Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»
КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №6

Студент Топорков Павел
Группа ИУ7-53Б
Дисциплина Анализ алгоритмов

Преподаватели: _____ Строганов Ю.В., Волкова Л.Л.
подпись, дата Фамилия, И.О.

Оценка _____

Москва — 2021 г.

Оглавление

Введение

Муравьиный алгоритм — один из эффективных полиномиальных алгоритмов для нахождения приближённых решений задачи коммивояжёра, а также решения аналогичных задач поиска маршрутов на графах.

Целью данной лабораторной работы является изучение муравьиных алгоритмов и приобретение навыков параметризации методов на примере муравьиного алгоритма, примененного к задаче коммивояжера.

Задачи данной лабораторной работы:

- рассмотренть муравьиный алгоритм и алгоритм полного перебора в задаче коммивояжера;
- реализовать эти алгоритмы;
- сравнить время работы этих алгоритмов.

1 Аналитическая часть

В данной части будут рассмотрены теоретические основы задачи коммивояжера и муравьиного алгоритма.

1.1 Постановка задачи

Имеется сильно связный взвешенный ориентированный граф с положительными весами, заданный в виде матрицы смежностей.[3] Количество вершин в нем лежит в диапазоне от 5 до 20.Требуется решить задачу коммивояжера для этого графа.

1.2 Задача коммивояжера

Коммивояжёр (фр. commis voyageur) — бродячий торговец. Задача коммивояжёра — важная задача транспортной логистики, отрасли, занимающейся планированием транспортных перевозок. Коммивояжёру, чтобы распродать нужные и не очень нужные в хозяйстве товары, следует объехать n пунктов и в конце концов вернуться в исходный пункт. Требуется определить наиболее выгодный маршрут объезда. В качестве меры выгодности маршрута (точнее говоря, невыгодности) может служить суммарное время в пути, суммарная стоимость дороги, или, в простейшем случае, длина маршрута.[4]

1.3 Решение полным перебором

Задача может быть решена перебором всех вариантов объезда и выбором оптимального. Но при таком подходе количество возможных маршрутов очень быстро возрастает с ростом n (оно равно n! — количеству способов упорядочения пунктов). К примеру, для 100 пунктов ко-

личество вариантов будет представляться 158-значным числом — не выдержит ни один калькулятор! Мощная ЭВМ, способная перебирать миллион вариантов в секунду, будет биться с задачей на протяжении примерно 3*10144 лет. Увеличение производительности ЭВМ в 1000 раз даст хоть и меньшее в 1000 раз, но по-прежнему чудовищное время перебора вариантов. Не спасает ситуацию даже то, что для каждого варианта маршрута имеется 2n равноценных, отличающихся выбором начального пункта (n вариантов) и направлением обхода (2 варианта). Перебор с учётом этого наблюдения сокращается незначительно. [5]

1.4 Муравьиные алгоритмы

Все муравьиные алгоритмы базируются на моделировании поведения колонии муравьев. Колония муравьев может рассматриваться как многоагентная система, в которой каждый агент (муравей) функционирует автономнопо очень простым правилам. В противовес почтипримитивному поведению агентов, поведениевсей системы получается на удивление разумным.

Муравьиные алгоритмы представляют собой вероятностную жадную эвристику, где вероятности устанавливаются, исходя из информации о качестве решения, полученной из предыдущих решений.

Идея муравьиного алгоритма - моделирование поведения муравьёв, связанного с их способностью быстро находить кратчайший путь от муравейника к источнику пищи и адаптироваться к изменяющимся условиям, находя новый кратчайший путь.[6] При своём движении муравей метит путь феромоном, и эта информация используется другими муравьями для выбора пути. Это элементарное правило поведения и определяет способность муравьёв находить новый путь, если старый оказывается недоступным.

Какие же механизмы обеспечивают столь сложное поведение муравьев, и что можем мыпозаимствовать у этих крошечных существ длярешения своих глобальных задач? Основу «социального» поведения муравьев составляет самоорганизация — множество динамических механизмов, обеспечивающих достижение системой глобальной цели в результате низкоуровневого взаимодействия ее элементов. Принципиальнойособенностью такого взаимодействия являетсяиспользование элементами системы только локальной информации. При этом исключается любое централизованное управление и обращениек глобальному образу, репрезентирующему систему во внешнем мире.[7] Самоорганизация является результатом взаимодействия

следующих четырех компонентов:

- случайность;
- многократность;
- положительная обратная связь;
- отрицательная обратная связь.

Рассмотрим случай, когда на оптимальном доселе пути возникает преграда. В этом случае необходимо определение нового оптимального пути. Дойдя до преграды, муравьи с равной вероятностью будут обходить её справа и слева. То же самое будет происходить и на обратной стороне преграды. Однако, те муравьи, которые случайно выберут кратчайший путь, будут быстрее его проходить, и за несколько передвижений он будет более обогащён феромоном. Поскольку движение муравьёв определяется концентрацией феромона, то следующие будут предпочитать именно этот путь, продолжая обогащать его феромоном до тех пор, пока этот путь по какой-либо причине не станет недоступен.

Очевидная положительная обратная связь быстро приведёт к тому, что кратчайший путь станет единственным маршрутом движения большинства муравьёв. Моделирование испарения феромона - отрицательной обратной связи - гарантирует нам, что найденное локально оптимальное решение не будет единственным - муравьи будут искать и другие пути. Если мы моделируем процесс такого поведения на некотором графе, рёбра которого представляют собой возможные пути перемещения муравьёв, в течение определённого времени, то наиболее обогащённый феромоном путь по рёбрам этого графа и будет являться решением задачи, полученным с помощью муравьиного алгоритма.

Обобщим все выше сказанное. Любой муравьиный алгоритм, независимо от модификаций, представим в следующем виде:

- Создание муравьев;
- Поиск решения;
- Обновление феромона;
- Дополнительные действия (опционально).

Теперь рассмотрим каждый шаг в цикле более подробно:

1. Создание муравьев

Стартовая точка, куда помещается муравей, зависит ограничений, накладываемых условиями задачи. Потому что для каждой задачи способ размещения муравьёв является определяющим. Либо все они помещаются в одну точку, либо в разные с повторения, либо без повторений.

На этом же этапе задается начальный уровень феромона. Он инициализируется небольшим положительным числом для того, чтобы на начальном шаге вероятности перехода в следующую вершину не были нулевыми.

2. Поиск решения

Вероятность перехода из вершины i в вершину j определяется по следующей формуле 1.1

$$p_{i,j} = \frac{(\tau_{i,j}^{\alpha})(\eta_{i,j}^{\beta})}{\sum(\tau_{i,j}^{\alpha})(\eta_{i,j}^{\beta})} \tag{1.1}$$

где $\tau_{i,j}-$ расстояние от города i до j;

$\eta_{i,j}-$количество феромонов на ребре ij;

$\alpha-$ параметр влияния длины пути;

$\beta-$ параметр влияния феромона.

3. Обновление феромона

Уровень феромона обновляется в соответствии с приведённой формулой:

После того, как муравей успешно проходит маршрут, он оставляет на всех пройденных ребрах след, обратно пропорциональный длине пройденного пути. Итого, новый след феромона вычисляется по формуле 1.2:

$$\tau_{i,j} = (1 - \rho)\tau_{i,j} + \Delta\tau_{i,j}, \tag{1.2}$$

где $\rho_{i,j}$ - доля феромона, который испарится;

$\tau_{i,j}$ - количество феромона на дуге ij;

$\Delta\tau_{i,j}$ - количество отложенного феромона, вычисляется по формуле 1.4.

4. Дополнительные действия

Обычно здесь используется алгоритм локального поиска, однако он может также появиться и после поиска всех решений.

1.5 Муравьиный алгоритм в задаче коммивояжера

Рассмотрим, как реализовать четыре составляющие самоорганизации муравьев при оптимизации маршрута коммивояжера. Многократность взаимодействия реализуется итерационным поиском маршрута коммивояжера одновременно несколькими муравьями. При этом каждый муравей рассматривается как отдельный, независимыйкоммивояжер, решающий свою задачу. За однуитерацию алгоритма каждый муравей совершаетполный маршрут коммивояжера.Положительная обратная связь реализуется как имитация поведения муравьев типа «оставление следов – перемещение по следам». Чем больше следов оставлено на тропе — ребре графа в задаче коммивояжера, тем больше муравьев будет передвигаться по ней. При этом на тропе появляются новые следы, привлекающие дополнительных муравьев. Для задачи коммивояжера положительная обратная связь реализуется следующим стохастическим правилом: вероятность включения ребра графа в маршрут муравья пропорциональна количеству феромона на нем.

Теперь с учетом особенностей задачи коммивояжёра, мы можем описать локальные правила поведения муравьев при выборе пути.

1. Муравьи имеют собственную «память». Поскольку каждый город может быть посещён только один раз, то у каждого муравья есть список уже посещенных городов - список запретов. Обозначим через J список городов, которые необходимо посетить муравью k , находящемуся в городе i .

2. Муравьи обладают «зрением» - видимость есть эвристическое желание посетить город j , если муравей находится в городе i . Будем считать, что видимость обратно пропорциональна расстоянию между городами.

3. Муравьи обладают «обонянием» - они могут улавливать след феромона, подтверждающий желание посетить город j из города i на основании опыта других муравьёв. Количество феромона на ребре (i, j) в момент времени t обозначим через $tau_{i,j}(t)$

4. На этом основании мы можем сформулировать вероятностнопропорциональное правило, определяющее вероятность перехода k-ого муравья из города i в город j.

5. Пройдя ребро (i, j) , муравей откладывает на нём некоторое количество феромона, которое должно быть связано с оптимальностью сделанного выбора. Пусть $T_k(t)$ есть маршрут, пройденный муравьем k к моменту времени t , $L_k(t)$ - длина этого маршрута, а Q - параметр,

имеющий значение порядка длины оптимального пути. Тогда откладываемое количество феромона может быть задано в виде:

$$\Delta\tau_{i,j}^{k} = \begin{cases} Q/L_k & \text{Если k-ый мурваей прошел по ребру ij;} \\ 0 & \text{Иначе} \end{cases} \tag{1.3}$$

где Q - количество феромона, переносимого муравьем;

Тогда

$$\Delta\tau_{i,j} = \tau_{i,j}^{0} + \tau_{i,j}^{1} + ... + \tau_{i,j}^{k} \tag{1.4}$$

где k - количество муравьев в вершине графа с индексами i и j.

Вывод

В данном разделе были рассмотрены общие принципы муравьиного алгоритма и применение его к задаче коммивояжера.

2 | Конструкторская часть

В данном разделе будут рассмотрены основные требования к программе и схемы алгоритмов.

2.1 Требования к программе

Требования к вводу: у ориентированного графа должно быть хотя бы 2 вершины.

Требования к программе:

- алгоритм полного перебора должен возвращать кратчайший путь в графе.

.

Входные данные - матрица смежности графа.

Выходные данные - самый выгодный путь.

2.2 Схемы алгоритмов

На рисунке 2.1 и 2.2 приведены схемы алгоритмов решения задачи коммивояжера.

Рис. 2.1: Схема алгоритма полного перебора

Рис. 2.2: Схема муравьиного агоритма

Вывод

В данном разделе были рассмотрены требования к программе и схемы алгоритмов.

3 | Технологическая часть

Замеры времени были произведены на: Intel(R) Core(TM) i5-8300H, 4 ядра, 8 логических процессоров.

3.1 Выбор ЯП

В качестве языка программирования был выбран Golang. [1] Средой разработки Visual Studio Code. Время работы алгоритмов было замерено с помощью встроенного класса Time.

3.2 Листинг кода алгоритмов

В этой части будут рассмотрены листинги кода (листинг 3.1 - 3.3) реализованых алгоритмов.

Листинг 3.1: Реализация алгоритмов решения задачи коммивояжера

```
1  func (c *Colony) SearchAnt(d int) []int {
2    r := make([]int, len(c.w))
3
4    for i := 0; i < d; i++ {
5      for j := 0; j < len(c.w); j++ {
6        a := c.CreateAnt(j)
7        a.moveAnt()
8        cur := a.getDistance()
9
10       if (r[j] == 0) || (cur < r[j]) {
11         r[j] = cur
12       }
13     }
```

```
14    }
15
16    return r
17  }
18
19  // SearchBrute used to perform brute algorithm.
20  func SearchBrute(w [][]int) []int {
21    var (
22      path = make([]int, 0)
23      r    = make([]int, len(w))
24    )
25
26    for i := 0; i < len(w); i++ {
27      var (
28        rts  = make([][]int, 0)
29        sum  = math.MaxInt64
30        curr = 0
31      )
32      getRoutes(i, w, path, &rts)
33
34      for j := 0; j < len(rts); j++ {
35        curr = 0
36
37        for k := 0; k < len(rts[j])-1; k++ {
38          curr += w[rts[j][k]][rts[j][k+1]]
39        }
40
41        if curr < sum {
42          sum = curr
43        }
44      }
45
46      r[i] = sum
```

```
47    }
48
49    return r
50  }
51
52  // CreateAnt used to create single Ant for Ant algorithm.
53  func (c *Colony) CreateAnt(pos int) *Ant {
54    a := Ant{
55      col: c,
56      vis: make([][]int, len(c.w)),
57      isv: make([][]bool, len(c.w)),
58      pos: pos,
59    }
60
61    for i := 0; i < len(c.w); i++ {
62      a.vis[i] = make([]int, len(c.w))
63      for j := 0; j < len(c.w[i]); j++ {
64        a.vis[i][j] = c.w[i][j]
65      }
66    }
67
68    for i := range a.isv {
69      a.isv[i] = make([]bool, len(c.w))
70    }
71
72    return &a
73  }
74
75  // CreateCol used to create Ants' colony.
76  func CreateCol(w [][]int) *Colony {
77    var (
78      c = Colony{
79        w:  w,
```

```
80        ph: make([][]float64, len(w), len(w)),
81        a:  3.0,
82        b:  7.0,
83        q:  20.0,
84        p:  0.6,
85      }
86    )
87
88    for i := 0; i < len(c.ph); i++ {
89      c.ph[i] = make([]float64, len(c.w[i]))
90      for j := range c.ph[i] {
91        c.ph[i][j] = 0.5
92      }
93    }
94
95    return &c
96  }
97
98  func (a *Ant) moveAnt() {
99    for {
100     prob := a.getProb()
101     way := getWay(prob)
102     if way == -1 {
103       break
104     }
105     a.follow(way)
106     a.updatePh()
107   }
108 }
109
110 func (a *Ant) getProb() []float64 {
111   var (
112     p   = make([]float64, 0)
```

```
113     sum float64
114   )
115
116   for i, l := range a.vis[a.pos] {
117     if l != 0 {
118       d := math.Pow((float64(1)/float64(l)), a.col.a) * math.Pow(a.col.ph[a.
              pos][i], a.col.b)
119       p = append(p, d)
120       sum += d
121     } else {
122       p = append(p, 0)
123     }
124   }
125
126   for _, l := range p {
127     l /= sum
128   }
129
130   return p
131 }
132
133 func (a *Ant) getDistance() int {
134   d := 0
135
136   for i, j := range a.isv {
137     for k, z := range j {
138       if z {
139         d += a.col.w[i][k]
140       }
141     }
142   }
143
144   return d
```

```
145 }
146
147 func (a *Ant) updatePh() {
148   delta := 0.0
149
150   for i := 0; i < len(a.col.ph); i++ {
151     for j, ph := range a.col.ph[i] {
152       if a.col.w[i][j] != 0 {
153         if a.isv[i][j] {
154           delta = a.col.q / float64(a.col.w[i][j])
155         } else {
156           delta = 0
157         }
158         a.col.ph[i][j] = (1 - a.col.p) * (float64(ph) + delta)
159       }
160
161       if a.col.ph[i][j] <= 0 {
162         a.col.ph[i][j] = 0.1
163       }
164     }
165   }
166 }
167
168 func (a *Ant) follow(path int) {
169   for i := range a.vis {
170     a.vis[i][a.pos] = 0
171   }
172   a.isv[a.pos][path] = true
173   a.pos = path
174 }
175
176 func getWay(p []float64) int {
177   var (
```

```
178     r       *rand.Rand
179     sum, rn float64
180   )
181
182   for _, j := range p {
183     sum += j
184   }
185
186   r = rand.New(rand.NewSource(time.Now().UnixNano()))
187   rn = r.Float64() * sum
188   sum = 0
189
190   for i, val := range p {
191     if rn > sum && rn < sum+val {
192       return i
193     }
194     sum += val
195   }
196
197   return -1
198 }
199
200 func getRoutes(pos int, w [][]int, path []int, rts *[][]int) {
201   path = append(path, pos)
202
203   if len(path) < len(w) {
204     for i := 0; i < len(w); i++ {
205       if !isExist(path, i) {
206         getRoutes(i, w, path, rts)
207       }
208     }
209   } else {
210     *rts = append(*rts, path)
```

```
211   }
212 }
213
214 func isExist(a []int, v int) bool {
215   for _, val := range a {
216     if v == val {
217       return true
218     }
219   }
220
221   return false
222 }
```

Листинг 3.2: Реализация типов данных

```
1  // Ant used to represent sigle ant in colony.
2  type Ant struct {
3    col *Colony
4    vis [][]int
5    isv [][]bool
6    pos int
7  }
8
9  // Colony used to represent ants' environment.
10 type Colony struct {
11   w  [][]int
12   ph [][]float64
13   a  float64
14   b  float64
15   q  float64
16   p  float64
17 }
```

Листинг 3.3: Реализация вспомогательных программ

```
1
2  // Compare used to show time difference between Ant algorithm and Brute
       algorithm.
3  func Compare(fn string) {
4    ant := make([]time.Duration, 0)
5    brute := make([]time.Duration, 0)
6    for i := 2; i < 11; i++ {
7      genData(fn, i)
8      w := getWeights(fn)
9      col := CreateCol(w)
10
11     start := time.Now()
12     col.SearchAnt(100)
13     end := time.Now()
14     ant = append(ant, end.Sub(start))
15
16     start = time.Now()
17     SearchBrute(w)
18     end = time.Now()
19     brute = append(brute, end.Sub(start))
20   }
21
22   logTime("ANT ALGORITHM", ant)
23   logTime("BRUTE ALGORITHM", brute)
24 }
25
26 func genData(fn string, n int) {
27   os.Remove(fn)
28   f, err := os.OpenFile(fn, os.O_RDWR|os.O_CREATE, 0644)
29   if err != nil {
30     log.Fatal(err)
31   }
32   defer f.Close()
```

```
33
34    for i := 0; i < n; i++ {
35      for j := 0; j < n; j++ {
36        if i != j {
37          str := fmt.Sprintf("%d ", rand.Intn(10)+1)
38          f.WriteString(str)
39        } else {
40          str := fmt.Sprintf("%d ", 0)
41          f.WriteString(str)
42        }
43      }
44
45      str := fmt.Sprintf("\n")
46      f.WriteString(str)
47    }
48
49    f.Close()
50  }
51
52  func getWeights(fn string) [][]int {
53    w := make([][]int, 0)
54    f, err := os.Open(fn)
55    if err != nil {
56      log.Fatal(err)
57    }
58    defer f.Close()
59
60    rd := bufio.NewReader(f)
61    for {
62      str, err := rd.ReadString('\n')
63      if err == io.EOF {
64        break
65      }
```

```
66      str = strings.TrimSuffix(str, "\n")
67      str = strings.TrimSuffix(str, "\r")
68      str = strings.TrimRight(str, " ")
69      cur := strings.Split(str, " ")
70
71      path := make([]int, 0)
72      for _, i := range cur {
73        i, err := strconv.Atoi(i)
74        if err != nil {
75          fmt.Println(err)
76        }
77        path = append(path, i)
78      }
79      w = append(w, path)
80    }
81
82    return w
83  }
84
85  func logTime(h string, a []time.Duration) {
86    fmt.Printf("%20v\n", aurora.BgRed(h))
87    fmt.Printf("%v%18v%v\n", "+", strings.Repeat("-", 18), "+")
88    fmt.Printf("|%3v|%14v|\n", aurora.Green("N"), aurora.Green("Time"))
89    fmt.Printf("%v%18v%v\n", "+", strings.Repeat("-", 18), "+")
90    for i := 0; i < len(a); i++ {
91      fmt.Printf("|%3v|%14v|\n", i+2, a[i])
92    }
93    fmt.Printf("%v%18v%v\n", "+", strings.Repeat("-", 18), "+")
94  }
```

Вывод

В данном разделе были рассмотрены основные сведения о модулях программы и листинг кода алгоритмов.

4 Исследовательская часть

В даннном разделе будет проведен сравнительный временной анализ алгоритмов и рассмотрена параметризация муравьиного алгоритма.

4.1 Сравнительный анализ на основе замеров времени

Был проведен замер времени работы алгоритмов при разных размерах графа. На рис.4.1 и рис.4.2 показаны результаты замеров времени.

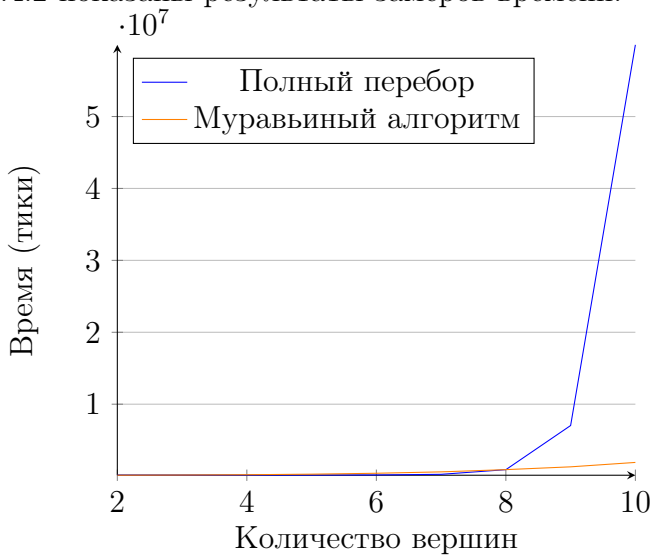

Рис. 4.1: Сравнение времени работы алгоритмов при увеличении размера графа.

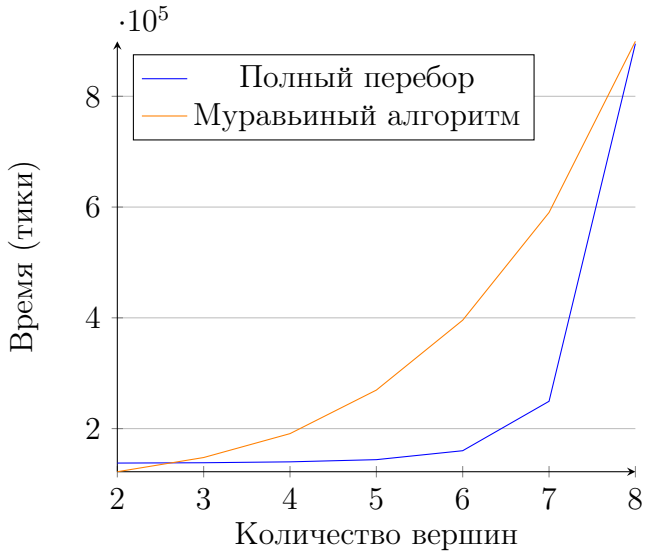

Рис. 4.2: Сравнение времени работы алгоритмов на малых размерах графа.

На рисунках 4.1 и 4.2 видно, что муравьиный алгоритм значительно выигрывает по скорости полному перебору при размере графа больше 9. До 8 вершины алгоритм полного перебора работает быстрее, с максимальным выигрышем по времени в 2,5 раза, при 6 вершинах. На графе размера 10 полный перебор работает в 30 раз медленнее.

4.2 Параметризация муравьиного алгоритма

Для различных значений параметров α, β, ρ и t_{max} для каждой из нескольких матриц смежности с помощью муравьиного алгоритма и перебора была найдена некоторая длина маршрута. Далее выбраны наилучшие сочетания параметров муравьиного алгоритма на этих данных. Параметр α менялся от 0 до 10, параметр ρ менялся от 0.1 до 0.9, параметр t_{max} менялся от 5 до 100.

Итого были выявлены оптимальные сочетания параметров (представлены в таблице 4.1):

Вывод

Сравнительный анализ по времени показал, что после на больших размерностях (размер графа больше 9) полный перебор крайне медленен относительно муравьиного алгоритма.

Таблица 4.1: Результаты решения задачи параметризации

α	β	ρ	t_{max}
4	6	0.6	20
6	4	0.3	40
1	9	0.7	50
6	4	0.9	70
3	7	0.6	80
6	4	0.25	90

Заключение

В ходе лабораторной работы я изучил возможности применения и реализовала алгоритм полного перебора и муравьиный алгоритм.

Временной анализ показал, что неэффективно использовать полный перебор на графе размера больше 10.

Литература

[1] Руководство по языку Golang [Электронный ресурс], - режим доступа: https://golang.org

[2] Основные сведения о модульных тестах [Электронный ресурс], - режим доступа: https://docs.microsoft.com/ru-ru/visualstudio/test/unit-test-basics?view=vs-2019

[3] Белоусов А.И., Ткачев С.Б(2006). Дискретная математика, 4-е издание.

[4] Т.М. Товстик, Е.В. Жукова - Алгоритм приближенного решения задачи коммивояжера.

[5] Задача коммивояжера[Электронный ресурс] - режим доступа http://mech.math.msu.su/ shvetz/54/inf/perl-problems/chCommisVoyageur.xhtml

[6] Муравьиные алгоритмы[Электронный ресурс] - режим доступа http://www.machinelearning.ru/wiki/index.php?title=

[7] Штовба С.Д. - Муравьиные алгоритмы.

[8] И. В. Белоусов(2006), Матрицы и определители, учебное пособие по линейной алгебре, с. 1 - 16