

Государственное образовательное учреждение
высшего профессионального образования
«Московский Государственный Технический Университет
имени Н.Э. Баумана»

Отчет

По лабораторной работе №6
По курсу «Анализ Алгоритмов»
На тему «Анализ муравьиного алгоритма»

Щербатюк Дарья, ИУ7-54

МОСКВА, 2017

Оглавление

Постановка задачи	2
Описание алгоритма	2
Листинг	3
Временные эксперименты	7
Выводы	9
Заключение	10

Постановка задачи

Проанализировать влияние на время выполнения муравьиного алгоритма различных вариаций его параметров

Описание алгоритма

Задача формулируется как задача поиска минимального по стоимости замкнутого маршрута по всем вершинам без повторений на полном взвешенном графе с n вершинами. Содержательно вершины графа являются городами, которые должен посетить коммивояжер, а веса ребер отражают расстояния (длины) или стоимости проезда. Эта задача является NP-трудной, и точный переборный алгоритм ее решения имеет факториальную сложность. Моделирование поведения муравьев связано с распределением феромона на тропе — ребре графа в задаче коммивояжера. При этом вероятность включения ребра в маршрут отдельного муравья пропорциональна количеству феромона на этом ребре, а количество откладываемого феромона пропорционально длине маршрута. Чем короче маршрут, тем больше феромона будет отложено на его ребрах, следовательно, большее количество муравьев будет включать его в синтез собственных маршрутов. Моделирование такого подхода, использующего только положительную обратную связь, приводит к преждевременной сходимости — большинство муравьев двигается по локально оптимальному маршруту. Избежать этого можно, моделируя отрицательную обратную связь в виде испарения феромона. При этом если феромон испаряется быстро, то это приводит к потере памяти колонии и забыванию хороших решений, с другой стороны, большое время испарения может привести к получению устойчивого локально оптимального решения.

Листинг

ACO.PY:

```
1 import numpy as np
2 import random as rnd
3
4 MAX_DIS = 10 # maximum distance
5 MIN_DIS = 1 # minimum distance
6
7
8 def fill_dis_matr(n):
9     m = np.zeros((n, n))
10    for i in range(n):
11        for j in range(i+1, n):
12            t = rnd.randint(MIN_DIS, MAX_DIS)
13            m[i][j], m[j][i] = t, t
14    return m
15
16
17 m = 5 # amount of ants and cities
18 e = 2 # amount of elite ants
19
20 a = 2 # coefficient of strengthen the sense of smell
21 b = 1 # coefficient of strengthen of desire
22 Q = MIN_DIS * m # coefficient of the alleged best way
23 t_max = 200 # the amount of "generations"
24 p = 0.5 # coefficient of evaporation
25
26
27 def aco(m, e, d, t_max, alpha, beta, p, q):
28     nue = 1 / d # matrix of desire
29     teta = np.random.sample((m, m)) # init ferromon paths, here may
        be np.zeros((m,m))
30     T_min = None # min path
31     L_min = None # min len of path
32
33     t = 0 # the first "generation"
34
35     while t < t_max:
36         teta_k = np.zeros((m, m))
37
38         for k in range(m): # for each ant, who are in its own town
39             Tk = [k]
40             Lk = 0
41             i = k # current town
42
43             while len(Tk) != m:
44                 J = [r for r in range(m)] # generate possible to
                    visit towns
45                 for c in Tk: # remove visited towns
```

```

46         J.remove(c)
47
48     P = [0 for a in J] # probability that ant select a-
        town
49
50     for j in J:
51         if d[i][j] != 0: # if the path exist
52             buf = sum((teta[i][l] ** alpha) * (nue[i][l]
                    ** beta) for l in J)
53             P[J.index(j)] = (teta[i][j] ** alpha) * (nue
                    [i][j] ** beta) / buf
54         else:
55             P[J.index(j)] = 0
56
57     Pmax = max(P)
58     if Pmax == 0: # if all paths are zero, it's signal
        that ant is isolated
59         break
60
61     index = P.index(Pmax) # index of selected town
62     Tk.append(J[index]) # add town to way
63     Lk += d[i][J[index]] # add distance
64     i = J.pop(index) # go to selected town
65
66     if L_min is None or (Lk + d[Tk[0]][Tk[-1]]) < L_min: #
        check that it's not minimum,
67         L_min = Lk + d[Tk[0]][Tk[-1]] #
        do not forget about the way back
68     T_min = Tk
69
70     for g in range(len(Tk) - 1): # update ferromons path
71         a = Tk[g]
72         b = Tk[g + 1]
73         teta_k[a][b] += q / Lk
74
75     teta_e = (e * Q / L_min) if L_min else 0 # elite ants
76     teta = (1 - p) * teta + teta_k + teta_e # update
        ferromons after generation
77     t += 1
78
79     return T_min, L_min
80
81
82 if __name__ == "__main__":
83     D = fill_dis_matr(m) # matrix of distance
84
85     print(aco(m, e, D, t_max, a, b, p, Q))

```

TEST.PY:

```

1 import asyncio

```

```

2 from aco import aco, fill_dis_matr, MAX_DIS, MIN_DIS
3 from concurrent.futures import ThreadPoolExecutor
4 import numpy as np
5 from itertools import product
6 import time
7
8 executor = ThreadPoolExecutor(max_workers=10) # thread pool
9 loop = asyncio.get_event_loop() # event loop
10
11
12 m = 5 # amount of ants and cities
13 Q = (MIN_DIS * m, MAX_DIS * m) # coefficient of the alleged best
    way
14
15 p = (0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7) # coefficient of
    evaporation
16 a = (1, 2, 3) # coefficient of strengthen the sense of smell
17 b = (1, 2, 3) # coefficient of strengthen of desire
18 e = (0, 1, 2, 3) # amount of elite ants
19
20 good_answers = ([0, 4, 1, 3, 2], 26), ([3, 4, 0, 1, 2], 26), \
21                ([2, 0, 4, 1, 3], 26), ([2, 3, 4, 1, 0], 26), \
22                ([1, 4, 0, 2, 3], 26), ([0, 4, 3, 2, 1], 26)
23
24
25 TIMES = 3
26
27 best_time = None
28 best_set = None
29 best_q = None
30
31
32 def test_on_set(foo, params):
33     global best_time, best_set, best_q
34     s, d, t, q = params[0], params[1], params[2], params[3]
35     st = time.time()
36     res = aco(m, s[0], d, t, s[1], s[2], s[3], q)
37     en = time.time() - st
38     if res in good_answers:
39         if best_time is None or en < best_time:
40             best_time = en
41             best_set = s
42             best_q = q
43
44
45 async def level_T(d, gen):
46     await asyncio.gather(*[level_times(d, t, TIMES) for t in gen])
47
48
49 async def level_times(d, t, times):

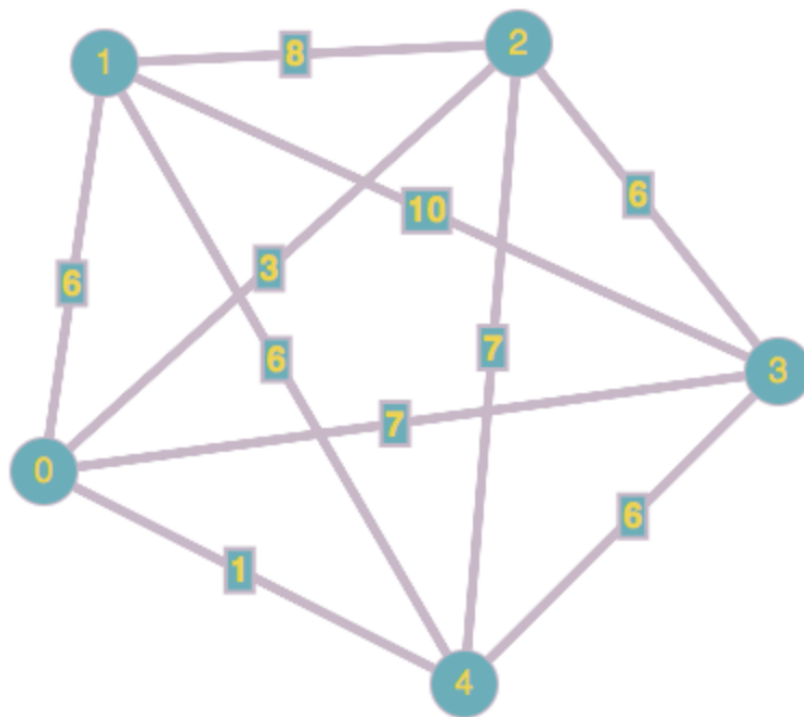
```

```

50     await asyncio.gather(*[level_Q(d, t, Q) for i in range(times)])
51
52
53 async def level_Q(d, t, Q):
54     global best_time, best_set, best_q
55     await asyncio.gather(*[level_set(d, t, q, e, a, b, p) for q in Q
56         ])
57     try:
58         print("For_{}_generations_with_{}_approx_best_time_is_{:.5f}
59             _with_set_of_parameters_"
60                 "e={};_a={};_b={};_p={}".format(
61                     t, best_q, best_time, best_set[0], best_set[1],
62                     best_set[2], best_set[3]))
63
64         best_time = None
65         best_set = None
66         best_q = None
67     except TypeError:
68         print("Oooops")
69
70
71
72 async def level_set(d, t, q, e, a, b, p):
73     await asyncio.gather(*[runner(s, d, t, q) for s in product(e, a,
74         b, p)])
75
76
77
78 async def runner(s, d, t, q):
79     param = (s, d, t, q)
80     await loop.run_in_executor(executor, test_on_set, 1, param)
81
82
83
84 async def test(d):
85     global best_time, best_set, best_q
86     t_max = (100, 200, 300, 400, 500) # the amount of "generations"
87
88     await level_T(d, t_max)
89
90
91
92 if __name__ == "__main__":
93     d_fix = np.array([[0, 6, 3, 7, 1],
94         [6, 0, 8, 10, 6],
95         [3, 8, 0, 6, 7],
96         [7, 10, 6, 0, 5],
97         [1, 6, 7, 5, 0]])
98
99     print("Test_matrix_of_distance:_{ }\n\n\n".format(d_fix))
100    loop.run_until_complete(test(d_fix))
101    #test(d_fix)

```

Временные эксперименты



	0	1	2	3	4
0	0	6	3	7	1
1	6	0	8	10	6
2	3	8	0	6	7
3	7	10	6	0	5
4	1	6	7	5	0

Рис. 1: Графовое и матричное представление

Задача приближена к реальной и имеет несколько решений:

1. 0, 4, 1, 3, 2
2. 3, 4, 0, 1, 2
3. 2, 0, 4, 1, 3
4. 2, 3, 4, 1, 0
5. 1, 4, 0, 2, 3
6. 0, 4, 3, 2, 1

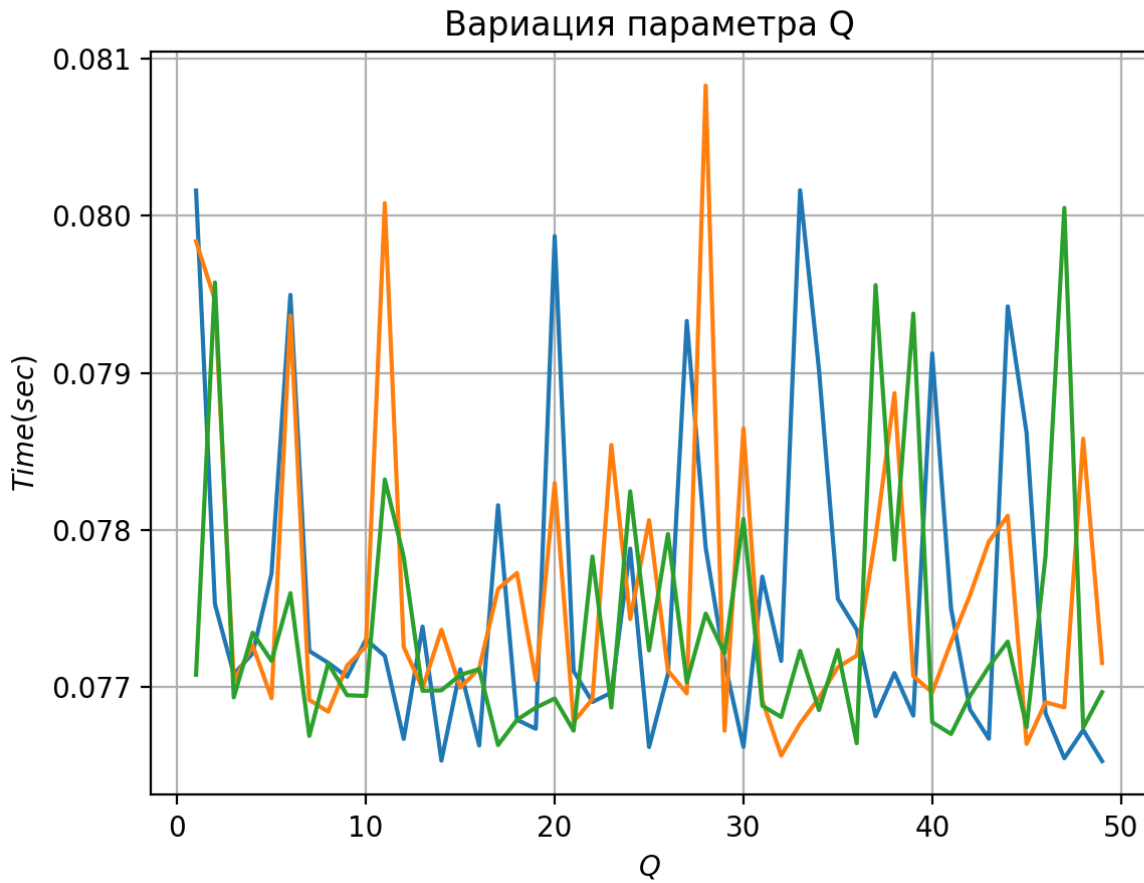
Длина кратчайший гамильтонов путь – 26. Очевидно, что количество поколений прямопропорционально времени выполнения алгоритма. Поэтому рассматривать получившиеся результаты будем в контексте каждого числа поколений.

Число поколений	Время	Q	e	α	β	p
100	0.05924	5	3	1	1	0.3
100	0.06120	5	0	2	2	0.6
100	0.05999	50	1	1	3	0.6
100	0.06030	25	2	2	2	0.6
100	0.06169	5	1	3	3	0.2
200	0.13157	5	3	3	3	0.4
200	0.13318	5	1	1	3	0.6
200	0.14144	50	2	3	3	0.3
200	0.13866	5	2	3	3	0.5
200	0.13210	5	1	3	3	0.1
300	0.22170	5	0	1	2	0.6
300	0.22347	25	1	2	1	0.4
300	0.21567	25	2	3	2	0.1
300	0.21659	50	1	1	3	0.6
300	0.21404	50	2	3	3	0.1
400	0.29997	50	3	3	1	0.5
400	0.29534	50	1	1	1	0.1
400	0.29586	25	0	2	1	0.1
400	0.28744	25	1	2	3	0.5
400	0.29609	25	2	3	2	0.2
500	0.38180	50	2	3	3	0.5
500	0.38951	50	1	2	2	0.4
500	0.39321	25	3	3	2	0.1
500	0.37518	25	0	1	1	0.5
500	0.37586	50	2	2	2	0.5

Выводы

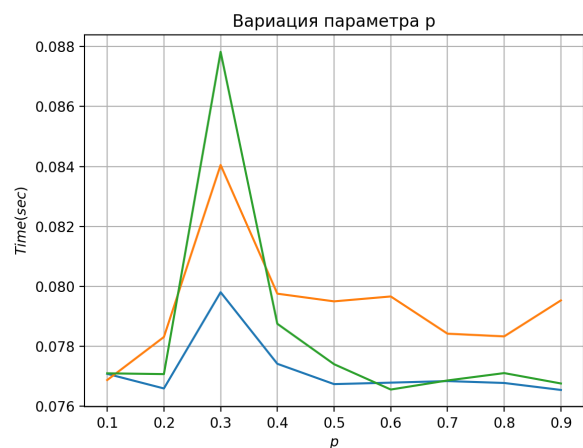
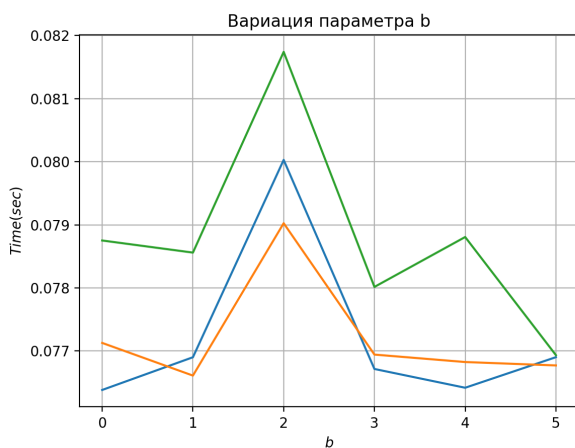
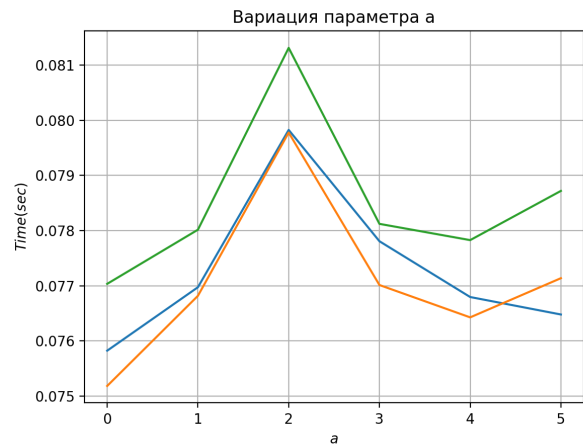
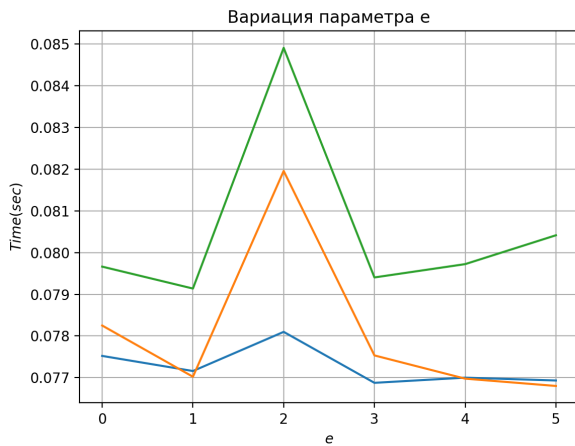
В результате проведенных испытаний алгоритма было установлено, что:

1. Судя по проведенным выше данным, самое точное приближение не всегда является самым выгодным. Проверим это утверждение на тестовом случае с фиксацией всех параметров, кроме Q . Проведем несколько тестовых запусков с $Q \in [1, 50]; t = 200; m = 5; e = 1; p = 0.3; \alpha = 1; \beta = 1$.



Как видим, результат варьируется от случая к случаю.

2. Наличие элитных муравьев улучшает сходимость. Однако в оптимальных наборах их количество сильно связано с коэффициентами α и β , отвечающими за мощность феромонного запаха и за желание муравья передвигаться по этому пути соответственно. Объяснить это можно так: так и тот, и другой аспект алгоритма усиливает ферромоновую дорожку, то их подбор должен быть сбалансирован, так как возможно "зависание" на локальных экстремумах.



3. На графиках явно видны "нежелательные" значения параметров: $e = 2$; $p = 0.3$; $\alpha = 2$; $\beta = 2$. Это может быть обусловлено математическими особенностями алгоритма.

Заключение

В ходе лабораторной работы был проанализирован муравьиный алгоритм с точки зрения оптимального набора параметров.