.1.4 Общая картина

Раскрасим ребра на графе из п.1 по такому же принципу как и раскрасили дерево — красный - ребра, которые мы обошли, желтый - путь

```
nx.draw_networkx_edges(G_, pos, edgelist=G_.edges)
nx.draw_networkx_edges(G_, pos, edgelist=bfs_edges, edge_color=(1, 0, 0),
width=2)
nx.draw_networkx_edges(G_, pos, edgelist=bfs_path, edge_color='#ffee00',
width=2)
nx.draw_networkx_nodes(G_, pos, node_size=20)
nx.draw_networkx_nodes(G_, pos, nodelist=[SOURCE_NODE, DESTINATION_NODE],
node_size=20, node_color='#ffee00')
plt.title("bfs")
plt.axis("off")
plt.show()
```

Результат:



2.2 Поиск в глубину

2.2.1 Список ребер

Воспользуемся nx.dfs_edges, получим список ребер, которые обойдет алгоритм поиска в глубину. Сохраним список ребер до содержащего DESTINATION_NODE, для построения пути (аналогично действиям в пункте 2.1.1)

```
dfs_edges = list()
for e in nx.dfs_edges(G_, source=SOURCE_NODE):
    dfs_edges.append(e)
    if e.__contains__(DESTINATION_NODE):
        break
print("all dfs edges:")
for e in nx.dfs_edges(G_, source=SOURCE_NODE):
    print(e)
print("dfs edges till DESTINATION_NODE:")
for e in dfs_edges:
    print(e)
```

Результат:

```
all dfs edges:
('Брест', 'Вильнюс')
('Вильнюс', 'Даугавпилс')
('Вильнюс', 'Калининград')
('Вильнюс', 'Каунас')
('Каунас', 'Рига')
('Рига', 'С.Петербург')
('С.Петербург', 'Витебск')
('Витебск', 'Вильюс')
('Витебск', 'Воронеж')
('Воронеж', 'Волгоград')
('Волгоград', 'Житомир')
('Житомир', 'Киев')
('Киев', 'Кишинев')
('Кишинев', 'Донецк')
('Донецк', 'Москва')
('Москва', 'Казань')
('Казань', 'Уфа')
('Уфа', 'Самара')
('Москва', 'Ниж.Новгород')
('Москва', 'Минск')
('Минск', 'Мурманск')
('Минск', 'Ярославль')
('Москва', 'Орел')
('Киев', 'Одесса')
('Киев', 'Харьков')
('Харьков', 'Симферополь')
('С.Петербург', 'Калининград')
('Рига', 'Таллинн')
dfs edges till DESTINATION_NODE:
('Брест', 'Вильнюс')
('Вильнюс', 'Даугавпилс')
('Вильнюс', 'Калининград')
('Вильнюс', 'Каунас')
('Каунас', 'Рига')
('Рига', 'С.Петербург')
('С.Петербург', 'Витебск')
('Витебск', 'Вильюс')
('Витебск', 'Воронеж')
('Воронеж', 'Волгоград')
('Волгоград', 'Житомир')
('Житомир', 'Киев')
('Киев', 'Кишинев')
('Кишинев', 'Донецк')
('Донецк', 'Москва')
('Москва', 'Казань')
```

2.2.2 Путь

Найдем путь в графе dfs_edges

```
G__ = nx.Graph()
G__.add_edges_from(dfs_edges)
dfs_shortest_path_verts = nx.shortest_path(G__, SOURCE_NODE, DESTINATION_NODE)
dfs_path = list()
for i in range(len(dfs_shortest_path_verts) - 1):
    dfs_path.append((dfs_shortest_path_verts[i], dfs_shortest_path_verts[i +
1]))
print("dfs path:")
for e in dfs_path:
    print(e)
```

Результат

```
dfs path:
('Брест', 'Вильнюс')
('Вильнюс', 'Каунас')
('Каунас', 'Рига')
('Рига', 'С.Петербург')
('С.Петербург', 'Витебск')
('Витебск', 'Воронеж')
('Воронеж', 'Волгоград')
('Волгоград', 'Житомир')
('Житомир', 'Киев')
('Киев', 'Кишинев')
('Кишинев', 'Донецк')
('Донецк', 'Москва')
('Москва', 'Казань')
```

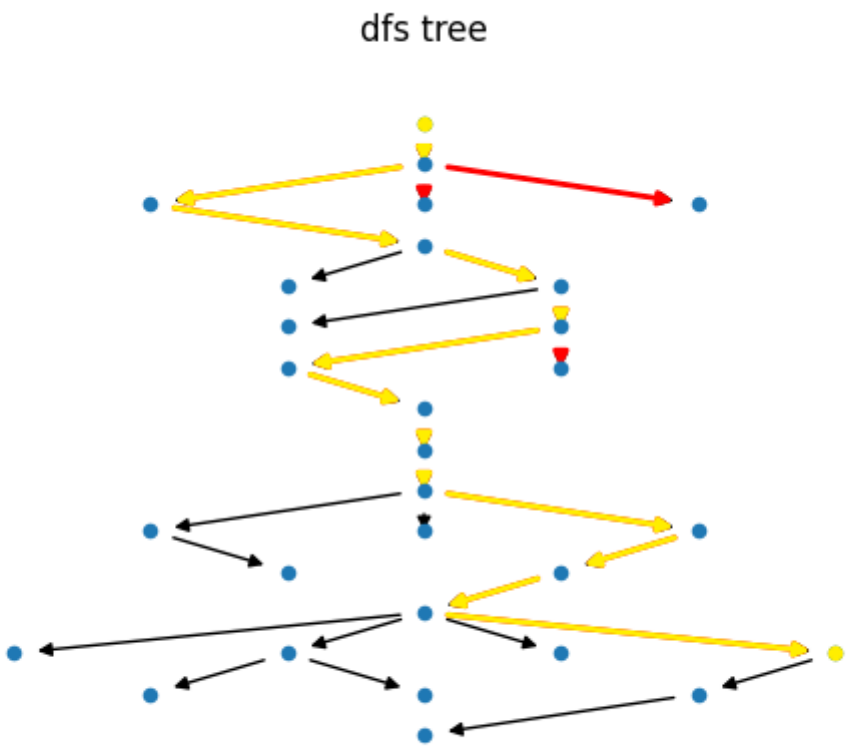
Как видим, путь, полученный обходом dfs, получился длиньше чем при использовании bfs

2.2.3 Дерево

Построим дерево, получаещееся в процессе обхода графа алгоритмом DFS. Красным обозначены ребра, которые обходит алгоритм, но не относящиеся непосредственно к основному пути (обозначен желтым) (аналогично пункту 2.1.3)

```
dfs_tree_ = nx.dfs_tree(G_, source=SOURCE_NODE)
for i, layer in enumerate(nx.topological_generations(dfs_tree_)):
    for n in layer:
        dfs_tree_.nodes[n]["layer"] = i
pos_dfs_tree = nx.multipartite_layout(dfs_tree_, subset_key="layer",
align="horizontal")
for k in pos_dfs_tree:
    pos_dfs_tree[k][-1] *= -1
nx.draw_networkx_edges(dfs_tree_, pos_dfs_tree, edgelist=dfs_tree_.edges)
nx.draw_networkx_edges(dfs_tree_, pos_dfs_tree, edgelist=dfs_edges, edge_color=
(1, 0, 0), width=2)
nx.draw_networkx_edges(dfs_tree_, pos_dfs_tree, edgelist=dfs_path,
edge_color='#ffee00', width=2)
nx.draw_networkx_nodes(dfs_tree_, pos_dfs_tree, node_size=20)
nx.draw_networkx_nodes(dfs_tree_, pos_dfs_tree, nodelist=[SOURCE_NODE,
DESTINATION_NODE], node_size=20,
                       node_color='#ffee00')
plt.title("dfs tree")
plt.axis("off")
plt.show()
```

Результат

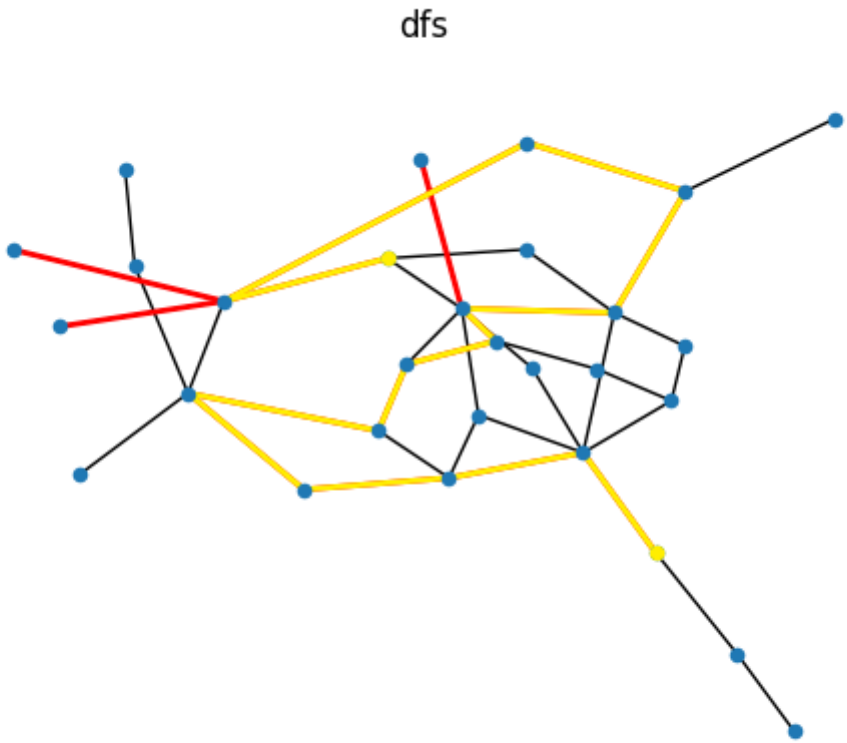


2.2.4 Общая картина

Совершим действия аналогичные п. 2.1.4

```
nx.draw_networkx_edges(G_, pos, edgelist=G_.edges)
nx.draw_networkx_edges(G_, pos, edgelist=dfs_edges, edge_color=(1, 0, 0),
width=2)
nx.draw_networkx_edges(G_, pos, edgelist=dfs_path, edge_color='#ffee00',
width=2)
nx.draw_networkx_nodes(G_, pos, node_size=20)
nx.draw_networkx_nodes(G_, pos, nodelist=[SOURCE_NODE, DESTINATION_NODE],
node_size=20, node_color='#ffee00')
plt.title("dfs")
plt.axis("off")
plt.show()
```

Результат



2.3 Поиск с ограничением глубины

2.3.1 Список ребер

Воспользуемся функцией nx.dfs_tree из п.2.2.1. Благодаря аргументу depth_limit, мы можем ограничить глубину дерева, которое будет построено в ходе работы алгоритмо DFS. Ограничим глубину числом 8.

```
dfs_tree_ = nx.dfs_tree(G_, source=SOURCE_NODE, depth_limit=8)
G__ = nx.Graph()
G__.add_edges_from(dfs_tree_.edges)
dfs_shortest_path_verts = nx.shortest_path(G__, SOURCE_NODE, DESTINATION_NODE)
dfs_path8 = list()
for i in range(len(dfs_shortest_path_verts) - 1):
    dfs_path8.append((dfs_shortest_path_verts[i], dfs_shortest_path_verts[i +
1]))
print("dfs tree limit=8 path:")
for e in dfs_path8:
    print(e)
for i, layer in enumerate(nx.topological_generations(dfs_tree_)):
    for n in layer:
        dfs_tree_.nodes[n]["layer"] = i
pos_dfs_tree = nx.multipartite_layout(dfs_tree_, subset_key="layer",
align="horizontal")
for k in pos_dfs_tree:
    pos_dfs_tree[k][-1] *= -1
nx.draw_networkx_edges(dfs_tree_, pos_dfs_tree, edgelist=dfs_tree_.edges)
nx.draw_networkx_nodes(dfs_tree_, pos_dfs_tree, node_size=20)
nx.draw_networkx_nodes(dfs_tree_, pos_dfs_tree, nodelist=[SOURCE_NODE],
node_color='#ffee00',
                       node_size=20)
nx.draw_networkx_nodes(dfs_tree_, pos_dfs_tree, nodelist=[DESTINATION_NODE],
node_color='#ff0000',
                       node_size=20)
nx.draw_networkx_edges(dfs_tree_, pos_dfs_tree, edgelist=dfs_path8,
edge_color='#ffee00', width=2)
plt.title("dfs tree limit=8 path")
plt.axis("off")
plt.show()
```

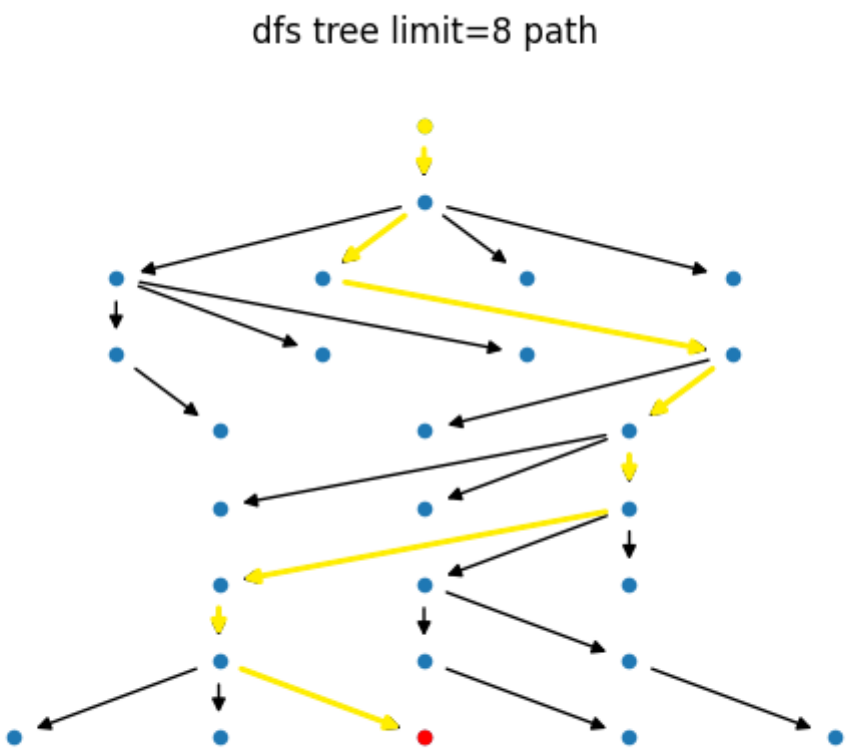
2.3.2 Путь

Результат выполнения кода из п.2.3.1:

```
dfs tree limit=8 path:
('Брест', 'Вильнюс')
('Вильнюс', 'Каунас')
('Каунас', 'Рига')
('Рига', 'С.Петербург')
```

2.3.3 Дерево

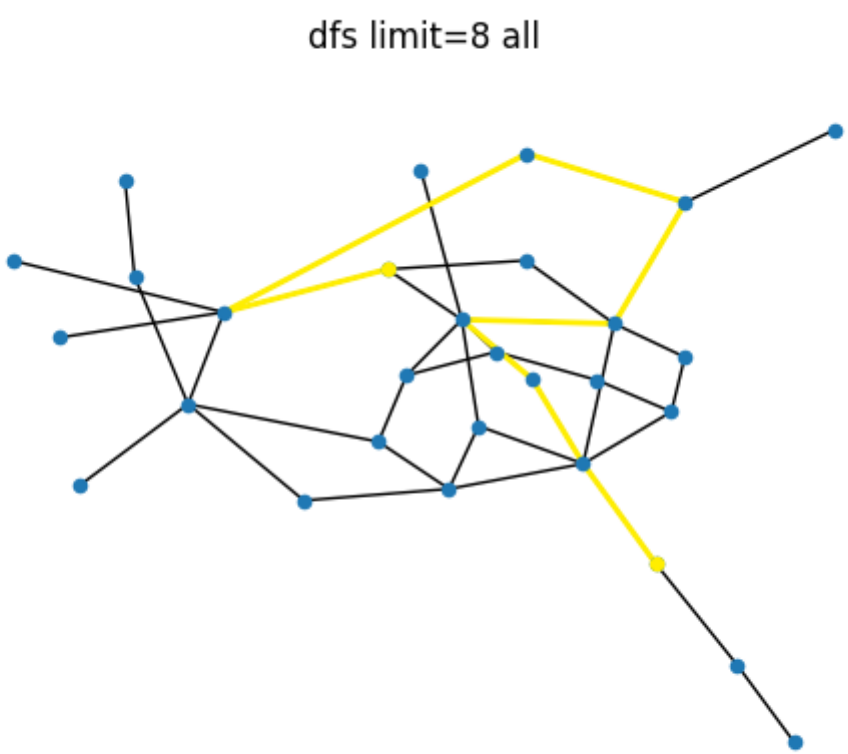
Результат выполнения кода из п.2.3.1:



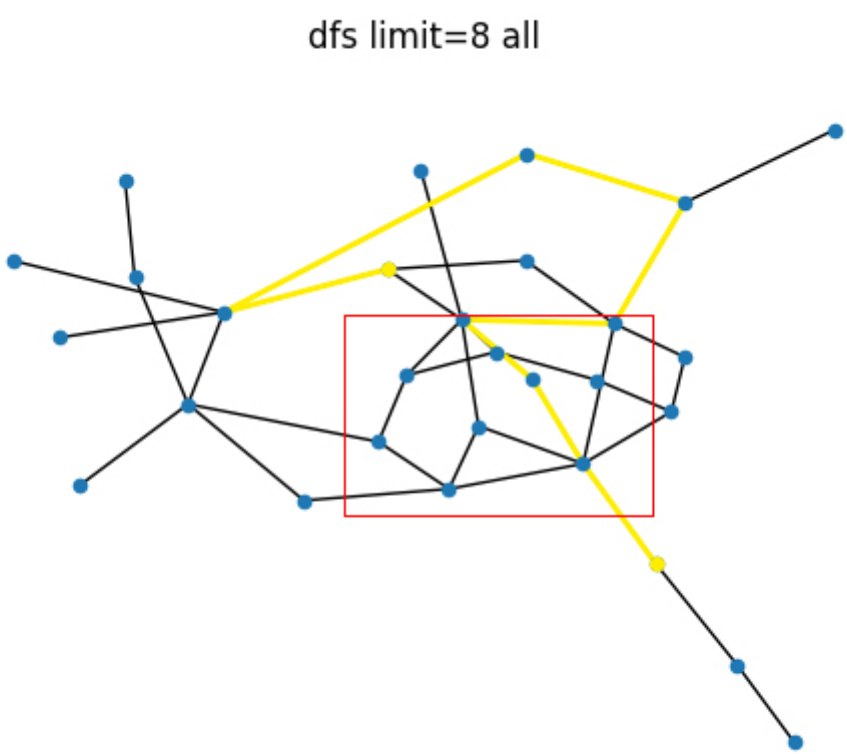
2.3.4 Общая картина

```
nx.draw_networkx_edges(G_, pos, edgelist=G_.edges)
nx.draw_networkx_edges(G_, pos, edgelist=dfs_path8, edge_color='#ffee00',
width=2)
nx.draw_networkx_nodes(G_, pos, node_size=20)
nx.draw_networkx_nodes(G_, pos, nodelist=[SOURCE_NODE, DESTINATION_NODE],
node_size=20, node_color='#ffee00')
plt.title("dfs limit=8 all")
plt.axis("off")
plt.show()
```

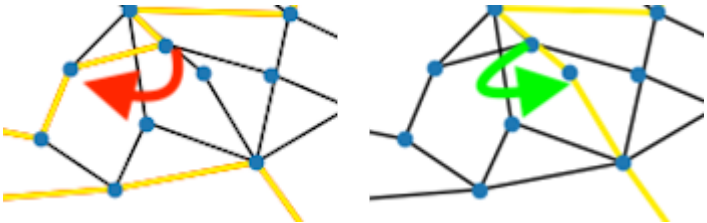
Результат:



Как видим, по сравнению с п.2.2.4, ограничение глубины сократило путь за счет того, что мы не стали спускаться ниже по графу. Рассмотрим ближе участок:



Сравним результаты:



На рисунке слева видим, что путь получается длиньше из-за того, что алгоритм на одном из этапов выбирает пойти глубже (красная стрелка). Эффективнее было бы пойти по сравнению с этим по зеленой.