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 по курсу «Анализ алгоритмов»

Тема Муравьиный алгоритм

Студент Кононенко С.С.

Группа ИУ7-53Б

Оценка (баллы) _____

Преподаватели Волкова Л.Л., Строганов Ю.В.

Москва — 2020 г.

Оглавление

Введение

Целью данной лабораторной работы является изучение муравьиного алгоритма на примере задачи коммивояжера.

Муравьиный алгоритм – один из эффективных полиномиальных алгоритмов для нахождения приближённых решений задачи коммивояжёра, а также решения аналогичных задач поиска маршрутов на графах. Суть подхода заключается в анализе и использовании модели поведения муравьёв, ищущих пути от колонии к источнику питания, и представляет собой метаэвристическую оптимизацию.

Задачи лабораторной работы:

- рассмотреть и изучить муравьиный алгоритм и алгоритм полного перебора в задаче коммивояжера;
- сравнить временные характеристики каждого из рассмотренных алгоритмов;
- на основании проделанной работы сделать выводы.

1 Аналитическая часть

В данном разделе представлены теоретические сведения о рассматриваемых алгоритмах.

1.1 Постановка задачи

Для сильно связного взвешенного ориентированного графа с положительными весами, заданного в виде матрицы смежностей, решить задачу коммивояжера. Количество вершин в графе находится в диапазоне от 5 до 20.

1.2 Задача коммивояжера

Задача коммивояжёра – 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 может служить суммарное время в пути, суммарная стоимость дороги, или, в простейшем случае, длина маршрута.

1.3 Алгоритм полного перебора

Чтобы решить задачу алгоритмом полного перебора, нужно рассмотреть модель. Пусть все города пронумерованы от 1 до n, где n – количество городов. Если базовому городу присвоить номер n, то каждый тур по городам однозначно соответствует перестановке целых чисел $1, 2, ..., n-1$.

Задачу коммивояжера можно решить образуя все перестановки первых $n-1$ целых положительных чисел. Для каждой перестановки строится соответствующий тур и вычисляется его стоимость. Обрабатывая таким образом все перестановки, запоминается тур, который к текущему моменту

имеет наименьшую стоимость. Если находится тур с более низкой стоимостью, то дальнейшие сравнения производятся с ним.

Сложность алгоритма полного перебора составляет $O(n!)$ [1].

1.4 Муравьиный алгоритм

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. Теперь, с учетом особенностей задачи коммивояжера, можно описать локальные правила поведения муравьев при выборе пути:

1) муравьи имеют собственную «память». Поскольку каждый город может быть посещеи только один раз, у каждого муравья есть список уже посещенных городов – список запретов. Обозначим через $J_{i,k}$ 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 i и j – D_{ij}

$$\eta_{ij} = \frac{1}{D_{ij}} \tag{1.1}$$

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

На этом основании можно сформулировать вероятностно-пропорциональное правило 1.2, определяющее вероятность перехода k-ого муравья из города i в город j:

$$\begin{cases} P_{i,j,k}(t) = \frac{[\tau_{ij}(t)]^{\alpha} * [\eta_{ij}]^{\beta}}{\sum_{l \in J_{i,k}} [\tau_{il}(t)]^{\alpha} * [\eta_{il}]^{\beta}}, & j \in J_{i,k}; \\ P_{i,j,k}(t) = 0, & j \notin J_{i,k}, \end{cases} \tag{1.2}$$

где α, β – параметры, задающие веса следа феромона, при $\alpha = 0$ алгоритм вырождается до жадного алгоритма. Выбор города является вероятностным, правило 1.2 определяет ширину зоны города j; в общую зону всех городов $J_{i,k}$ бросается случайное число, которое и определяет выбор муравья. Правило 1.2 не изменяется в ходе алгоритма, но у двух разных муравьев значение вероятности перехода будут отличаться, т. к. они имеют разный список разрешенных городов.

Пройдя ребро (i, j), муравей откладывает на нем некоторое количество феромона, которое должно быть связано с оптимальностью сделанного выбора. Пусть $T_k(t)$ есть маршрут, пройденный муравьем k к моменту времени t, а $L_k(t)$ – длина этого маршрута. Пусть также Q – параметр, имеющий значение порядка длины оптимального пути. Тогда откладываемое количество феромона может быть задано в виде:

$$P_{i,j,k}(t) = \begin{cases} \frac{Q}{L_k(t)}, & (i, j) \in T_k(t); \\ 0, & (i, j) \notin T_k(t). \end{cases} \tag{1.3}$$

Правила внешней среды определяют, в первую очередь, испарение феромона. Пусть $\rho \in [0, 1]$ есть коэффициент испарения, тогда правило испарения имеет вид:

$$\tau_{ij}(t + 1) = (1 - \rho) * \tau_{ij}(t) + \Delta\tau_{ij}(t); \qquad \Delta\tau_{ij}(t) = \sum_{k=1}^{m} \Delta\tau_{ij,k}(t), \tag{1.4}$$

где m – количество муравьев в колонии.

В начале алгоритма количество феромона на ребрах принимается равным небольшому положительному числу. Общее количество муравьев остается постоянным и равным количеству городов, каждый муравей начинает маршрут из своего города.

Сложность алгоритма: $O(t_{max} * max(m, n^2))$, где t_{max} – время жизни колонии, m – количество муравьев в колонии, n – размер графа [2].

Вывод

В данной работе стоит задача реализации решения задачи коммивояжера. Были рассмотрены алгоритм полного перебора и муравьиный алгоритм.

2 Конструкторская часть

В данном разделе представлены схемы рассматриваемых алгоритмов.

2.1 Разработка алгоритмов

На рисунках 2.1 и 2.2 приведены схемы алгоритмов решения задачи коммивояжера перебором и муравьиным алгоритмом соответственно.

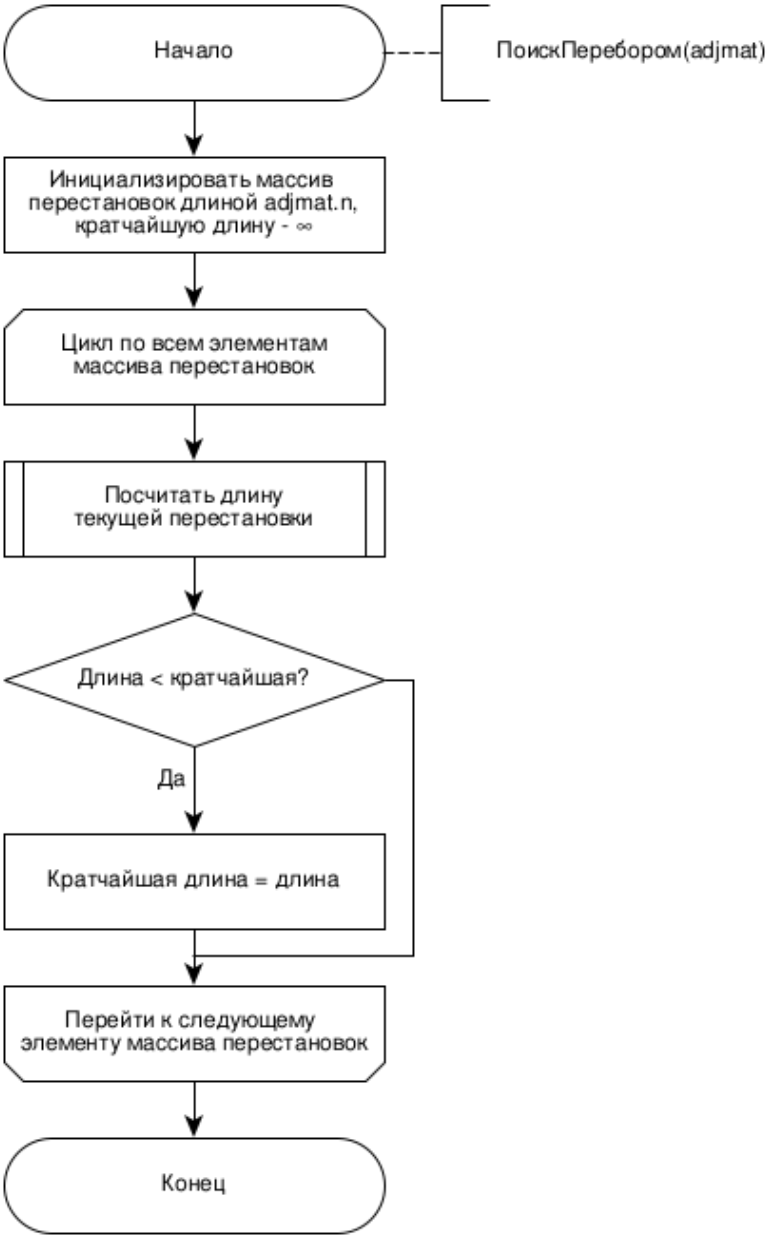

Рисунок 2.1 – Схема алгоритма решения задачи коммивояжера с использованием перебора

Рисунок 2.2 – Схема алгоритма решения задачи коммивояжера с использованием муравьиного алгоритма

2.2 Автоматическая параметризация

Автоматическая параметризация выполняет проверку дней $= [1..v]$ (где v — размер графа), $\alpha = [0..1]$, $\rho = [0..1]$ независимо друг от друга.

Алгоритм запускается три раза и минимальное значение сравнивается с эталонным. Затем на экран выводятся параметры и сравниваются с эталонными значениями.

Вывод

На основе теоретических данных, полученных из аналитического раздела, были построены схемы требуемых алгоритмов.

3 Технологическая часть

В данном разделе приведены требования к программному обеспечению, средства реализации и листинги кода.

3.1 Требования к ПО

К программе предъявляется ряд требований:

- на вход подаётся граф, представленный в виде матрицы смежности;
- на выходе – решение задачи коммивояжера для текущего графа.

3.2 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран многопоточный язык Golang [3]. Данный выбор обусловлен моим желанием расширить свои знания в области применения данного язкыа. Помимо этого, встроенные средства языка предоставляют высокоточные средства тестирования разработанного ПО.

3.3 Листинг кода

В листингах **??**, 3.3 и **??** приведены реализации алгоритмов решения задачи коммивояжера, дополнительные типы данных и вспомогательные программы соответственно.

Листинг 3.1 – Реализация алгоритмов решения задачи коммивояжера

```
package ant

import (
    "math"
    "math/rand"
    "time"
)

// SearchAnt used to perform Ant algorithm for given colony in d days.
func (c *Colony) SearchAnt(d int) []int {
    r := make([]int, len(c.w))

    for i := 0; i < d; i++ {
        for j := 0; j < len(c.w); j++ {
            a := c.CreateAnt(j)
            a.moveAnt()
            cur := a.getDistance()

            if (r[j] == 0) || (cur < r[j]) {
                r[j] = cur
            }
        }
    }

    return r
}

// SearchBrute used to perform brute algorithm.
func SearchBrute(w [][]int) []int {
    var (
        path = make([]int, 0)
        r = make([]int, len(w))
    )

    for i := 0; i < len(w); i++ {
        var (
            rts = make([][]int, 0)
            sum = math.MaxInt64
            curr = 0
        )
        getRoutes(i, w, path, &rts)

        for j := 0; j < len(rts); j++ {
            curr = 0

            for k := 0; k < len(rts[j])-1; k++ {
                curr += w[rts[j][k]][rts[j][k+1]]
```

```
48            }
49
50            if curr < sum {
51                sum = curr
52            }
53        }
54
55        r[i] = sum
56    }
57
58    return r
59}
60
61// CreateAnt used to create single Ant for Ant algorithm.
62func (c *Colony) CreateAnt(pos int) *Ant {
63    a := Ant{
64        col: c,
65        vis: make([][]int, len(c.w)),
66        isv: make([][]bool, len(c.w)),
67        pos: pos,
68    }
69
70    for i := 0; i < len(c.w); i++ {
71        a.vis[i] = make([]int, len(c.w))
72        for j := 0; j < len(c.w[i]); j++ {
73            a.vis[i][j] = c.w[i][j]
74        }
75    }
76
77    for i := range a.isv {
78        a.isv[i] = make([]bool, len(c.w))
79    }
80
81    return &a
82}
83
84// CreateCol used to create Ants' colony.
85func CreateCol(w [][]int) *Colony {
86    var (
87        c = Colony{
88            w: w,
89            ph: make([][]float64, len(w), len(w)),
90            a: 3.0,
91            b: 7.0,
92            q: 20.0,
93            p: 0.6,
94        }
95    )
```

```
96
97      for i := 0; i < len(c.ph); i++ {
98          c.ph[i] = make([]float64, len(c.w[i]))
99          for j := range c.ph[i] {
100             c.ph[i][j] = 0.5
101         }
102     }
103
104     return &c
105 }
106
107 func (a *Ant) moveAnt() {
108     for {
109         prob := a.getProb()
110         way := getWay(prob)
111         if way == -1 {
112             break
113         }
114         a.follow(way)
115         a.updatePh()
116     }
117 }
118
119 func (a *Ant) getProb() []float64 {
120     var (
121         p = make([]float64, 0)
122         sum float64
123     )
124
125     for i, l := range a.vis[a.pos] {
126         if l != 0 {
127             d := math.Pow((float64(1)/float64(l)), a.col.a) *
                    math.Pow(a.col.ph[a.pos][i], a.col.b)
128             p = append(p, d)
129             sum += d
130         } else {
131             p = append(p, 0)
132         }
133     }
134
135     for _, l := range p {
136         l /= sum
137     }
138
139     return p
140 }
141
142 func (a *Ant) getDistance() int {
```

```
143     d := 0
144
145     for i, j := range a.isv {
146         for k, z := range j {
147             if z {
148                 d += a.col.w[i][k]
149             }
150         }
151     }
152
153     return d
154 }
155
156 func (a *Ant) updatePh() {
157     delta := 0.0
158
159     for i := 0; i < len(a.col.ph); i++ {
160         for j, ph := range a.col.ph[i] {
161             if a.col.w[i][j] != 0 {
162                 if a.isv[i][j] {
163                     delta = a.col.q / float64(a.col.w[i][j])
164                 } else {
165                     delta = 0
166                 }
167                 a.col.ph[i][j] = (1 - a.col.p) * (float64(ph) + delta)
168             }
169
170             if a.col.ph[i][j] <= 0 {
171                 a.col.ph[i][j] = 0.1
172             }
173         }
174     }
175 }
176
177 func (a *Ant) follow(path int) {
178     for i := range a.vis {
179         a.vis[i][a.pos] = 0
180     }
181     a.isv[a.pos][path] = true
182     a.pos = path
183 }
184
185 func getWay(p []float64) int {
186     var (
187         r *rand.Rand
188         sum, rn float64
189     )
190
```

```
191     for _, j := range p {
192         sum += j
193     }
194 
195     r = rand.New(rand.NewSource(time.Now().UnixNano()))
196     rn = r.Float64() * sum
197     sum = 0
198 
199     for i, val := range p {
200         if rn > sum && rn < sum+val {
201             return i
202         }
203         sum += val
204     }
205 
206     return -1
207 }
208 
209 func getRoutes(pos int, w [][]int, path []int, rts *[][]int) {
210     path = append(path, pos)
211 
212     if len(path) < len(w) {
213         for i := 0; i < len(w); i++ {
214             if !isExist(path, i) {
215                 getRoutes(i, w, path, rts)
216             }
217         }
218     } else {
219         *rts = append(*rts, path)
220     }
221 }
222 
223 func isExist(a []int, v int) bool {
224     for _, val := range a {
225         if v == val {
226             return true
227         }
228     }
229 
230     return false
231 }
```

Листинг 3.2 – Реализация дополнительных типов данных

```
1 package ant
2 
3 // Ant used to represent sigle ant in colony.
4 type Ant struct {
5     col *Colony
```

```
    vis [][]int
    isv [][]bool
    pos int
}

// Colony used to represent ants' environment.
type Colony struct {
    w [][]int
    ph [][]float64
    a float64
    b float64
    q float64
    p float64
}
```

Листинг 3.3 – Реализация вспомогательных программ

```
package ant

import (
    "bufio"
    "fmt"
    "io"
    "log"
    "math/rand"
    "os"
    "strconv"
    "strings"
    "time"

    "github.com/logrusorgru/aurora"
)

// Compare used to show time difference between Ant algorithm and Brute algorithm.
func Compare(fn string) {
    ant := make([]time.Duration, 0)
    brute := make([]time.Duration, 0)
    for i := 2; i < 11; i++ {
        genData(fn, i)
        w := getWeights(fn)
        col := CreateCol(w)

        start := time.Now()
        col.SearchAnt(100)
        end := time.Now()
        ant = append(ant, end.Sub(start))

        start = time.Now()
        SearchBrute(w)
```

```
        end = time.Now()
        brute = append(brute, end.Sub(start))
    }

    logTime("ANT␣ALGORITHM", ant)
    logTime("BRUTE␣ALGORITHM", brute)
}

func genData(fn string, n int) {
    os.Remove(fn)
    f, err := os.OpenFile(fn, os.O_RDWR|os.O_CREATE, 0644)
    if err != nil {
        log.Fatal(err)
    }
    defer f.Close()

    for i := 0; i < n; i++ {
        for j := 0; j < n; j++ {
            if i != j {
                str := fmt.Sprintf("%d␣", rand.Intn(10)+1)
                f.WriteString(str)
            } else {
                str := fmt.Sprintf("%d␣", 0)
                f.WriteString(str)
            }
        }

        str := fmt.Sprintf("\n")
        f.WriteString(str)
    }

    f.Close()
}

func getWeights(fn string) [][]int {
    w := make([][]int, 0)
    f, err := os.Open(fn)
    if err != nil {
        log.Fatal(err)
    }
    defer f.Close()

    rd := bufio.NewReader(f)
    for {
        str, err := rd.ReadString('\n')
        if err == io.EOF {
            break
        }
```

```
        str = strings.TrimSuffix(str, "\n")
        str = strings.TrimSuffix(str, "\r")
        str = strings.TrimRight(str, "␣")
        cur := strings.Split(str, "␣")

        path := make([]int, 0)
        for _, i := range cur {
            i, err := strconv.Atoi(i)
            if err != nil {
                fmt.Println(err)
            }
            path = append(path, i)
        }
        w = append(w, path)
    }

    return w
}

func logTime(h string, a []time.Duration) {
    fmt.Printf("%20v\n", aurora.BgRed(h))
    fmt.Printf("%v%18v%v\n", "+", strings.Repeat("-", 18), "+")
    fmt.Printf("|%3v|%14v|\n", aurora.Green("N"), aurora.Green("Time"))
    fmt.Printf("%v%18v%v\n", "+", strings.Repeat("-", 18), "+")
    for i := 0; i < len(a); i++ {
        fmt.Printf("|%3v|%14v|\n", i+2, a[i])
    }
    fmt.Printf("%v%18v%v\n", "+", strings.Repeat("-", 18), "+")
}
```

3.4 Тестирование функций

В таблице 3.1 приведены функциональные тесты для функций, реализующих алгоритмы решения задачи коммивояжера.

Все тесты пройдены успешно.

Таблица 3.1 – Тестирование функций

Матрица смежности	Результат	Ожидаемый результат
$\begin{bmatrix} 0 & 3 & 1 & 6 & 8 \\ 3 & 0 & 4 & 1 & 0 \\ 1 & 4 & 0 & 5 & 0 \\ 6 & 1 & 5 & 6 & 1 \\ 8 & 0 & 0 & 1 & 0 \end{bmatrix}$	15	15
$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 10 & 0 & 35 & 25 \\ 15 & 35 & 0 & 30 \\ 20 & 25 & 30 & 0 \end{bmatrix}$	80	80

Вывод

Были разработаны реализации алгоритмов решения задачи коммивояжера: с использование перебора и с использованием муравьиного алгоритма.

4 Исследовательская часть

В данном разделе приведены примеры работы и анализ характеристик разработанного программного обеспечения.

4.1 Пример работы

Демонстрация работы программы приведена на рисунке 4.1.

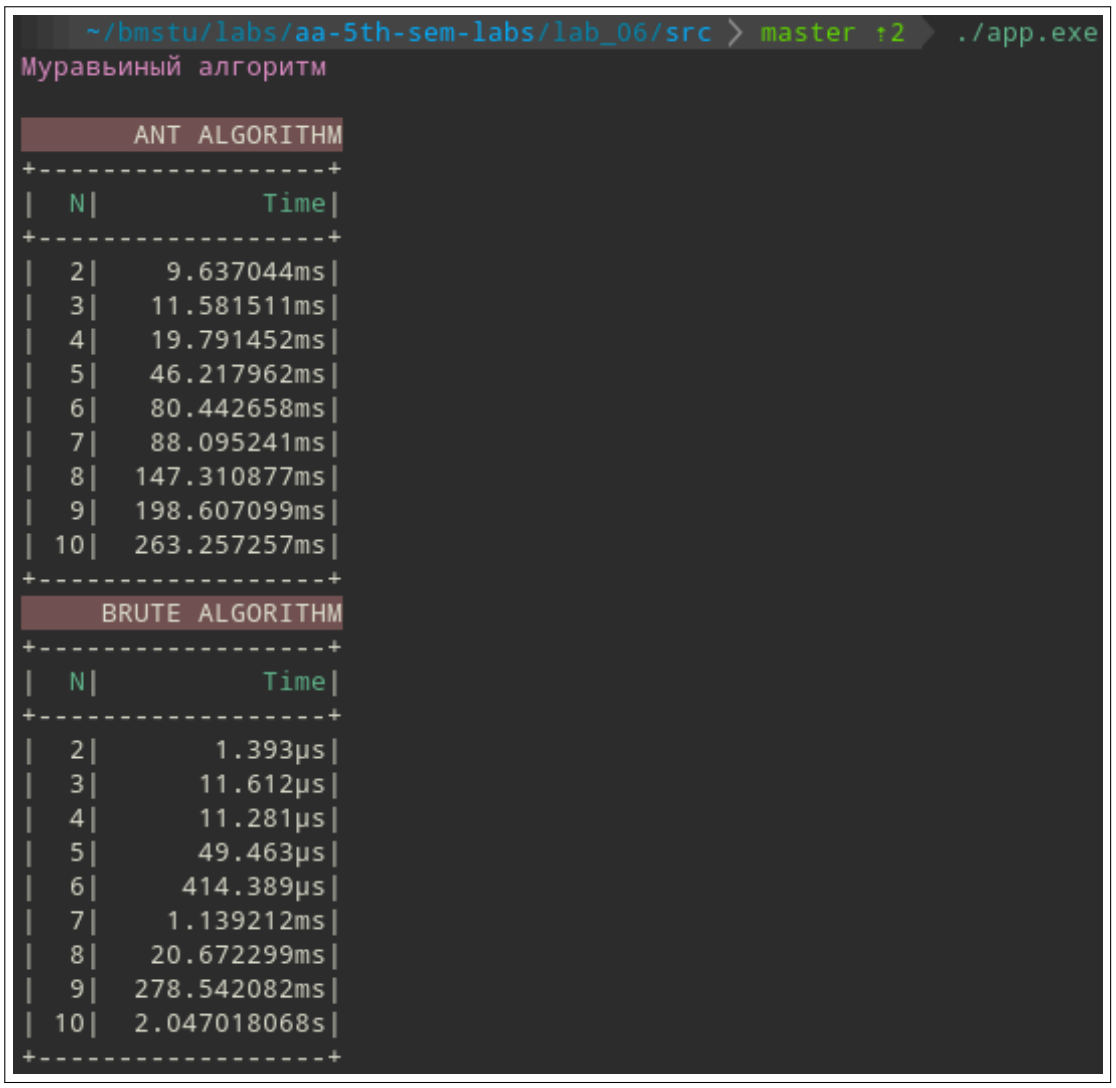

Рисунок 4.1 – Демонстрация работы алгоритмов решения задачи коммивояжера

4.2 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- Операционная система: Manjaro [4] Linux [5] 20.1 64-битная.
- Память: 16 ГБ.
- Процессор: AMD Ryzen™ 7 3700U [6] ЦПУ @ 2.30ГГц

Тестирование проводилось на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения рабочего стола, окружением рабочего стола, а также непосредственно системой тестирования.

4.3 Время выполнения алгоритмов

Время выполнения алгоритмов замерялось при помощи встроенного в Go пакета time [7], позволяющего замерить процессорное время при помощи функции time.Time.

Результаты замеров приведены в таблице 4.1. На рисунке 4.2 приведены график, иллюстрирующий зависимость времени решения задачи коммивояжера от размерности исходной матрицы смежности.

4.4 Автоматическая параметризация

В таблице 4.2 приведена выборка результатов параметризации для матрицы смежности размером 10×10. Количество дней принято равным 100. Полным перебором был посчитан оптимальный путь – он составил 130.

Таблица 4.1 – Время работы реализации алгоритмов решения задачи коммивояжера

Размерность матрицы	**Время поиска, мс**	
	Перебор	**Муравьиный**
2	0.001393	9.637044
3	0.011612	11.581511
4	0.011281	19.791452
5	0.049463	46.217962
6	0.414389	80.442658
7	1.139212	88.095241
8	20.672299	147.310877
9	278.542082	198.607099
10	2047.018	263.257257

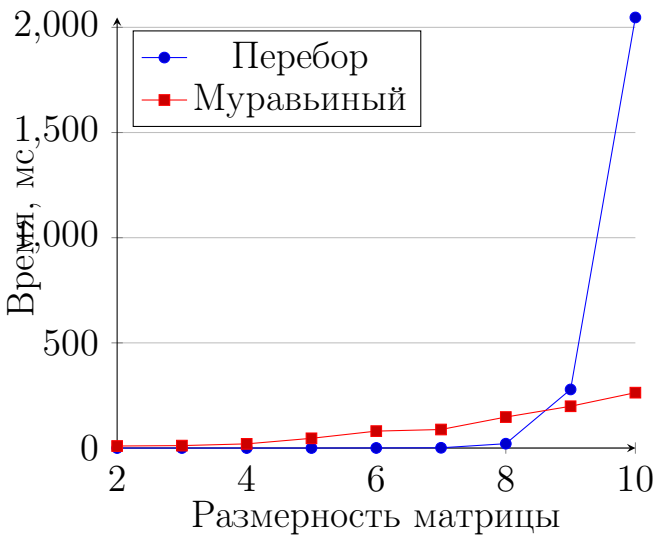

Рисунок 4.2 – Зависимость времени работы реализации алгоритмов решения задачи коммивояжера от размерности

Таблица 4.2 – Выборка из параметризации для матрицы размером 10×10.

α	β	ρ	Длина	Разница
0	1	0.0	130	0
0	1	0.3	130	0
0	1	0.5	131	1
0	1	1.0	130	0
0.1	0.9	0.0	130	0
0.1	0.9	0.3	130	0
0.1	0.9	0.6	131	1
0.1	0.9	1.0	130	0
0.2	0.8	0.0	130	0
0.2	0.8	0.3	131	1
0.2	0.8	0.6	131	1
0.2	0.8	1.0	130	0
0.3	0.7	0.0	131	1
0.3	0.7	0.4	130	0
0.3	0.7	0.9	131	1
0.3	0.7	1.0	130	0
0.4	0.6	0.0	130	0
0.4	0.6	0.4	131	1
0.4	0.6	0.5	130	0
0.4	0.6	1.0	130	0
0.5	0.5	0.0	130	0
0.5	0.5	0.3	131	1
0.5	0.5	0.7	131	1
0.5	0.5	1.0	130	0
0.6	0.4	0.2	136	6
0.6	0.4	0.6	133	3
0.6	0.4	0.7	130	0
0.7	0.3	0.0	130	0
0.7	0.3	0.3	134	4
0.7	0.3	0.6	132	2
0.8	0.2	0.0	140	10
0.8	0.2	0.5	134	4
0.8	0.2	0.7	131	1
0.8	0.2	1.0	130	0
0.9	0.1	0.0	134	4
0.9	0.1	0.3	132	2
0.9	0.1	0.5	134	4
1.0	0.0	0.0	145	25
1.0	0.0	0.4	133	3
1.0	0.0	0.7	142	22

Вывод

Исходя из проведенных исследований, можно сделать вывод, что муравьиный алгоритм решения задачи коммивояжера выигрывает у алгоритма полного перебора начиная с графа, количество вершин в котором равно 9. В случае, если количество вершин в графе меньше 9, лучше воспользоваться алгоритмом полного перебора.

Заключение

В ходе выполнения работы была достигнута цель выполнены все поставленные задачи:

- рассмотреть и изучить муравьиный алгоритм и алгоритм полного перебора в задаче коммивояжера;
- сравнить временные характеристики каждого из рассмотренных алгоритмов;
- на основании проделанной работы сделать выводы.

Муравьиный алгоритм решения задачи коммивояжера выигрывает у алгоритма полного перебора начиная с графа, количество вершин в котором равно 9. В случае, если количество вершин в графе меньше 9, лучше воспользоваться алгоритмом полного перебора.

Литература

[1] Гудман С. Хидетниеми С. Введение в разработку и анализ алгоритмов. Мир, 1981. с. 368.

[2] М.В. Ульянов. Ресурсно-эффективные компьютерные алгоритмы. Разработка и анализ. ФИЗМАТЛИТ, 2007. с. 308.

[3] The Go Programming Language [Электронный ресурс]. Режим доступа: `https://golang.org/` (дата обращения: 11.09.2020).

[4] Manjaro – enjoy the simplicity [Электронный ресурс]. Режим доступа: `https://manjaro.org/` (дата обращения: 14.09.2020).

[5] Linux – Википедия [Электронный ресурс]. Режим доступа: `https://ru.wikipedia.org/wiki/Linux` (дата обращения: 14.09.2020).

[6] Мобильный процессор AMD Ryzen™ 7 3700U с графикой Radeon™ RX Vega 10 [Электронный ресурс]. Режим доступа: `https://www.amd.com/ru/products/apu/amd-ryzen-7-3700u` (дата обращения: 14.09.2020).

[7] time – The Go Programming Language [Электронный ресурс]. Режим доступа: `https://golang.org/pkg/time/` (дата обращения: 12.09.2020).