# 1. Введение

1) Кратчайшие пути на графе.

2) Код, решающий данную задачу

3) Скриншот программы

# 2. Ход работы

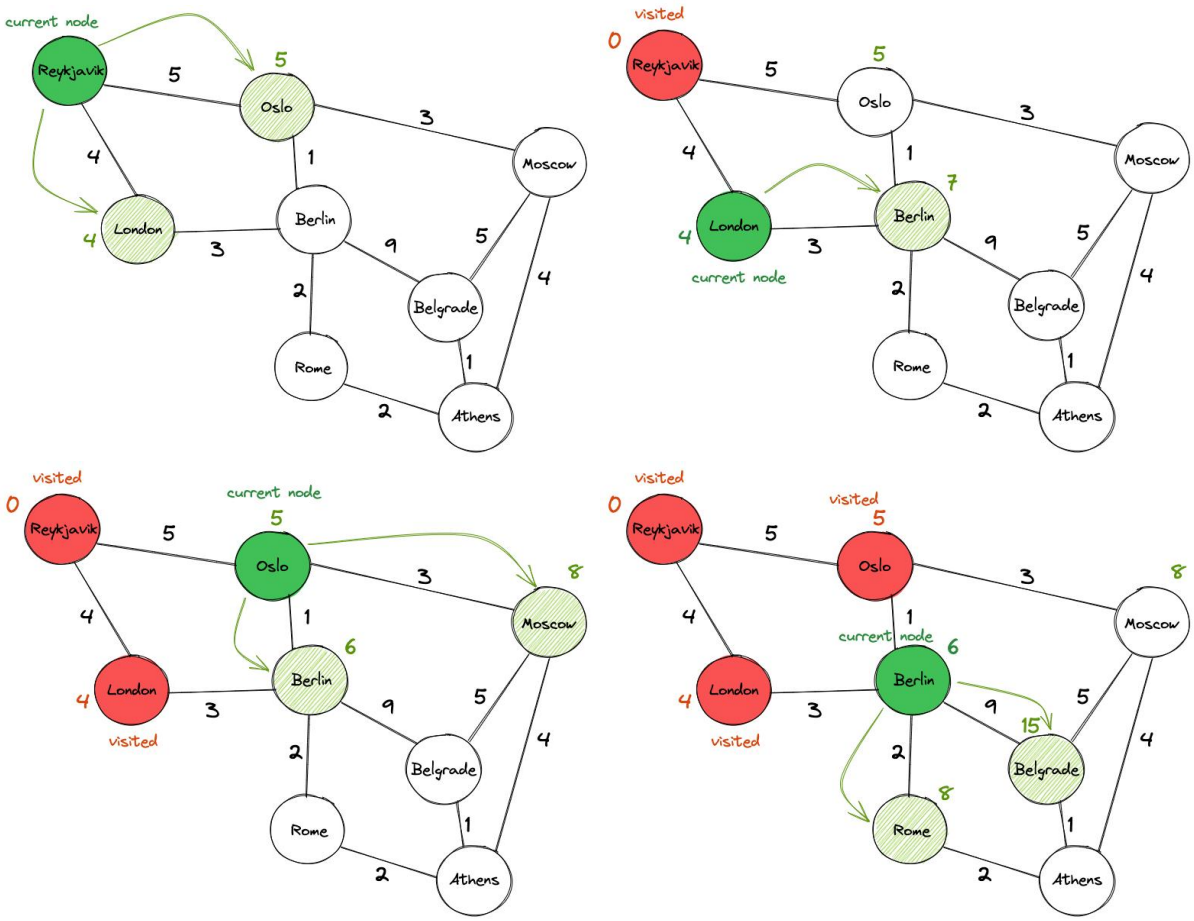
## 2.1. Код приложения

```
import sys

class Graph(object):
    def __init__(self, nodes, init_graph):
        self.nodes = nodes
        self.graph = self.construct_graph(nodes, init_graph)

    def construct_graph(self, nodes, init_graph):
        '''
        Этот метод обеспечивает симметричность графика. Другими словами, если
        существует путь от узла A к B со значением V, должен быть путь от
        узла B к узлу A со значением V.
        '''
        graph = {}
        for node in nodes:
            graph[node] = {}

        graph.update(init_graph)

        for node, edges in graph.items():
            for adjacent_node, value in edges.items():
                if graph[adjacent_node].get(node, False) == False:
                    graph[adjacent_node][node] = value

        return graph

    def get_nodes(self):
        "Возвращает узлы графа"
        return self.nodes

    def get_outgoing_edges(self, node):
        "Возвращает соседей узла"
```

```
        connections = []
        for out_node in self.nodes:
            if self.graph[node].get(out_node, False) != False:
                connections.append(out_node)
        return connections

    def value(self, node1, node2):
        "Возвращает значение ребра между двумя узлами."
        return self.graph[node1][node2]
def dijkstra_algorithm(graph, start_node):
    unvisited_nodes = list(graph.get_nodes())

    # Мы будем использовать этот словарь, чтобы сэкономить на посещении
    каждого узла и обновлять его по мере продвижения по графику
    shortest_path = {}

    # Мы будем использовать этот dict, чтобы сохранить кратчайший
    известный путь к найденному узлу
    previous_nodes = {}

    # Мы будем использовать max_value для инициализации значения
    "бесконечности" непосещенных узлов
    max_value = sys.maxsize
    for node in unvisited_nodes:
        shortest_path[node] = max_value
    # Однако мы инициализируем значение начального узла 0
    shortest_path[start_node] = 0

    # Алгоритм выполняется до тех пор, пока мы не посетим все узлы
    while unvisited_nodes:
        # Приведенный ниже блок кода находит узел с наименьшей оценкой
        current_min_node = None
        for node in unvisited_nodes: # Iterate over the nodes
            if current_min_node == None:
                current_min_node = node
            elif shortest_path[node] < shortest_path[current_min_node]:
                current_min_node = node

        # Приведенный ниже блок кода извлекает соседей текущего узла и
        обновляет их расстояния
        neighbors = graph.get_outgoing_edges(current_min_node)
        for neighbor in neighbors:
            tentative_value = shortest_path[current_min_node] +
            graph.value(current_min_node, neighbor)
```

```
            if tentative_value < shortest_path[neighbor]:
                shortest_path[neighbor] = tentative_value
                # We also update the best path to the current node
                previous_nodes[neighbor] = current_min_node

        # После посещения его соседей мы отмечаем узел как "посещенный"
        unvisited_nodes.remove(current_min_node)

    return previous_nodes, shortest_path
def print_result(previous_nodes, shortest_path, start_node, target_node):
    path = []
    node = target_node

    while node != start_node:
        path.append(node)
        node = previous_nodes[node]

   # Добавить начальный узел вручную
    path.append(start_node)

    print("Найден следующий лучший маршрут с ценностью
    {}.".format(shortest_path[target_node]))
    print(" -> ".join(reversed(path)))
nodes = ["Reykjavik", "Oslo", "Moscow", "London", "Rome", "Berlin", "Belgrade",
"Athens"]

init_graph = {}
for node in nodes:
    init_graph[node] = {}

init_graph["Reykjavik"]["Oslo"] = 5
init_graph["Reykjavik"]["London"] = 4
init_graph["Oslo"]["Berlin"] = 1
init_graph["Oslo"]["Moscow"] = 3
init_graph["Moscow"]["Belgrade"] = 5
init_graph["Moscow"]["Athens"] = 4
init_graph["Athens"]["Belgrade"] = 1
init_graph["Rome"]["Berlin"] = 2
init_graph["Rome"]["Athens"] = 2
graph = Graph(nodes, init_graph)
previous_nodes, shortest_path = dijkstra_algorithm(graph=graph,
start_node="Reykjavik")
print_result(previous_nodes, shortest_path, start_node="Reykjavik",
target_node="Belgrade")
```

## 2.2. Пример решения




# 3. Код после выполнения программы

```
18      for node, edges in graph.items():
19
Найден следующий лучший маршрут с ценностью 11.
Reykjavik -> Oslo -> Berlin -> Rome -> Athens -> Belgrade


...Program finished with exit code 0
```

# 4. Пример библиографических ссылок

Для написания «программы» необходимо изучить [1], для использования LaTeX лучше почитать [2], а для работы с Git [3].

# Список литературы

[1] Самир Мадхаван Mastering Python for Data Science : Изд. Packt Publishing, 2015г.

[2] Львовский С.М. Набор и верстка в системе LaTeX. — 3-е издание, исправленное и дополненное, 2003 г.

[3] Скоттом Чаконом, Беном Штраубом Pro Git —2-е издание 2014г.