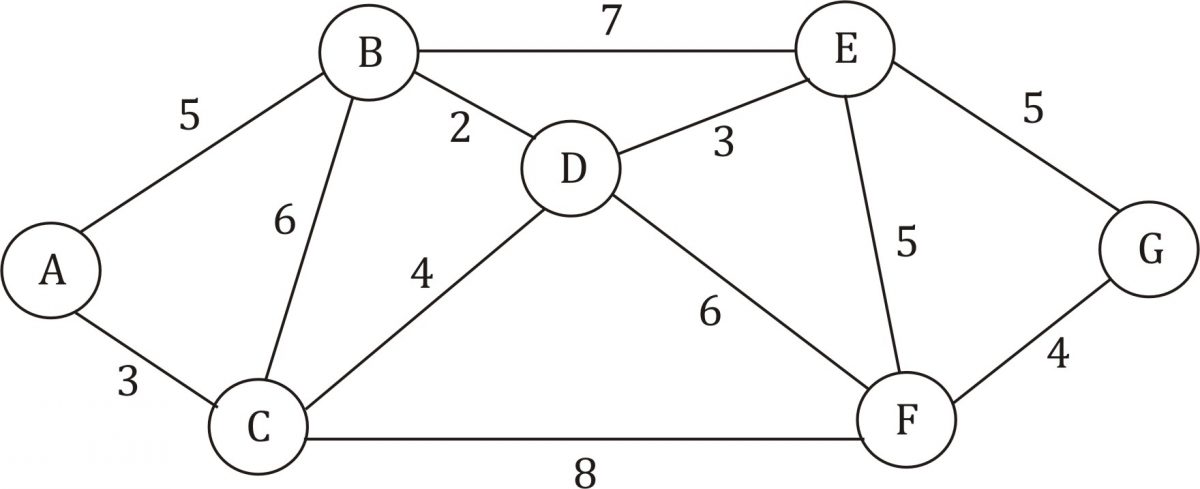
**Модуль 10. Практическая работа (4 ак. ч.)**

**Вариант 2 Поиск маршрута между двумя пунктами**

**(Реализация алгоритма Дейкстры на Python)**



Что общего у устройств GPS‑навигации с веб‑сайтами для бронирования рейсов? Оказывается, очень много! Во‑первых, обе технологии используют алгоритм Дейкстры для поиска кратчайшего пути. После простого русского языка, на котором всё будет понятно, вы увидите, что кодирование на Python не представляет больших сложностей.

## Что такое алгоритм Дейкстры?

В 1956 году у голландского программиста Эдсгера В. Дейкстры возникла практическая задача — найти самый короткий путь из Роттердама в Гронинген. Однако, он не просто сверился с картой для рассчёта расстояния, которые ему нужно было пройти, использую длину дорог.

Вместо этого, Дейкстра поступил, как [информатик](https://ru.wikipedia.org/wiki/Special:Search/%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA): он абстрагировался от проблемы, отбросив такие мелочи, как путешествие из города А в город Б. Он сформулировал более общую проблему поиска по графу.

Вот так и родился [алгоритм Дейкстры](https://ru.wikipedia.org/wiki/Special:Search/%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC%20%D0%94%D0%B5%D0%B9%D0%BA%D1%81%D1%82%D1%80%D1%8B).

Алгоритм Дейкстры — это популярный алгоритм поиска, используемый для определения кратчайшего пути между двумя узлами в графе.

В исходном сценарии граф представлял Нидерланды, узлы графа представляли разные голландские города, а рёбра представляли дороги между городами. **Алгоритм Дейкстры можно использовать для решения любой задачи, которая может быть представлена в виде графа**.

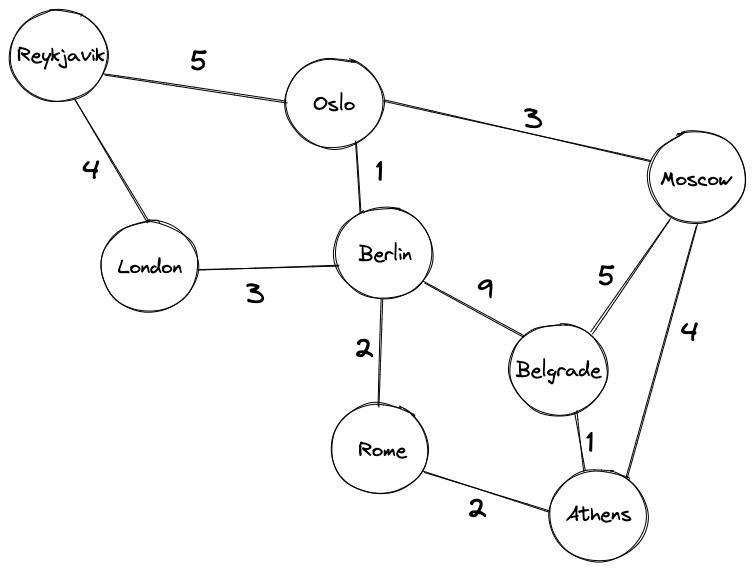
Предложения друзей в социальных сетях, маршрутизация пакетов через Интернет или поиск пути через лабиринт — все это может сделать алгоритмом Дейкстры.

Но как на самом деле это работает?

Напомним, что алгоритм Дейкстры для графов, а это означает, что он может решить проблему, только если она может быть представлена в графоподобном виде. Пример, который будет здесь показан, возможно, самый интуитивно понятный — кратчайший путь между двумя городами.

Мы будем работать с картой, показанной ниже, для поиска наилучшего маршрута между двумя европейскими городами Рейкьявиком и Белградом.

Для простоты представим, что все города связаны дорогами (реальный маршрут должен включать как минимум один паром).



Обратите внимание на следующее:

* Каждый город — это узел.
* Каждая дорога — это ребро.
* У каждой дороги есть своя ценность. Это может быть расстояние между городами, плата за проезд или интенсивностью движения. Как правило, мы отдаем предпочтение рёбрам с более низкими значениями. В нашем конкретном случае связанное значение определяется расстоянием между двумя городами.

Вы должны уже заметить, что добраться из Рейкьявика до Белграда напрямую невозможно. Это сделало бы наши упражнения бессмысленными. Но есть несколько путей из Рейкьявика в Белград, которые проходят через другие города:

* Reykjavik → Oslo → Berlin → Belgrade
* Reykjavik → London → Berlin → Rome → Athens → Belgrade
* Reykjavik → London → Berlin → Rome → Athens → Moscow → Belgrade

Каждый из этих путей заканчивается в Белграде, но все они имеют разные ценности. Мы можем использовать алгоритм Дейкстры, чтобы найти путь с наименьшим общим значением.

Прежде чем погрузиться в код, посмотрим на алгоритм Дейкстры с высоты птичьего полета.

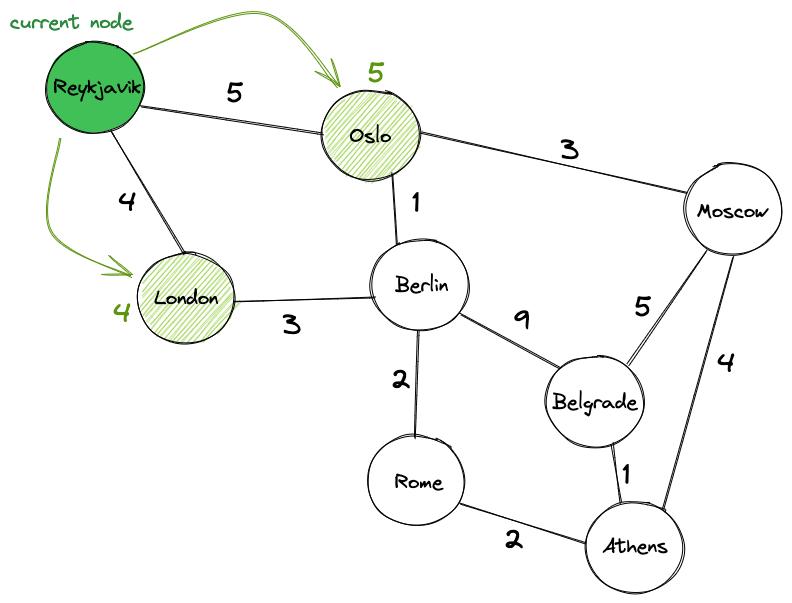
Сначала инициализируем алгоритм:

1. Устанавливаем Рейкьявик в качестве начального узла.
2. Устанавливаем расстояния между Рейкьявиком и всеми другими городами в **∞ (бесконечность)**, за исключением расстояния между Рейкьявиком и самим собой, которое принимаем равным **0**.

После этого, итеративно выполняем следующие шаги:

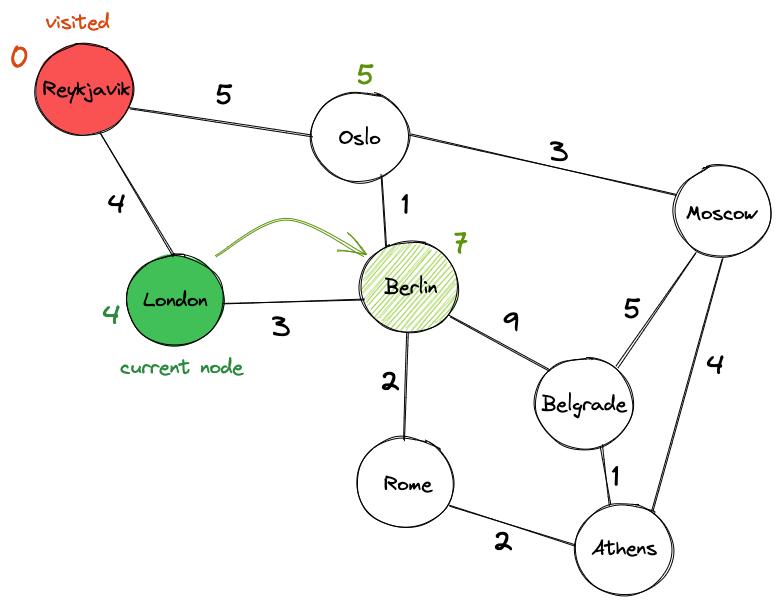
1. Выбираем узел с наименьшим значением в качестве «текущего узла» и посещаем всех соседей. При посещении каждого соседа, обновляем их ориентировочное расстояние от начального узла.
2. Как только мы посетим всех соседей текущего узла и обновили их расстояния, помечаем текущий узел как **«visited»** **«посещенный»**). Отметка узла как «посещенного» означает, что найдена его окончательная ценность.
3. Вернемся к первому шагу и повторяем до тех пор, пока не посетим все узлы графа.

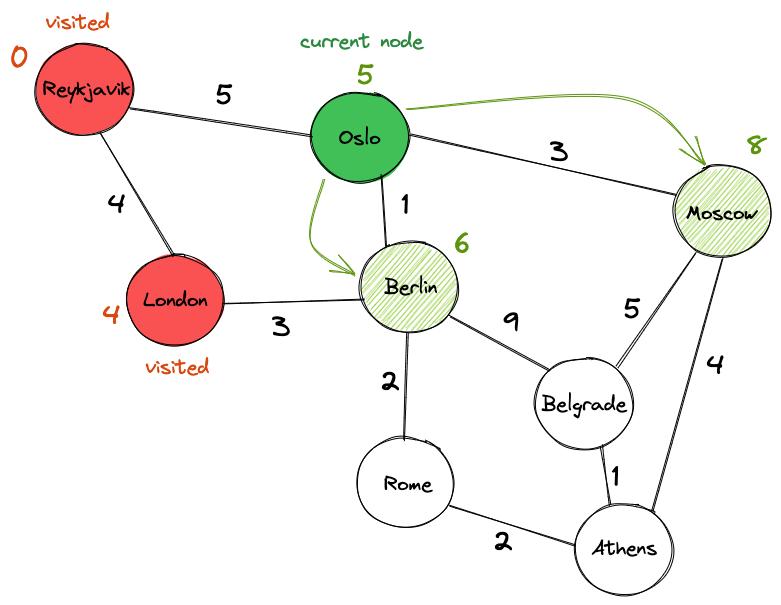
В нашем примере начинаем с отметки Рейкьявика, поскольку его значение равно 0, как «текущего узла». Далее посетим два соседних узла **Рейкьявика**: **Лондон** и **Осло**. В начале алгоритма их значения устанавливаются на бесконечность, но, когда мы посещаем узлы, то обновляем значение для Лондона до 4 и Осло до 5.



Затем отмечаем Рейкьявик как «посещенный». Мы знаем, что его окончательная стоимость равна нулю, и нам больше не нужно его посещать. Продолжаем со следующего узла с наименьшим значением, которым является **Лондон**.

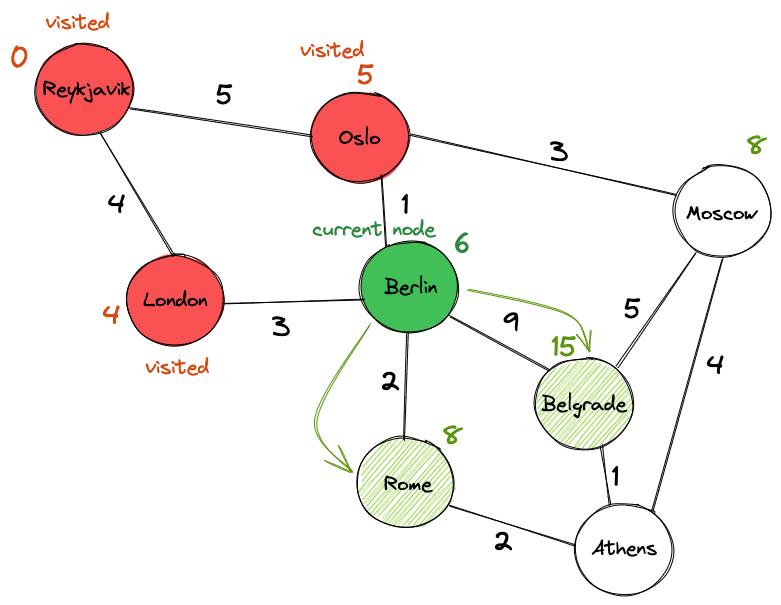
Посещаем все соседние узлы **Лондона**, которые не помечены как «посещенные». Соседи **Лондона** — **Рейкьявик** и **Берлин**. Но мы игнорируем **Рейкьявик**, потому что мы уже были в нем. Вместо этого обновляем значение **Берлина**, добавляя значение ребра, соединяющего **Лондон** и **Берлин** (3), к значению **Лондона** (4), что дает нам значение 7.



Отмечаем **Лондон** как посещенный и выбираем следующий узел — **Осло**. Посещаем соседей **Осло** и обновляем их ценности. Оказывается, можно лучше добраться до **Берлина** через **Осло** (со значением 6), чем через **Лондон**, поэтому соответствующим образом обновляем его значение. Также обновляем текущее значение **Москвы** с бесконечности до 8.  


Отмечаем **Осло** как «посещенный» и обновляем его окончательное значение до 5. Между **Берлином** и **Москвой** выбираем **Берлин** в качестве следующего узла, потому что его значение (6) ниже, чем у **Москвы** (8).

Действуем так же, как и раньше: посещаем **Рим** и **Белград** и обновляем их предварительные значения, прежде чем пометить **Берлин** как «посещенный» и перейти к следующему городу.



Обратите внимание, что мы уже нашли путь из **Рейкьявика** в **Белград** со значением 15! Но лучший ли он?

В конце концов, это не так. Пропустим остальные шаги, а для вас это упражнение. Лучшим путем оказывается **Рейкьявик → Осло → Берлин → Рим → Афины → Белград** со значением 11.

Теперь, посмотрим, как это реализовать на Python.

## Алгоритм Дейкстры на Python

#### 1) КЛАСС GRAPH

Сначала создадим класс Graph. Этот класс не охватывает никакой логики алгоритма Дейкстры, но сделает реализацию алгоритма более лаконичной.

Реализуем граф как словарь Python. Ключи словаря будут соответствовать городам, а его значения будут соответствовать рёбрам, где записывают расстояния до других городов на графе.

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]

#### 2) АЛГОРИТМ ДЕЙКСТРЫ

Далее реализуем алгоритм Дейкстры. Начнем с определения функции.

**def** dijkstra\_algorithm(graph, start\_node):

Функция принимает два аргумента: graph и start\_node. graph — это экземпляр класса Graph, который мы создали на предыдущем шаге, тогда как start\_node — это узел, с которого мы начнем вычисления. Мы вызовем метод get\_nodes() для инициализации списка непосещенных узлов:

unvisited\_nodes = list(graph.get\_nodes())

Затем создадим два словаря, shorttest\_path и previous\_nodes:

* Shorttest\_path будет хранить наиболее известную стоимость посещения каждого города на графике, начиная с start\_node.
* Вначале стоимость начинается с бесконечности, но мы будем обновлять значения по мере продвижения по графику.
* previous\_nodes будет хранить траекторию текущего наиболее известного пути для каждого узла.
* Например, если нам известен лучший способ добраться до Берлина через Осло, previous\_nodes[«Берлин»] вернет «Осло», а previous\_nodes[«Осло»] вернет «Рейкьявик».
* Мы будем использовать этот словарь, чтобы найти кратчайший путь.

shortest\_path = {}

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

Теперь можно запустить алгоритм. Помните, что алгоритм Дейкстры выполняется до тех пор, пока не посетит все узлы в графе, поэтому представим это как условие выхода из цикла while.

**while** unvisited\_nodes:

Теперь алгоритм может начать посещать узлы. Приведенный ниже блок кода сначала инструктирует алгоритм найти узел с наименьшим значением.

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

После этого алгоритм посещает всех соседей узла, которые еще не посещены. Если новый путь к соседу лучше, чем текущий лучший путь, алгоритм вносит корректировки в словари shorttest\_path и previous\_nodes.

# 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

# Мы также обновляем лучший путь к текущему узлу

previous\_nodes[neighbor] = current\_min\_node

После посещения всех его соседей мы можем пометить текущий узел как «посещенный»:

unvisited\_nodes.remove(current\_min\_node

Наконец, мы можем вернуть два словаря:

**return** previous\_nodes, shortest\_path

#### 3) ВСПОМОГАТЕЛЬНАЯ ФУНКЦИЯ

Наконец, нам нужно создать функцию, которая печатает результаты. Эта функция будет принимать два словаря, возвращаемые функцией dijskstra\_algorithm, а также имена начального и целевого узлов. Он будет использовать два словаря, чтобы найти лучший путь и сделать его оценку.

**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]

# Add the start node manually

path.append(start\_node)

print("Найден следующий лучший маршрут {}.".format(shortest\_path[target\_node]))

print(" -> ".join(reversed(path)))

#### АЛГОРИТМ В ДЕЙСТВИИ

Теперь посмотрим, как работает алгоритм. Вручную инициализируем узлы и их рёбра.

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

Будем использовать эти значения для создания объекта класса Graph.

graph = Graph(nodes, init\_graph)

Когда наш граф полностью построен, можно передать его функции dijkstra\_algorithm().

previous\_nodes, shortest\_path = dijkstra\_algorithm(graph=graph, start\_node="Reykjavik")

А теперь распечатываем результаты:

print\_result(previous\_nodes, shortest\_path, start\_node="Reykjavik", target\_node="Belgrade")

Вот и все! Не стесняйтесь экспериментировать с кодом. Например, можно добавить больше узлов к графу, настроить значения ребер или выбрать разные начальный и конечный города, считать города и расстояния из файла или базы данных.

#### ПОЛНЫЙ КОД ПРИЛОЖЕНИЯ ДЛЯ ПРОВЕРКИ АЛГОРИТМА ДЕЙКСТРЫ

**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")