МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное
учреждение высшего образования

«Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского»
(ННГУ)

Институт информационных технологий, математики и механики

Центр прикладных информационных технологий

Направление подготовки: «Фундоментальная информатика и информационные технологии»
Профиль подготовки: «...»

ЛАБОРАТОРНАЯ РАБОТА

на тему:

«Алгоритм Дейкстры на 3-куче и на 15-куче»

Выполнил:
студент группы 3821Б1ФИ3
Сафронов М. А.

:
Преподаватель
Уткин Г. В.

Нижний Новгород
2023 г.

Содержание

1 Введение

Поставлена задача "Алгоритм Дейкстры на троичной и пятнадцатиричной куче". Для того чтобы разобраться в этой теме, введем некоторые понятия, которые понадобяться в процессе выполнения.

Кучи - являются важными структурами данных, которые широко используются в области алгоритмов и программирования.

Алгоритм Дейкстры — это один из наиболее популярных алгоритмов, применяемых для поиска кратчайшего пути в графе. Комбинируя эти два концепта, мы получаем интересный подход: алгоритм Дейкстры, использующий 3-кучу и 15- кучу.

Одна из главных задач алгоритма Дейкстры заключается в нахождении кратчайшего пути между двумя узлами во взвешенном графе. Кратчайший путь определяется как путь с наименьшей суммой весов ребер. Алгоритм Дейкстры решает эту задачу путем осуществления пошагового обхода графа из начального узла и нахождения кратчайших путей до остальных узлов.

Введение в 3-кучу и 15-кучу: 3-куча и 15-куча представляют собой особые варианты куч, которые дополнительно учитывают третий и пятнадцатый наименьшие элементы соответственно. Эти усовершенствованные кучи позволяют ускорить алгоритм Дейкстры, разработанный Эдсгером Дейкстрой, и сделать его более эффективным при работе с большими объемами данных.

Одна из главных особенностей использования куч в алгоритме Дейкстры заключается в оптимизации операций вставки нового элемента и удаления минимального элемента из кучи. Учитывая третий и пятнадцатый наименьшие элементы вместе с минимальным элементом, мы можем более эффективно оптимизировать выбор наименьшего пути в ходе выполнения алгоритма Дейкстры.

Изначально алгоритм Дейкстры разработан для работы с обычными кучами, однако использование 3-кучи и 15-кучи позволяет достичь еще более высокой производительности и эффективности при поиске кратчайшего пути в графе.

Таким образом, в данной лабораторной работе мы исследуем и реализуем алгоритм Дейкстры, используя 3-кучу и 15-кучу, и оценим

2 Определение графа

Определение графа
Пусть $G = (V, E, W)$ - ориентированный граф без петель со взвешанными ребрами, где множество вершин $V = (1, ..., n)$, множество ребер $E \in V * V, |E| = m$, и весовая функция $W(u, v)$ каждому ребру $(u, v) \in E$ ставит в соответсвии его вес - неотрицательное число. Требуется найти кратчайшие пути от заданной вершины $s \in V$ до всех остальных вершин.

3 Определение Алгоритма Дейкстры

Алгоритм дейкстры служит для нахождения кратчайшего пути, в нашем случае на графах. Находит кратчайшее расстояние от одной из вершин графа до всех остальных. Работает только для графов без рёбер отрицательного веса. Сложность $O(n^2)$ Задача о кратчайших путях заключается в поиске кратчайшего пути между двумя вершинами в графе. Алгоритм Дейкстры - это один из алгоритмов, который позволяет решать эту задачу. Он работает следующим образом:

1. Начинаем с вершины, из которой нужно найти кратчайший путь.
2. Для каждой смежной вершины вычисляем расстояние от начальной вершины до этой вершины.
3. Если это расстояние меньше, чем уже известное расстояние до этой вершины, то обновляем значение расстояния.
4. Повторяем шаги 2-3 для всех смежных вершин.
5. Повторяем шаги 2-4 для всех вершин графа.

4 Определенеи кучи

Куча (или бинарная куча) - это структура данных, которая представляет собой полное бинарное дерево, в котором каждый узел имеет значение, которое меньше или равно значению его потомков. Куча может

быть использована для реализации алгоритмов сортировки, таких как сортировка кучей (heap sort), а также для решения других задач, таких как поиск максимального или минимального элемента.

Троичная куча и **пятнадцатиричная куча** - это разновидности куч, которые отличаются от обычной кучи тем, что они имеют более сложную структуру. Троичная куча позволяет хранить элементы в виде бинарного дерева, в котором каждый узел имеет три потомка. Пятнадцатиричная куча - это куча, в которой каждый узел имеет 15 потомков. Основное отличие между троичной кучей и обычной бинарной кучей заключается в том, что в троичной куче каждый узел имеет больше потомков, что позволяет уменьшить высоту дерева и ускорить операции вставки и удаления элементов. Кроме того, троичная куча может быть более эффективной, чем бинарная куча, когда требуется хранить большое количество элементов.

Пятнадцатиричная куча - это структура данных, которая является модификацией бинарной кучи и троичной кучи, в которой каждый узел имеет не два или три, а пятнадцать потомков.

Однако, пятнадцатиричная куча не является стандартной структурой данных, и ее использование не распространено. Это связано с тем, что узлы с таким большим количеством потомков могут быть сложными для обработки и хранения, что может привести к увеличению времени выполнения операций вставки, удаления и поиска элементов.

5 Результаты

5.1 Проверка алгоритма Дейкстры. Первый тест.

Сравнение алгоритма Дейкстры на троичной и пятнадцатиричной куче. Проверка коректности работы алгоритма дейкстры на малых данных. Создаем граф с 8-ю вершинами и с 11-ю ребрами.

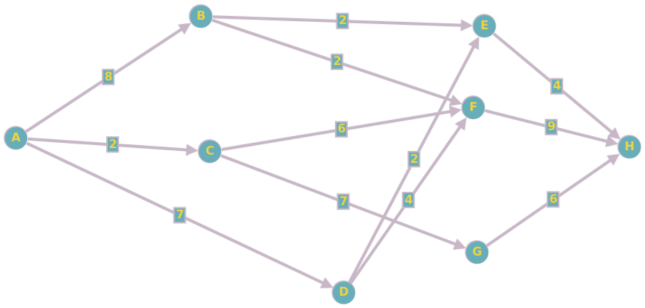

Рис. 1: Граф из первого теста

```
Результат работы программы:
{'A': 0, 'B': 8, 'C': 2, 'D': 7, 'F': 8, 'G': 9, 'E': 9, 'H': 13}
```

Не трудно заметить, что вывод верный и действительно вес кратчайшего пути от A до H составляет 13.

5.2 Проверка алгоритма Дейкстры. Второй тест

, похож на предыдущий только немного больше данных, чтобы убедиться в корректности алгоритма и также сравнить отличия троичное кучи от пятнадцатиричной кучи.

На картинке 1.2 можно увидеть граф с большим количеством вершин и

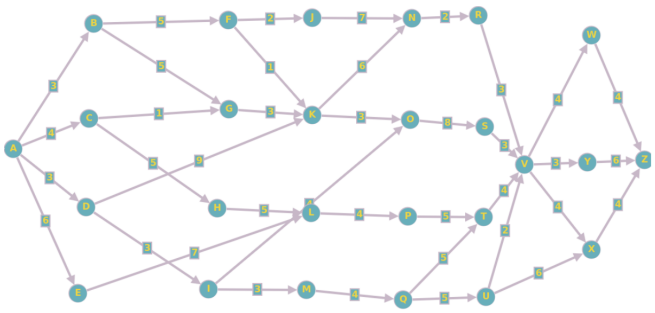

Рис. 2: Граф из второго теста

ребер, чем на предыдущем примере. Применив алгоритм Дейкстры для графа на 3-куче и 15-куче получим следующий вывод.

```
Результат работы программы:
{'A': 0, 'B': 3, 'C': 4, 'D': 3, 'E': 6, 'F': 8, 'G': 5, 'K': 8, 'I': 6, 'H': 9, 'L': 1
second_test:
3-куча: 0.039903 сек.
15-куча: 0.075074 сек.
```

Отсюда делаем вывод, что алгоритм Дейкстры так-же работает корректно и теперь займемся временем работы алгоритма, в данном примере алгоритм на 3-куче лидирует по времени.

Далее проведем значительное количество тестов для исследования.

5.3 Результаты тестирование алгоритма Дейкстры на различных входных данных

В этом разделе приведены таблицы со временем работы, отсюда можно сделать вывод как ведет себя алгоритм на различных входных данных

по времени. Тест производится 10 раз дабы более явно видеть отличия между разными реализациями. При вводе больших данных

Таблица 1: 3-куча

Вершины	Ребра	Диапазон мощности	Время
10	10	[1, 10]	0.000366
10	100	[1, 10]	0.001599
100	1000	[1, 10]	0.135762
1000	500	[1, 10]	1.397146
1000	500	[1, 10]	0.274
1000	1000	[1, 10]	2.815561
1000	5000	[1, 10]	11.588598
1000	10000	[1,10]	19.61
100	50000	[1, 10]	6.868
1000	25000	[1,10]	51.83
1000	50000	[1,10]	99.407265
1000	75000	[1,10]	99.407265
1000	100000	[1,10]	208.88

Таблица 2: 15-куча

Вершины	Ребра	Диапазон мощности	Время
10	10	[1, 10]	0.000455
10	100	[1, 10]	0.001662
100	1000	[1, 10]	0.139681
1000	500	[1, 10]	1.420670
1000	1000	[1, 10]	2.851219
1000	5000	[1,10]	11.161763
1000	10000	[1,10]	19.19
100	50000	[1, 10]	6.773
1000	25000	[1,10]	50.68
1000	50000	[1, 10]	99.397131
1000	75000	[1,10]	99.407265
1000	100000	[1,10]	204.41

5.4 Конфигурация компьютера

```
[mikhail@archlinux-t480 ~]$ neofetch
mikhail@archlinux-t480
----------------------
OS: Arch Linux x86_64
Host: 20L6S3U201 ThinkPad T480
Kernel: 6.5.8-arch1-1
Uptime: 13 mins
Packages: 790 (pacman), 32 (flatpak)
Shell: bash 5.1.16
Resolution: 1920x1080
DE: GNOME 45.0
WM: Mutter
WM Theme: Adwaita
Theme: Adwaita [GTK2/3]
Icons: Adwaita [GTK2/3]
Terminal: kgx
CPU: Intel i5-8350U (8) @ 3.600GHz
GPU: Intel UHD Graphics 620
Memory: 3639MiB / 15876MiB

3-куча 1000,75000:
Execution Time: 141.861822 seconds
15-куча 1000,75000:
Execution Time: 140.770057 seconds
15-куча 1000,75000:
Execution Time: 142.955417 seconds
3-куча 1000,75000:
Execution Time: 144.291224 seconds
```

6 График зависимости входных данных от времени

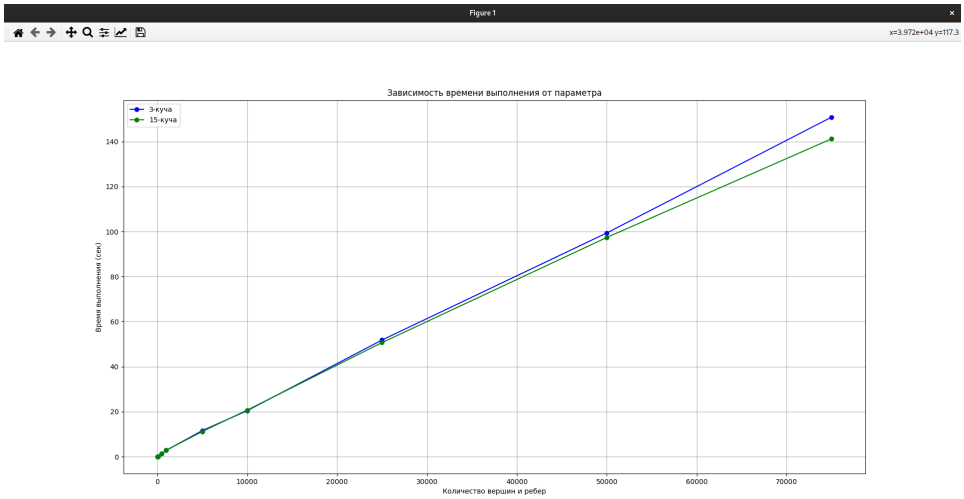

Рис. 3: График зависимости входных данных от веремени 3-кучи и 15-кучи

7 Вывод

В ходе выполнения лабораторной работы удалось досич следующих целей

- Разобраться в понятих 3-куча, 15-куча, понять их различия.
- Реализовать классы куч и класса Граф.
- Проверить корректную работу алгоритма Дейкстры в нескольких тестах.
- Исследовать отличия алгоритма Дейкстры на 3-куче и 15-куче.

Так как алгоритм Дейкстры решает немалую долю задач в жизни человека, то исходя из тестов, найдутся такие задачи в которых алгоритм на 15-куче будет быстрее находить кратчайшие пути, а значит будет использоваться в решение задач.

8 Приложение

8.1 main.py

```
from tests.test3_15heap import *
from tests.dijkstraTest import *
from tests.showTime import *


if __name__ == '__main__':

# first_test()
# second_test()
# third_test()
# fourth_test()
# fiveth_test()
# sixth_test()
# seventh_test()
# eight_test()
# nineth_test()
# tenth_test()
# eleventh_test()
# twelve_test()
```

```
# test_rand()
# test_poln(100)
showTime()
```

8.2 graph3.py

```
import random
import networkx as nx
import matplotlib
import matplotlib.pyplot as plt
import timeit
matplotlib.use('Qt5Agg')

class TernaryHeap:
def __init__(self):
self.heap = []

def push(self, value):
self.heap.append(value)
self._sift_up(len(self.heap) - 1)

def pop(self):
if len(self.heap) == 1:
return self.heap.pop()
value = self.heap[0]
self.heap[0] = self.heap.pop()
self._sift_down(0)
return value

def _sift_up(self, index):
parent = (index - 1) // 3
if parent >= 0 and self.heap[index][0] < self.heap[parent][0]:
self.heap[index], self.heap[parent] = self.heap[parent], self.heap[index]
self._sift_up(parent)

def _sift_down(self, index):
child1 = 3 * index + 1
child2 = 3 * index + 2
child3 = 3 * index + 3
```

```
smallest = index
if child1 < len(self.heap) and self.heap[child1][0] < self.heap[smallest][0]:
smallest = child1
if child2 < len(self.heap) and self.heap[child2][0] < self.heap[smallest][0]:
smallest = child2
if child3 < len(self.heap) and self.heap[child3][0] < self.heap[smallest][0]:
smallest = child3
if smallest != index:
self.heap[index], self.heap[smallest] = self.heap[smallest], self.heap[index]
self._sift_down(smallest)


class Graph3:
def __init__(self, num_vertices=None):
self.nodes = set()
self.edges = {}
self.distances = {}
self.num_vertices = num_vertices

def add_node(self, value):
self.nodes.add(value)
self.edges[value] = []

def add_edge(self, from_node, to_node, distance):
self.edges[from_node].append(to_node)
self.distances[(from_node, to_node)] = distance

def dijkstra(self, initial_node):
visited = {initial_node: 0}
heap = TernaryHeap()
heap.push((0, initial_node))
while heap.heap:
(current_weight, min_node) = heap.pop()
if current_weight > visited[min_node]:
continue
for edge in self.edges[min_node]:
weight = current_weight + self.distances[(min_node, edge)]
if edge not in visited or weight < visited[edge]:
visited[edge] = weight
heap.push((weight, edge))
```

```
return visited

def generate_random_graph(self, num_nodes, num_edges):
for i in range(num_nodes):
self.add_node(str(i))

available_nodes = [str(i) for i in range(num_nodes)]

for _ in range(num_edges):
if available_nodes:
index1 = random.randint(0, len(available_nodes) - 1)
node1 = available_nodes[index1]

if available_nodes:
index2 = random.randint(0, len(available_nodes) - 1)
node2 = available_nodes[index2]

self.add_edge(node1, node2, random.randint(1, 10))




def generate_complete_graph(self, num_nodes):
for i in range(num_nodes):
self.add_node(str(i))
for i in range(num_nodes):
for j in range(i + 1, num_nodes):
distance = random.randint(1, 10)
self.add_edge(str(i), str(j), distance)

def visualize_graph(self):
G = nx.Graph()
for node in self.nodes:
G.add_node(node)
for from_node in self.edges:
for to_node in self.edges[from_node]:
G.add_edge(from_node, to_node, weight=self.distances[(from_node, to_node)])
pos = nx.spring_layout(G)
```

```
nx.draw_networkx_nodes(G, pos)
nx.draw_networkx_edges(G, pos)
nx.draw_networkx_labels(G, pos)
nx.draw_networkx_edge_labels(G, pos, edge_labels=nx.get_edge_attributes(G, 'weight'))
plt.axis('off')
plt.show()
```

8.3 graph15.py

```
import random
import networkx as nx
import matplotlib.pyplot as plt
import matplotlib
matplotlib.use('Qt5Agg')


class HexadecimalHeap:
def __init__(self):
self.heap = []

def push(self, value):
self.heap.append(value)
self._sift_up(len(self.heap) - 1)

def pop(self):
if len(self.heap) == 1:
return self.heap.pop()
value = self.heap[0]
self.heap[0] = self.heap.pop()
self._sift_down(0)
return value

def _sift_up(self, index):
parent = (index - 1) // 15
if parent >= 0 and self.heap[index][0] < self.heap[parent][0]:
self.heap[index], self.heap[parent] = self.heap[parent], self.heap[index]
self._sift_up(parent)

def _sift_down(self, index):
```

```
smallest = index
for i in range(1, 16):
child = 15 * index + i
if child < len(self.heap) and self.heap[child][0] < self.heap[smallest][0]:
smallest = child
if smallest != index:
self.heap[index], self.heap[smallest] = self.heap[smallest], self.heap[index]
self._sift_down(smallest)


class Graph15:
def __init__(self, num_vertices=None):
self.nodes = set()
self.edges = {}
self.distances = {}
self.num_vertices = num_vertices

def add_node(self, value):
self.nodes.add(value)
self.edges[value] = []

def add_edge(self, from_node, to_node, distance):
self.edges[from_node].append(to_node)
self.distances[(from_node, to_node)] = distance

def dijkstra(self, initial_node):
if initial_node not in self.nodes:
return f"Узел {initial_node} отсутствует в графе"
if initial_node not in self.edges:
return f"Нет ребер, исходящих из узла {initial_node}"
visited = {initial_node: 0}
heap = HexadecimalHeap()
heap.push((0, initial_node))
while heap.heap:
current_weight, min_node = heap.pop()
if current_weight > visited[min_node]:
continue
for edge in self.edges[min_node]:
weight = current_weight + self.distances[(min_node, edge)]
if edge not in visited or weight < visited[edge]:
```

```
visited[edge] = weight
heap.push((weight, edge))
return visited



def generate_random_graph(self, num_nodes, num_edges):
for i in range(num_nodes):
self.add_node(str(i))

available_nodes = [str(i) for i in range(num_nodes)]

for _ in range(num_edges):
if available_nodes:
index1 = random.randint(0, len(available_nodes) - 1)
node1 = available_nodes[index1]

if available_nodes:
index2 = random.randint(0, len(available_nodes) - 1)
node2 = available_nodes[index2]

self.add_edge(node1, node2, random.randint(1, 10))



def generate_complete_graph(self, num_nodes):
for i in range(num_nodes):
self.add_node(str(i))
for i in range(num_nodes):
for j in range(i + 1, num_nodes):
distance = random.randint(1, 10)
self.add_edge(str(i), str(j), distance)

def visualize_graph(self):
G = nx.Graph()
for node in self.nodes:
G.add_node(node)
for from_node in self.edges:
for to_node in self.edges[from_node]:
```

```
G.add_edge(from_node, to_node, weight=self.distances[(from_node, to_node)])
pos = nx.spring_layout(G)
nx.draw_networkx_nodes(G, pos)
nx.draw_networkx_edges(G, pos)
nx.draw_networkx_labels(G, pos)
nx.draw_networkx_edge_labels(G, pos, edge_labels=nx.get_edge_attributes(G, 'weight'))
plt.axis('off')
plt.show()
```

8.4 dijkstraTest.py

```
from graph3 import Graph3
from graph15 import Graph15

def first_test():
g3 = Graph3(8)
g3.add_node('A')
g3.add_node('B')
g3.add_node('C')
g3.add_node('D')
g3.add_node('E')
g3.add_node('F')
g3.add_node('G')
g3.add_node('H')
g3.add_edge('A', 'B', 8)
g3.add_edge('A', 'C', 2)
g3.add_edge('A', 'D', 7)
g3.add_edge('B', 'E', 2)
g3.add_edge('B', 'F', 2)
g3.add_edge('C', 'F', 6)
g3.add_edge('C', 'G', 7)
g3.add_edge('D', 'E', 2)
g3.add_edge('D', 'F', 4)
g3.add_edge('E', 'H', 4)
g3.add_edge('F', 'H', 9)
g3.add_edge('G', 'H', 6)
shortest_paths = g3.dijkstra('A')
print(shortest_paths)
```

```
g15 = Graph15(8)
g15.add_node('A')
g15.add_node('B')
g15.add_node('C')
g15.add_node('D')
g15.add_node('E')
g15.add_node('F')
g15.add_node('G')
g15.add_node('H')
g15.add_edge('A', 'B', 8)
g15.add_edge('A', 'C', 2)
g15.add_edge('A', 'D', 7)
g15.add_edge('B', 'E', 2)
g15.add_edge('B', 'F', 2)
g15.add_edge('C', 'F', 6)
g15.add_edge('C', 'G', 7)
g15.add_edge('D', 'E', 2)
g15.add_edge('D', 'F', 4)
g15.add_edge('E', 'H', 4)
g15.add_edge('F', 'H', 9)
g15.add_edge('G', 'H', 6)
shortest_paths = g15.dijkstra('A')
print(shortest_paths)

def second_test():
g3 = Graph3(26)
g15 = Graph15(26)
g3.add_node('A')
g15.add_node('A')

for i in range(1, 26):
g3.add_node(chr(ord('A') + i))
#print(chr(ord('A') + i))

for i in range(1, 26):
g15.add_node(chr(ord('A') + i))

g3.add_node('AA')
g15.add_node('AA')
g3.add_node('AB')
```

```
g15.add_node('AC')
g3.add_node('AD')
g15.add_node('AD')
g3.add_node('AE')
g15.add_node('AE')
g3.add_node('AF')
g15.add_node('AF')
g3.add_node('AG')
g15.add_node('AG')
g3.add_node('AH')
g15.add_node('AH')
g3.add_node('AI')
g15.add_node('AI')
g3.add_node('AJ')
g15.add_node('AJ')
g3.add_node('AK')
g15.add_node('AK')
g3.add_node('AL')
g15.add_node('AL')
g3.add_node('AM')
g15.add_node('AM')
g3.add_edge('A', 'B', 3)
g3.add_edge('A', 'C', 4)
g3.add_edge('A', 'D', 3)
g3.add_edge('A', 'E', 6)
g3.add_edge('B', 'F', 5)
g3.add_edge('B', 'G', 5)
g3.add_edge('C', 'G', 1)
g3.add_edge('C', 'H', 5)
g3.add_edge('D', 'K', 9)
g3.add_edge('D', 'I', 3)
g3.add_edge('E', 'L', 7)
g3.add_edge('F', 'J', 2)
g3.add_edge('F', 'K', 1)
g3.add_edge('G', 'K', 3)
g3.add_edge('H', 'L', 5)
g3.add_edge('I', 'O', 4)
g3.add_edge('I', 'M', 3)
g3.add_edge('J', 'N', 7)
g3.add_edge('K', 'N', 6)
```

```
g3.add_edge('K', 'O', 3)
g3.add_edge('L', 'P', 4)
g3.add_edge('M', 'Q', 4)
g3.add_edge('N', 'R', 2)
g3.add_edge('O', 'S', 8)
g3.add_edge('P', 'T', 5)
g3.add_edge('Q', 'T', 5)
g3.add_edge('Q', 'U', 5)
g3.add_edge('R', 'V', 3)
g3.add_edge('S', 'V', 3)
g3.add_edge('T', 'V', 4)
g3.add_edge('U', 'V', 2)
g3.add_edge('U', 'X', 6)
g3.add_edge('V', 'W', 4)
g3.add_edge('V', 'Y', 3)
g3.add_edge('V', 'X', 4)
g3.add_edge('W', 'Z', 4)
g3.add_edge('Y', 'Z', 6)
g3.add_edge('X', 'Z', 4)

g15.add_edge('A', 'B', 3)
g15.add_edge('A', 'C', 4)
g15.add_edge('A', 'D', 3)
g15.add_edge('A', 'E', 6)
g15.add_edge('B', 'F', 5)
g15.add_edge('B', 'G', 5)
g15.add_edge('C', 'G', 1)
g15.add_edge('C', 'H', 5)
g15.add_edge('D', 'K', 9)
g15.add_edge('D', 'I', 3)
g15.add_edge('E', 'L', 7)
g15.add_edge('F', 'J', 2)
g15.add_edge('F', 'K', 1)
g15.add_edge('G', 'K', 3)
g15.add_edge('H', 'L', 5)
g15.add_edge('I', 'O', 4)
g15.add_edge('I', 'M', 3)
g15.add_edge('J', 'N', 7)
g15.add_edge('K', 'N', 6)
g15.add_edge('K', 'O', 3)
```

```
g15.add_edge('L', 'P', 4)
g15.add_edge('M', 'Q', 4)
g15.add_edge('N', 'R', 2)
g15.add_edge('O', 'S', 8)
g15.add_edge('P', 'T', 5)
g15.add_edge('Q', 'T', 5)
g15.add_edge('Q', 'U', 5)
g15.add_edge('R', 'V', 3)
g15.add_edge('S', 'V', 3)
g15.add_edge('T', 'V', 4)
g15.add_edge('U', 'V', 2)
g15.add_edge('U', 'X', 6)
g15.add_edge('V', 'W', 4)
g15.add_edge('V', 'Y', 3)
g15.add_edge('V', 'X', 4)
g15.add_edge('W', 'Z', 4)
g15.add_edge('Y', 'Z', 6)
g15.add_edge('X', 'Z', 4)
shortest_paths = g3.dijkstra('A')
print(shortest_paths)
```

8.5 showTime.py

```
import matplotlib.pyplot as plt
import matplotlib
matplotlib.use('Qt5Agg')



def showTime():
x1 = [10, 100, 500, 1000, 5000, 25000, 50000]
y1 = [0.0003, 0.0015, 1.39, 2.81, 11.58, 51.83, 99.40]

x2 = [10, 100, 500, 1000, 5000, 25000, 50000]
y2 = [0.0004, 0.0016, 1.42,  2.85,  11.16, 50.68, 99.39]

plt.plot(x1, y1, marker='o', linestyle='-', color='b', label='3-куча')
plt.plot(x2, y2, marker='o', linestyle='-', color='g', label='15-куча')
plt.title('Зависимость времени выполнения от параметра')
plt.xlabel('Количество вершин и ребер')
```

```
plt.ylabel('Время выполнения (сек)')
plt.legend()
plt.grid(True)
plt.show()
```

8.6 test3.15heap.py

```
from graph3 import Graph3
from graph15 import Graph15
import random
import timeit
import time
import matplotlib.pyplot as plt
import matplotlib

def run_dijkstra3(num_vertices, num_edges):
graph = Graph3(num_vertices)

# Добавление вершин
for vertex in range(num_vertices):
graph.add_node(vertex)

# Добавление ребер
for vertex in range(num_vertices):
for _ in range(num_edges):
to_node = random.randint(0, num_vertices-1)
distance = random.randint(1, 10)
graph.add_edge(vertex, to_node, distance)

# Замер времени выполнения
def run_algorithm():
initial_node = 0
return graph.dijkstra(initial_node)

execution_time = timeit.timeit(run_algorithm, number=10)
print(f"3-куча {num_vertices},{num_edges}:\n    Execution Time: {execution_time:.6f} se

def run_dijkstra15(num_vertices, num_edges):
graph = Graph15(num_vertices)
```

```
# Добавление вершин
for vertex in range(num_vertices):
graph.add_node(vertex)

# Добавление ребер
for vertex in range(num_vertices):
for _ in range(num_edges):
to_node = random.randint(0, num_vertices-1)
distance = random.randint(1, 10)
graph.add_edge(vertex, to_node, distance)

# Замер времени выполнения
def run_algorithm():
initial_node = 0
return graph.dijkstra(initial_node)

execution_time = timeit.timeit(run_algorithm, number=10)
print(f"15-куча {num_vertices},{num_edges}:\n   Execution Time: {execution_time:.6f} se

def third_test():
run_dijkstra3(10, 10)
run_dijkstra15(10, 10)

def fourth_test():
run_dijkstra3(10, 100)
run_dijkstra15(10, 100)

def fiveth_test():
run_dijkstra3(100, 1000)
run_dijkstra15(100, 1000)

def sixth_test():
run_dijkstra3(1000, 500)
run_dijkstra15(1000, 500)

def seventh_test():
run_dijkstra3(1000, 1000)
run_dijkstra15(1000, 1000)
```

```
def eight_test():
run_dijkstra3(1000, 5000)
run_dijkstra15(1000, 5000)

def nineth_test():
run_dijkstra3(1000, 50000)
run_dijkstra15(1000, 50000)

def tenth_test():
run_dijkstra3(100, 50000)
run_dijkstra15(100, 50000)

def eleventh_test():
run_dijkstra3(1000, 25000)
run_dijkstra15(1000, 25000)

def twelve_test():
run_dijkstra3(10000, 500000)
run_dijkstra15(10000, 500000)

def test_rand():
g3 = Graph3()
g15 = Graph15()
g3.generate_random_graph(100, 100000)
g15.generate_random_graph(100, 10000)
t1 = timeit.timeit(lambda: g3.dijkstra('0'), number=10)
t2 = timeit.timeit(lambda: g15.dijkstra('0'), number=10)
print(f"Случайный граф 100 вершин, 1000 ребер\nВремя выполнения алгоритма Дейкстры на т
print(f"Время выполнения алгоритма Дейкстры на пятнадцатеричной куче: {t2:6f} сек.")
#print(g.dijkstra('0'))
g3.visualize_graph()
g15.visualize_graph()
def test_poln(vertices):
g3 = Graph3()
g15 = Graph15()
g3.generate_complete_graph(vertices)
g15.generate_complete_graph(vertices)
t1 = timeit.timeit(lambda: g3.dijkstra('0'), number=10)
t2 = timeit.timeit(lambda: g15.dijkstra('0'), number=10)
print(f"Полный граф\nВремя выполнения алгоритма Дейкстры на троичной куче: {t1:6f} сек
```

```
print(f"Время выполнения алгоритма Дейкстры на пятнадцатеричной куче: {t2:6f} сек.")
#print(g.dijkstra('0'))
```