**Санкт-Петербургский государственный университет**

**Факультет прикладной математики и процессов управления**

**Лабораторная работа №6**

**Решение задачи о коммивояжере с помощью муравьиного алгоритма**

|  |  |  |
| --- | --- | --- |
| Студентка гр. Б15-ПУ |  | Гладкая М.В. |
| Преподаватель |  | Дик А.Г. |

Санкт-Петербург

2023 г.

**Оглавление**

1. **Цель работы ………………………………………………………………3**
2. **Задача………………………………………………………………………3**
3. **Теоритическая часть……………………………………………………..3**
4. **Описание программы ……………………………………………………4**
   1. **Список используемых библиотек……...………………………...…4**
   2. **Описание функций и классов..…………...…………………………4**
   3. **Общий ход программы………………………………………………6**
5. **Рекомендации программиста……………………………………………7**
6. **Контрольный пример…………………………………………………….7**
7. **Результаты работы алгоритма на заданном графе……..…………....9**
8. **Сравнение алгоритма имитации отжига, алгоритма поиска ближайшего соседа и муравьиного алгоритма………………………11**
9. **Вывод…………………………………………..…………………..…..….12**

**Приложения…………..…………………………………………………………13**

1. **Цель работы**

Исследование муравьиного алгоритма оптимизации для решения задачи о коммивояжере на примере графов.

1. **Задачи**

* Изучить муравьиный алгоритм
* Написать программу поиска кратчайшего гамильтонова цикла в графе с помощью муравьиного алгоритма
* Протестировать программу
* Реализовать GUI
* Найти кратчайший гамильтонов цикл для заданного графа

1. **Теоритическая часть**

Муравьиные алгоритмы основаны на использовании популяции потенциальных решений и разработаны для решения задач комбинаторной оптимизации, прежде всего, поиска различных путей на графах. Кооперация между особями (искусственными муравьями) здесь реализуется на основе моделирования стигметрии. При этом каждый агент, называемый искусственным муравьем, ищет решение поставленной задачи. Искусственные муравьи последовательно строят решение задачи, передвигаясь по графу из одной вершины в другую, откладывают феромон и при выборе дальнейшего участка пути учитывают концентрацию этого фермента. Чем больше концентрация феромона в последующем участке, тем больше вероятность его выбора.

Вероятность перехода из одной вершины в другую рассчитывается по формуле:

, где

α, β – коэффициенты (α, β ϵ (0, 1)),

, ,

*k –* вершина, взятая из списка всех вершин.

Значения феромона после прохождения полного маршрута обновляются по следующей формуле на всем маршруте:

, где

, Q – константа, ,

*p* – коэффициент высыхания феромона.

Муравьиный алгоритм в исходной формулировке заключается в нахождении замкнутой кривой минимальной длины, соединяющей заданный набор точек на плоскости. В нашем случае набором точек на плоскости будет являться взвешенный ориентированный граф.

1. **Описание программы**

Программный код написан на языке Python.С кодом можно ознакомиться по ссылке <https://github.com/9Neechan/antAlgorithm.git> .

* 1. **Список используемых библиотек**

Список используемых библиотек представлен в Таблице 1.

*Таблица 1. Список используемых библиотек*

|  |  |
| --- | --- |
| **Название** | **Описание** |
| PySimpleGUI | Реализация GUI |
| networkx | Работа с графами, как со структурой |
| matplotlib | Отрисовка графиков |
| random | Генерация рандомных значений |
| math | Использование математических функций |

* 1. **Описание функций и классов**

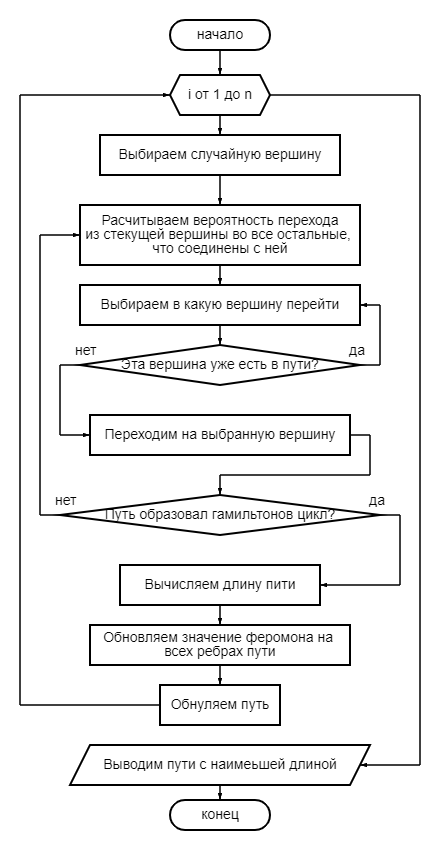
В таблице 2 представлено описание функций файла algorithm.py.

*Таблица 2. Функции algorithm.py*

|  |  |  |
| --- | --- | --- |
| **Имя функции** | **Входные данные** | **Описание функции** |
| create\_multigraph\_struct() | graph – матрица смежности графа | Создает экземпляр класса мультиграфа из библиотеки networkx на основе переданной матрицы смежности графа |
| create\_graph\_struct() | path\_arr - путь  graph – матрица смежности графа | Создает экземпляр класса орграфа из библиотеки networkx на основе переданного пути |
| make\_plt() | flag – флаг, казывающий на то, какой вид графа создавать,  path\_arr - путь,  graph – матрица смежности графа,  i – индекс пути в списке путей | Отрисовывает граф по переданным параметрам с помощью matplotlib и сохраняет в папку pictires |
| draw\_graph() | im\_arr – массив, в который будут сохраняться изображения графов  paths\_arr – массив путей обхода  graph – матрица смежности графа | Отрисовывает изображения результатов работы алгоритма, которые будут использоваться в GUI |
| check\_edges() | path\_arr - путь,  graph – матрица смежности графа, | Проверяет является ли путь гамильтоновым циклом в графе |
| count\_sum\_v() | v - вершина  graph – матрица смежности графа  feromon – матрица феромонов a, b – коэффициенты при расчете нормировки | Считает коэффициент нормировки суммы длин ребер, выходящих из заданной вершины |
| search\_v() | path - путь | Выясняет какой вершины нет в пути |
| calculate\_path() | graph – матрица смежности графа  feromon – матрица феромонов a, b – коэффициенты при расчете вероятности перехода в ту или иную вершину  v – первая вершина пути | Генерирует путь по всем правилам муравьиного алгоритма |
| ant\_algorithm() | graph – матрица смежности графа  a, b, p – коэффициенты при расчете вероятности перехода в ту или иную вершину  iters – количество итераций алгоритма | Реализует работу муравьиного алгоритма на ориентированном взвешенном графе |

* 1. **Общий ход программы**

На рисунке 1 изображен ход подпрограммы, реализующей работу метода имитации отжига.



*Рисунок 1. Блок-схема подпрограммы, реализующей работу метода имитации отжига*

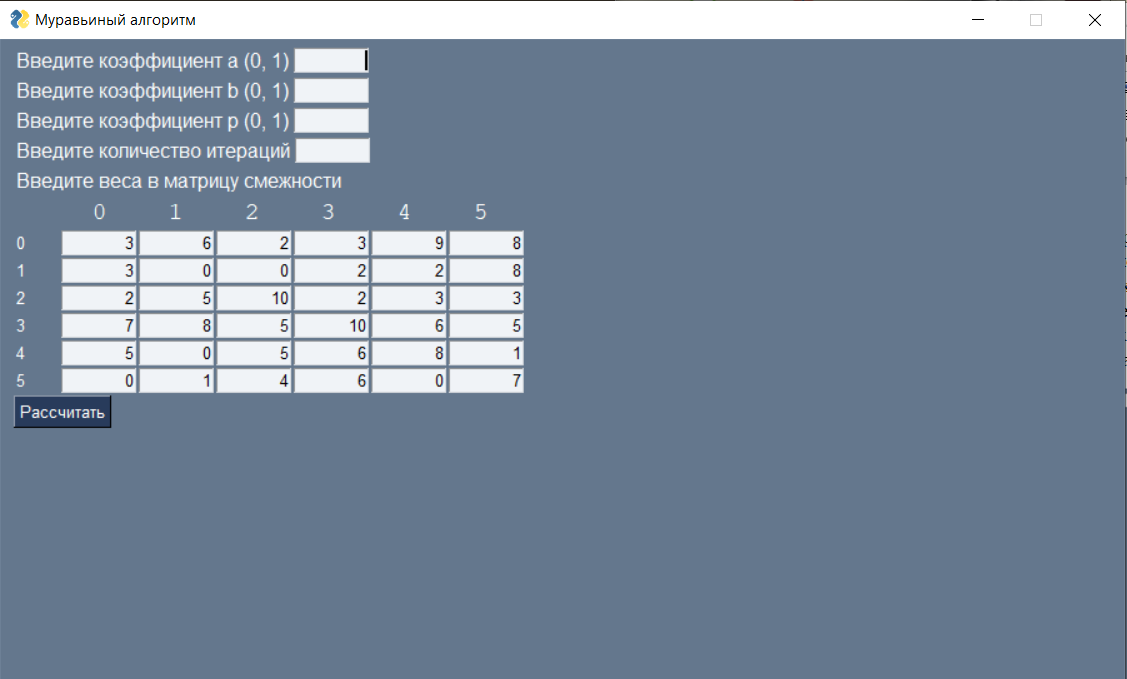
1. **Рекомендации программиста**

Для запуска программы необходима 64-битная операционная система Windows и python не ниже 3.9. Для работы с кодом необходима IDE PyCharm или другая любая среда разработки для python.

Импортируйте библиотеки из пункта 4.1.

1. **Контрольный пример**

Запустите программу. Перед вами откроется начальное окно программы (Рис. 2). Введите коэффициенты *a, b, p*, веса ребер в матрицу смежности графа и количество итераций алгоритма, для которого хотите произвести расчеты или оставьте автоматически сгенерированные значения. Затем нажмите кнопку «Рассчитать» для расчета наикратчайших гамильтоновых циклов, содержащихся в графе с помощью муравьиного алгоритма.

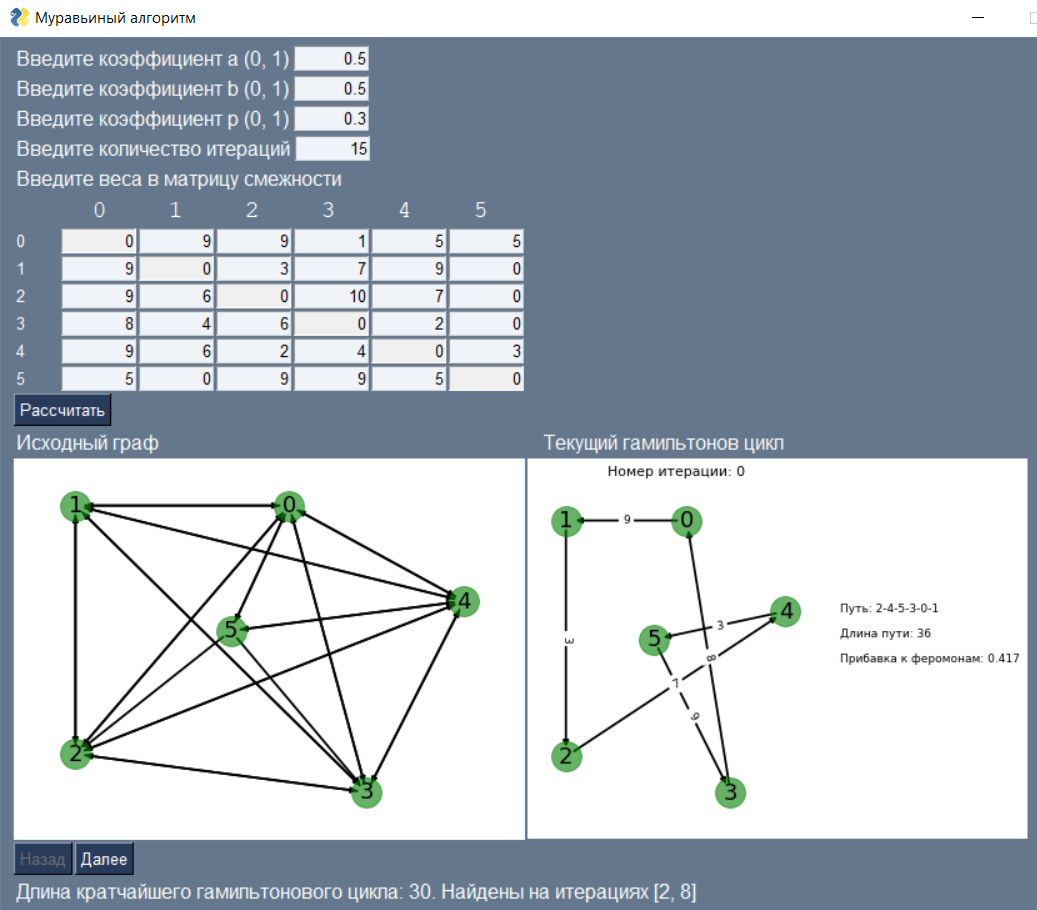


*Рисунок 2. Начальное окно*

По окончании расчетов будут выведены изображения графа, построенного на основе матрицы смежности и графы гамильтоновых циклов на каждой итерации работы алгоритма.

Послеизображений графов будет написана длина кратчайшего гамильтонового цикла, который был найден в ходе алгоритма и номера итераций, на которых он был найден.

Чтобы пролистывать между изображениями графов нажимайте на кнопки «Далее» и «Назад». На рисунке 3 изображен один из результатов работы программы.

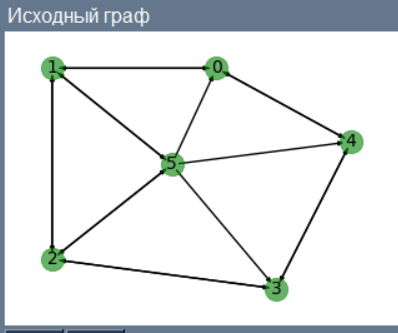


*Рисунок 3. Окно программы после нажатия кнопки «Рассчитать»*

Чтобы сделать расчеты для другого графа, введите новые значения в таблицу смежности и нажмите кнопку «Рассчитать».

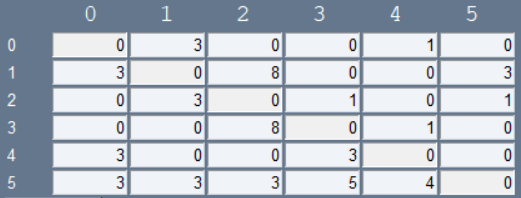
1. **Результаты работы алгоритма на заданном графе**

Расчеты были сделаны для графа следующего вида (рисунок 4).



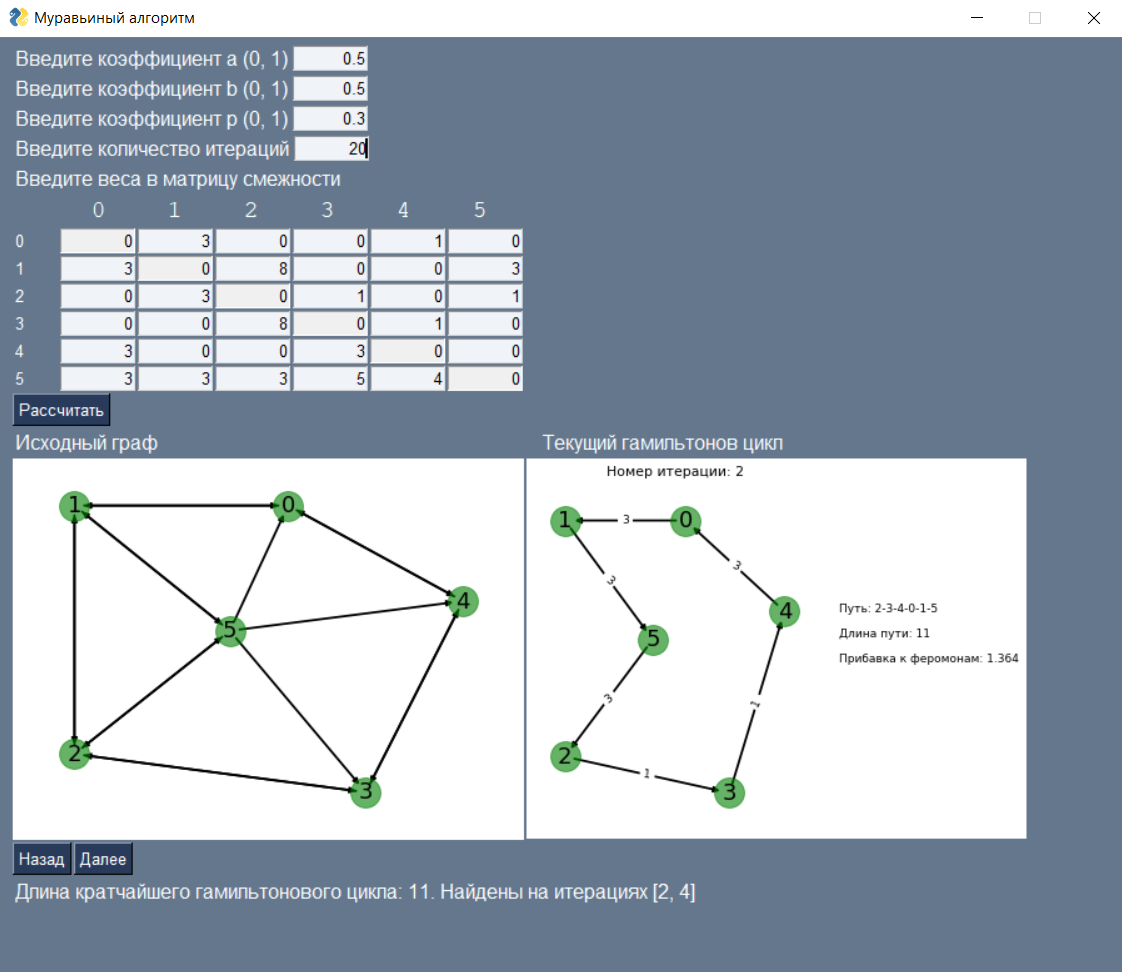
*Рисунок 4. Заданный орграф*

Матрица смежности имеет следующий вид (рисунок 5).



*Рисунок 5. Матрица смежности*

По результатам работы алгоритма длина кратчайшего гамильтонового цикла равна 11 (рисунок 6). Порядок вершин обходя для достижения такой длины следующий: 2-3-4-0-1-5.



*Рисунок 6. Результат работы программы*

Муравьиный алгоритм работает хорошо для очень маленьких графов и в большинстве случаев находит кратчайший путь, для больших графов характеристики ухудшаются, алгоритм становится менее стабильным и более чувствительным к выбору параметров. Сходимость адгоритма к кратчайшему пути хорошая при малом числе муравьев, в то время как большое количество муравьев часто ведет к тому, что процесс поиска не сходится.

Эффект испарения более важен для сложных графов. В этом случае при *p = 0* (нет испарения) алгоритм часто не сходится. С другой стороны, если феромон испаряется слишком быстро (большие значения *p*), алгоритм часто сходится к субоптимальным решениям.

Коэффициент α влияет на степень учета феромона, а коэффициент β влияет на степень учета длины ребра при выборе следующей вершины. При α = 0, муравей учитывает только длину ребра, а при β = 0 муравей учитывает только значение феромона.

Поскольку алгоритм по условиям задачи работает на неполном ориентированным графе, то зачастую важно, если отсутствует много ребер, с какой вершины происходит запуск алгоритма. Таким образом, была введена следующая модификация: при каждой новой итерации генерируется случайная начальная вершина.

1. **Сравнение алгоритма имитации отжига, алгоритма поиска ближайшего соседа и муравьиного алгоритма**

Алгоритм поиска ближайшего соседа относится к жадным алгоритмам, что означает большее время работы программы. Так как алгоритм перебирает все вершины, то он зачастую находит одинаковые циклы, которые были получены при начале расчета от разных вершин. Падает качество ответа при увеличении количества вершин.

Алгоритм имитации отжига находит более оптимальные пути, чем алгоритм поиска ближайшего соседа. Это видно по результатам их работы: алгоритм отжига находил минимальный цикл со стоимостью 13, а алгоритм поиска ближайшего соседа - со стоимостью 14.

Муравьиный алгоритм нашел самый короткий гамильтонов цикл из всех, что были найдены тремя алгоритмами – стоимость 11.

Если в муравьином алгоритме коэффициент α приравнять к 0, то, по сути, он превратится в алгоритм поиска ближайшего соседа.

Недостаток алгоритма имитации отжига по сравнению с муравьиным заключается в том, что с понижением температуры у него повышается вероятность “застрять” в худшем решении, чем могло бы быть, а понижение температуры зависит от случая. Муравьиный алгоритм в этом понимании более гибкий, потому что с каждой новой итерацией значения феромона на ребрах с маленькими весами увеличивается. Чем меньше вес ребра, тем больше будет на нем феромона с течением работы алгоритма.

Таким образом, алгоритм поиска ближайшего соседа – худший из алгоритмов, поскольку он не дает алгоритму шанса пойти по наиболее длинному ребру для, возможно, нахождения лучшего результата. Алгоритм имитации отжига сильнее зависит от случайных величин, чем муравьиный алгоритм.

Подводя итоги, можно отметить, что муравьиный алгоритм является самым точным и быстрым из всех трех, особенно на маленьких графах.

1. **Вывод**

В ходе данной работы был изучен метод имитации отжига, написана программа поиска кратчайшего гамильтонова цикла в орграфе с помощью метода имитации отжига, реализован GUI, найдены кратчайшие гамильтоновы циклы для заданного графа.

**Приложения**

Приложение А. algorithm.py

1. **import** matplotlib.pyplot as plt
2. **import** networkx as nx
3. **import** random
4. **import** math

7. **def** create\_multigraph\_struct(graph):
8. G **=** nx.MultiDiGraph()
9. **for** i **in** range(len(graph)):
10. **for** j **in** range(len(graph[0])):
11. **if** graph[i][j] !**=** 0 **and** i !**=** j:
12. G.add\_edge(f'{i}', f'{j}', weight**=**graph[i][j])
13. **return** G

16. **def** create\_graph\_struct(path\_arr, graph):
17. G **=** nx.DiGraph()
18. data **=** []
19. **for** j **in** range(len(path\_arr)):
20. **if** j **+** 1 < len(path\_arr):
21. data.append(path\_arr[j **+** 1])
22. data.append(path\_arr[0])
23. **for** i **in** range(len(path\_arr)):
24. G.add\_edge(f'{path\_arr[i]}', f'{data[i]}', weight**=**graph[path\_arr[i]][data[i]])
25. **return** G

28. **def** make\_plt(flag, path\_arr, graph, i, path\_len, tao):
29. """Рисует графы с помощью бибилиотеки matplotlib и сохраняет в папку pictures"""
30. pos **=** {'0': [0, 0.25],
31. '1': [**-**0.55, 0.25],
32. '2': [**-**0.55, **-**0.4],
33. '3': [0.2, **-**0.5],
34. '4': [0.45, 0],
35. '5': [**-**0.15, **-**0.0775903]}
36. path **=** ''
38. **if** flag **==** 'multi':
39. G **=** create\_multigraph\_struct(graph)
40. snapshot\_name **=** "pictures/initial.png"
41. **else**:
42. G **=** create\_graph\_struct(path\_arr, graph)
43. snapshot\_name **=** f"pictures/{i}.png"
44. path **=** f'{str(path\_arr[0])}-{str(path\_arr[1])}-' \
45. f'{str(path\_arr[2])}-{str(path\_arr[3])}-' \
46. f'{str(path\_arr[4])}-{str(path\_arr[5])}'
48. # nodes
49. nx.draw\_networkx\_nodes(G, pos, node\_size**=**700, node\_color**=**'green', alpha**=**0.6)
50. # edges
51. nx.draw\_networkx\_edges(G, pos, width**=**2)
52. # node labels
53. nx.draw\_networkx\_labels(G, pos, font\_size**=**20, font\_family**=**"sans-serif")
54. **if** flag **==** "basic":
55. # edge weight labels
56. edge\_labels **=** nx.get\_edge\_attributes(G, "weight")
57. nx.draw\_networkx\_edge\_labels(G, pos, edge\_labels)
58. plt.title(f'Номер итерации: {i}')
59. plt.text(0.7, 0, f'Путь: {path}')
60. plt.text(0.7, **-**0.07, f'Длина пути: {path\_len}')
61. plt.text(0.7, **-**0.14, f'Прибавка к феромонам: {tao}')
62. ax **=** plt.gca()
63. ax.margins(0.08)
64. plt.axis("off")
65. plt.tight\_layout()
66. plt.savefig(snapshot\_name, dpi**=**65, bbox\_inches**=**'tight')
67. plt.close()

70. **def** draw\_graph(data, best\_paths, graph):
71. """Визуализация графов"""
72. flag **=** 'multi'
73. make\_plt(flag, [], graph, 0, 0, 0)
75. # рисуем стадии работы цикла
76. **for** i **in** range(len(data)):
77. flag **=** 'basic'
78. make\_plt(flag, data[i][1][1], graph, data[i][0], data[i][1][0], round(data[i][2], 3))

81. **def** check\_edges(path, graph):
82. """Проверяет является ли путь гамильтоновым циклом"""
83. leng **=** **-**1
84. **for** i **in** range(len(path)):
85. **if** i **==** len(path)**-**1:
86. **if** graph[path[i]][path[0]] **==** 0:
87. **return** False, leng
88. **else**:
89. leng **+=** graph[path[i]][path[0]]
90. **else**:
91. **if** graph[path[i]][path[i**+**1]] **==** 0:
92. **return** False, leng
93. **else**:
94. leng **+=** graph[path[i]][path[i**+**1]]
95. **return** True, leng

98. **def** count\_sum\_v(v, graph, feromon, a, b):
99. """Считает сумму длин ребер, выходящих из заданной вершины"""
100. summ **=** 0
101. **for** i **in** range(len(graph[v])):
102. **if** graph[v][i] !**=** 0:
103. Lvk **=** graph[v][i]
104. nvk **=** 1**/**Lvk
105. summ **+=** math.pow(nvk, b) **\*** math.pow(feromon[v][i], a)
106. **return** summ

109. **def** search\_v(path):
110. """Выясняет какой вершины нет в пути"""
111. list **=** [0, 1, 2, 3, 4, 5]
112. **for** el **in** path:
113. list.remove(el)
114. **return** list[0]

117. **def** calculate\_path(graph, feromon, a, b, v):
118. """Создает путь"""
119. # случайно выбираем вершину
120. #v = random.randint(0, len(graph)-1)
121. path **=** [0, [v]]
122. summ **=** count\_sum\_v(v, graph, feromon, a, b)
124. # шаг для одной i-той вершины
125. **for** i **in** range(len(graph) **-** 1):
126. # считаем долю вероятности пойти в конкретную вершину из 100%
127. sum\_p **=** 0
128. p\_arr **=** []
129. **for** i **in** range(len(graph[v])):
130. **if** graph[v][i] !**=** 0 **and** i **not** **in** path[1]:
131. Lij **=** graph[v][i]
132. nij **=** 1 **/** Lij
133. p **=** 100 **\*** (math.pow(nij, b) **\*** math.pow(feromon[v][i], a) **/** summ)
134. sum\_p **+=** p
135. p\_arr.append([sum\_p, i])
137. **if** len(p\_arr) **==** 1:
138. next\_v **=** p\_arr[0][1]
139. **elif** len(path[1]) **==** 5:
140. next\_v **=** search\_v(path[1])
141. **else**:
142. # выбираем следующую вершину
143. rand\_num **=** random.randint(0, 100)
144. next\_v **=** **-**1
145. **for** i **in** range(len(p\_arr)):
146. **if** i **+** 1 >**=** len(p\_arr):
147. next\_v **=** p\_arr[i][1]
148. **break**
149. **else**:
150. next\_v **=** sum([1, 2, 3, 4, 5]) **-** sum(path[1])
151. **if** rand\_num >**=** p\_arr[i][0]:
152. **if** rand\_num < p\_arr[i **+** 1][0]:
153. next\_v **=** p\_arr[i][1]
154. **break**
155. # добавляем ее в путь
156. path[0] **+=** graph[v][next\_v]
157. path[1].append(next\_v)
159. v **=** next\_v
160. summ **=** count\_sum\_v(v, graph, feromon, a, b)
162. **return** path

165. **def** ant\_algorithm(graph, a, b, p, iters):
166. data\_for\_pics **=** []
167. min\_l **=** 10000
169. # заполняем матрицу феромонов
170. feromon **=** [[1 **for** \_ **in** range(len(graph))] **for** arr **in** range(len(graph))]
171. **for** i **in** range(len(graph)):
172. **for** j **in** range(len(graph)):
173. **if** graph[i][j] **==** 0:
174. feromon[i][j] **=** 0
176. # итерации алгоритма
177. **for** j **in** range(iters):
178. # считаем путь
179. v **=** random.randint(0, len(graph) **-** 1)
180. path **=** calculate\_path(graph, feromon, a, b, v)
181. # проверяем является ли он гамильтоновым циклом
182. booll, leng **=** check\_edges(path[1], graph)
184. **if** booll:
185. # обновляем матрицу феромонов
186. tao **=** 15 **/** path[0]
187. **for** i **in** range(len(path[1])):
188. **if** i**+**1 >**=** len(path[1]):
189. feromon[path[1][i]][path[1][0]] **=** feromon[path[1][i]][path[1][0]] **\*** (1 **-** p) **+** tao
190. **else**:
191. feromon[path[1][i]][path[1][i**+**1]] **=** feromon[path[1][i]][path[1][i**+**1]]**\***(1**-**p) **+** tao
192. data\_for\_pics.append([j, path, tao])
193. **if** path[0] < min\_l:
194. min\_l **=** path[0]
196. best\_paths **=** []
197. **for** el **in** data\_for\_pics:
198. print(el)
199. **if** el[1][0] **==** min\_l **and** el[1] **not** **in** best\_paths:
200. best\_paths.append(el)
202. print()
203. **for** el **in** best\_paths:
204. print(el)
206. draw\_graph(data\_for\_pics, best\_paths, graph)
207. **return** data\_for\_pics, min\_l, best\_paths

Приложение Б. GUI.py

1. **import** PySimpleGUI as psg
2. **from** random **import** randint
3. **import** algorithm
5. MAX\_ROWS, MAX\_COLS, COL\_HEADINGS **=** 6, 6, ('    ', '0', '1', '2', '3', '4', '5')
7. layout **=** [[psg.Text('Введите коэффициент a (0, 1)', font**=**'Default 12'), psg.InputText(key**=**'a', justification**=**'r')]] **+** \
8. [[psg.Text('Введите коэффициент b (0, 1)', font**=**'Default 12'), psg.InputText(key**=**'b', justification**=**'r')]] **+** \
9. [[psg.Text('Введите коэффициент p (0, 1)', font**=**'Default 12'), psg.InputText(key**=**'p', justification**=**'r')]] **+** \
10. [[psg.Text('Введите количество итераций', font**=**'Default 12'), psg.InputText(key**=**'iters', justification**=**'r')]] **+** \
11. [[psg.Text('Введите веса в матрицу смежности', font**=**'Default 12')]] **+** \
12. [[psg.Text(s, key**=**s, enable\_events**=**True, font**=**'Courier 14', size**=**(5, 1)) **for** i, s **in** enumerate(COL\_HEADINGS)]] **+** \
13. [[psg.T(r, size**=**(4, 1))] **+** [psg.Input(randint(0, 10), justification**=**'r', key**=**(r, c), disabled**=**False) **for** c **in** range(MAX\_COLS)] **for** r **in** range(MAX\_ROWS)] **+** \
14. [[psg.Button('Рассчитать')]] **+** \
15. [[psg.Text('Исходный граф', font**=**'Default 12', visible**=**False, key**=**'t1')]] **+** \
16. [[psg.Image('pictures/initial.png', key**=**'init\_img', visible**=**False, size**=**(10, 10))] **+**
17. [psg.Image('pictures/0.png', key**=**'i\_img', visible**=**False, size**=**(10, 10))]] **+** \
18. [[psg.Button('Назад', visible**=**False, key**=**'back\_but'), psg.Button('Далее', visible**=**False, key**=**'next\_but')]] **+** \
19. [[psg.Text('Кратчайший гамильтонов цикл', font**=**'Default 12', visible**=**False, key**=**'t2')]]
21. n **=** 0
22. data **=** []
23. max\_n **=** len(data) **-** 1
24. a **=** **-**1.0
25. b **=** **-**1.0
26. p **=** **-**1.0
27. pics **=** []
29. window **=** psg.Window('Муравьиный алгоритм', layout, size**=**(900, 750), default\_element\_size**=**(8, 1), element\_padding**=**(1, 1), return\_keyboard\_events**=**True)
30. **while** True:
31. event, values **=** window.read()
32. print(event, values)
33. **if** event **in** (None, 'Exit'):
34. **break**
36. **if** event **==** 'Рассчитать':
37. best\_iters **=** []
38. pics **=** []
39. # по диагонали ставим нули и запрещаем пользователю ввод
40. **for** i **in** range(MAX\_ROWS):
41. **for** j **in** range(MAX\_COLS):
42. **if** i **==** j:
43. window[(i, j)].update(0, disabled**=**True)
45. # считываем таблицу смежности с GUI
46. graph **=** [[int(values[(row, col)]) **for** col **in** range(MAX\_COLS)] **for** row **in** range(MAX\_ROWS)]
47. a **=** float(values['a'])
48. b **=** float(values['b'])
49. p **=** float(values['p'])
50. iters **=** int(values['iters'])
52. # запускаем алгоритм
53. data, min\_l, best\_paths **=** algorithm.ant\_algorithm(graph, a, b, p, iters)
55. **for** el **in** data:
56. pics.append(el[0])
58. **for** el **in** best\_paths:
59. best\_iters.append(el[0])
61. max\_n **=** len(pics) **-** 1
63. # отображаем результаты
64. window['t1'].update(f'Исходный граф                                                                             Текущий гамильтонов цикл',
65. visible**=**True)
66. window['init\_img'].update('pictures/initial.png', visible**=**True)
67. window['i\_img'].update('pictures/0.png', visible**=**True)
68. window['t2'].update(f'Длина кратчайшего гамильтонового цикла: {min\_l}. Найдены на итерациях {best\_iters}', visible**=**True)
69. **if** max\_n > 0:
70. window['back\_but'].update(visible**=**True, disabled**=**True)
71. window['next\_but'].update(visible**=**True, disabled**=**False)
73. # логика кнопки "Назад"
74. **if** event **==** 'back\_but':
75. n **-=** 1
76. window['i\_img'].update(f'pictures/{pics[n]}.png')
77. window['next\_but'].update(disabled**=**False)
78. **if** n **==** 0:
79. window['back\_but'].update(disabled**=**True)
81. # логика кнопки "Далее"
82. **if** event **==** 'next\_but':
83. n **+=** 1
84. window['i\_img'].update(f'pictures/{pics[n]}.png')
85. window['back\_but'].update(disabled**=**False)
86. **if** n **==** max\_n:
87. window['next\_but'].update(disabled**=**True)
89. window.close()