



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»

Студент _____ Миронов Григорий Александрович _____
Группа _____ ИУ7-53Б _____
Название предприятия _____ МГТУ им. Н. Э. Баумана, каф. ИУ7 _____
Тема _____ Задача коммивояжера _____

Студент	_____	Миронов Г. А.
	(подпись, дата)	(фамилия, и.о.)
Преподаватель	_____	Волкова Л. Л.
	(подпись, дата)	(фамилия, и.о.)

Москва
2021 г.

Оглавление

Введение	3
1 Аналитическая часть	4
1.1 Постановка задачи	4
1.2 Описание алгоритмов	4
1.3 Вывод	7
2 Конструкторская часть	8
2.1 Схемы	8
2.2 Вывод	9
3 Технологическая часть	10
3.1 Выбор средств реализации	10
3.2 Требования к программному обеспечению	10
3.3 Сведения о модулях программы	10
3.4 Вывод	19
4 Экспериментальная часть	20
4.1 Технические характеристики	20
4.2 Тестирование	20
4.3 Временные характеристики	21
4.4 Автоматическая параметризация	22
4.5 Вывод	23
Заключение	24
Литература	25

Введение

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.

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.

Целью данной работы является реализация и изучение следующих алгоритмов решения задачи коммивояжера:

- муравьиный алгоритм;
- наивный алгоритм.

Для достижения данной цели необходимо решить следующие задачи:

- изучить основные методы решения задачи коммивояжера;
- реализовать каждый из указанных алгоритмов;
- сравнить временные характеристики реализованных алгоритмов экспериментально.

1 Аналитическая часть

1.1 Постановка задачи

Задача коммивояжера - одна из самых известных задач комбинаторной оптимизации, заключающаяся в поиске самого выгодного маршрута, проходящего через указанные города хотя бы по одному разу с последующим возвратом в исходный город.

В условиях задачи указываются критерий выгодности маршрута (кратчайший, самый дешёвый, совокупный критерий и тому подобное) и соответствующие матрицы расстояний, стоимости и тому подобного.

Как правило, указывается, что маршрут должен проходить через каждый город только один раз – в таком случае выбор осуществляется среди гамильтоновых циклов.

Существует несколько частных случаев общей постановки задачи, в частности:

- геометрическая задача коммивояжёра (также называемая планарной или евклидовой, когда матрица расстояний отражает расстояния между точками на плоскости);
- метрическая задача коммивояжёра (когда на матрице стоимостей выполняется неравенство треугольника), симметричная;
- асимметричная задачи коммивояжёра.

Также существует обобщение задачи, так называемая обобщённая задача коммивояжёра

1.2 Описание алгоритмов

Алгоритм полного перебора

Чтобы решить задачу алгоритмом полного перебора, нужно ввести соответствующую модель.

Пусть:

- существует n городов;
- все города пронумерованы целыми числами от 1 до n ;
- базовый (начальный) город имеет номер n .

Тогда каждый тур по городам однозначно соответствует перестановке целых чисел $1, 2, \dots, n - 1$.

Задачу коммивояжера можно решить образуя все перестановки первых $n - 1$ натуральных чисел. Для каждой перестановки строится соответствующий тур и вычисляется его стоимость. Обработывая 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].

Муравьиный алгоритм

В реальном мире муравьи (первоначально) ходят в случайном порядке и по нахождении продовольствия возвращаются в свою колонию, прокладывая феромонами тропы. Если другие муравьи находят такие тропы, они, вероятнее всего, пойдут по ним. Вместо того, чтобы отслеживать цепочку, они укрепляют её при возвращении, если в конечном итоге находят источник питания.

Со временем феромонная тропа начинает испаряться, тем самым уменьшая свою привлекательную силу. Чем больше времени требуется для прохождения пути до цели и обратно, тем сильнее испарится феромонная тропа.

На коротком пути, для сравнения, прохождение будет более быстрым, и, как следствие, плотность феромонов остаётся высокой.

Испарение феромонов также имеет функцию избегания стремления к локально-оптимальному решению. Если бы феромоны не испарялись, то путь, выбранный первым, был бы самым привлекательным. В этом случае, исследования пространственных решений были бы ограниченными.

Таким образом, когда один муравей находит (например, короткий) путь от колонии до источника пищи, другие муравьи, скорее всего пойдут по этому пути, и положительные отзывы в конечном итоге приводят всех муравьёв к одному, кратчайшему, пути.

Пример приведен на Рисунке 1.1.

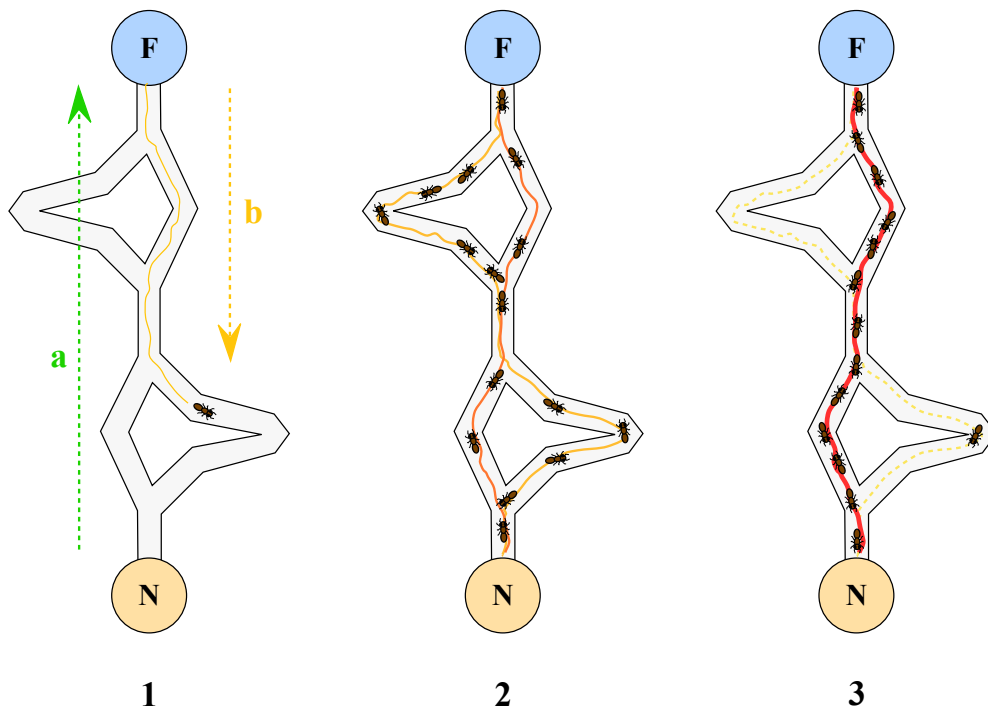


Рисунок 1.1 – Схема организации конвейерных вычислений

Применение муравьиного алгоритма к решению задачи коммивояжера имеет некоторые особенности:

- муравьи имеют собственную «память». Поскольку каждый город может быть посещен только один раз, у каждого муравья есть список уже посещенных городов. Обозначим через $J_{i,k}$ список городов, которые необходимо посетить муравью k , находящемуся в городе i ;
- 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}

$$\eta_{ij} = \frac{1}{D_{ij}} \quad (1.1)$$

- 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 $\tau_{ij}(t)$.

Приняв во внимание вышеописанные особенности конкретной реализации алгоритма, можно сформулировать правило 1.2, которое определяет вероятность перехода k -ого муравья из города i в город j :

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

где α, β – параметры, задающие веса следа феромона, при $\alpha = 0$ алгоритм вырождается до наивного (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:

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

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

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

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].

1.3 Вывод

Были рассмотрены основополагающие материалы, которые в дальнейшем потребуются при реализации алгоритмов сортировки.

2 Конструкторская часть

В данном разделе будут рассмотрены схемы вышеизложенных алгоритмов.

2.1 Схемы

На рисунке 2.1 представлена схема наивного алгоритма решения задачи коммивояжера.

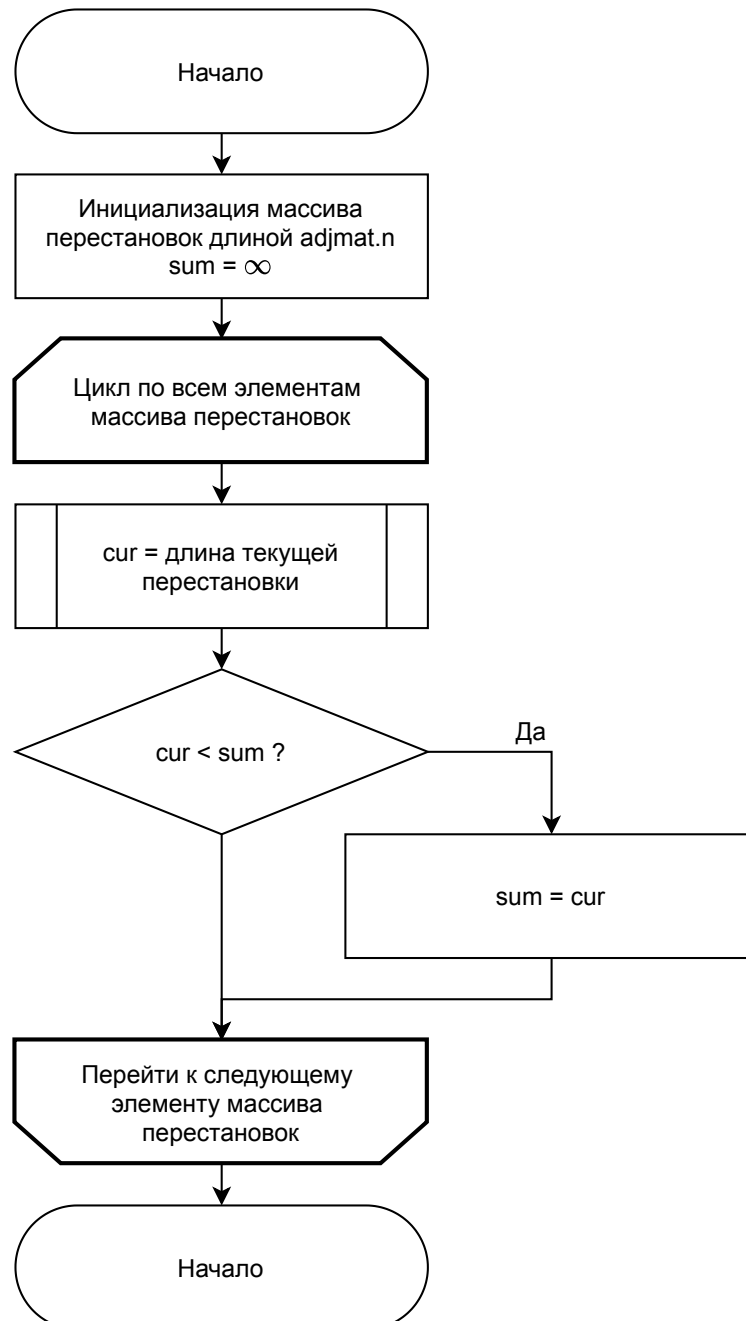


Рисунок 2.1 – Схема наивного алгоритма

На рисунке 2.2 представлена схема наивного алгоритма решения задачи коммивояжера.

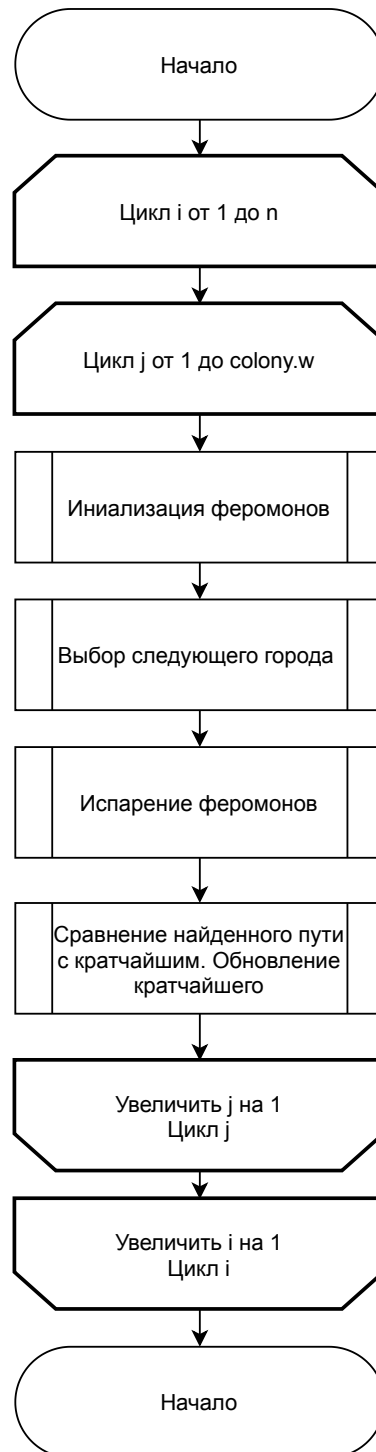


Рисунок 2.2 – Схема муравьиного алгоритма

2.2 Вывод

На основе теоретических данных, полученных из аналитического раздела, были построены схемы (рисунки 2.1 – 2.2) двух алгоритмов решения задачи коммивояжера.

3 Технологическая часть

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

3.1 Выбор средств реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран язык Golang [3]. Данный выбор обусловлен тем, что я имею некоторый опыт разработки на нем, а так же наличием у языка встроенных высокоточных средств тестирования и анализа разработанного ПО.

3.2 Требования к программному обеспечению

Входными данными являются:

- граф, представленный матрицей смежности;

На выходе - решение задачи коммивояжера для введенного графа.

3.3 Сведения о модулях программы

Данная программа разбита на следующие модули:

- `main.go` – Файл, содержащий точку входа в программу.
- `compare.go` – Файл содержит основную логику приложения.
- `utils.go` – Файл содержит реализацию утилитарных функций.
- `io.go` – Файл содержит реализацию функции ввода-вывода.
- `types.go` – Файл содержит определения и вспомогательные функции для пользовательских типов.
- `brute.go` – Файл содержит реализацию жадного алгоритма.
- `ant.go` – Файл содержит реализацию муравьиного алгоритма.

В листингах 3.1–3.11 представлены исходные коды разобранных ранее алгоритмов.

Листинг 3.1 – Основной файл программы main

```

1 package main
2
3 import (
4     "fmt"
5
6     "github.com/logrusorgru/aurora"
7
8     "lab_06/graph"
9 )
10
11 func main() {
12     fmt.Printf("%v\n\n", aurora.Magenta("Ant algorythm"))
13
14     graph.Compare("data/data.txt")
15 }

```

Листинг 3.2 – Основная логика приложения

```

1 func Compare(fn string) {
2     ant := make([]time.Duration, 0)
3     brute := make([]time.Duration, 0)
4     for i := 2; i <= 11; i++ {
5         genData(fn, i)
6         w := utils.GetWeights(fn)
7         col := CreateColony(w)
8
9         start := time.Now()
10        col.SearchAnt(100)
11        end := time.Now()
12        ant = append(ant, end.Sub(start))
13
14        start = time.Now()
15        SearchBrute(w)
16        end = time.Now()
17        brute = append(brute, end.Sub(start))
18    }
19
20    utils.LogTime("ANT ALGORITHM", ant)
21    utils.LogTime("BRUTE ALGORITHM", brute)
22 }

```

Листинг 3.3 – Утилитарные функции

```
1 func genData(fn string, n int) {
2     os.Remove(fn)
3     f, err := os.OpenFile(fn, os.O_RDWR|os.O_CREATE, 0644)
4     if err != nil {
5         log.Fatal(err)
6     }
7     defer f.Close()
8
9     for i := 0; i < n; i++ {
10         for j := 0; j < n; j++ {
11             if i != j {
12                 str := fmt.Sprintf("%d ", rand.Intn(10)+1)
13                 f.WriteString(str)
14             } else {
15                 str := fmt.Sprintf("%d ", 0)
16                 f.WriteString(str)
17             }
18         }
19         f.WriteString("\n")
20     }
21 }
```

Листинг 3.4 – Функции ввода-вывода

```

1 func GetWeights(fn string) [][]int {
2     w := make([][]int, 0)
3     f, err := os.Open(fn)
4     if err != nil {
5         log.Fatal(err)
6     }
7     defer f.Close()
8
9     rd := bufio.NewReader(f)
10    for {
11        str, err := rd.ReadString('\n')
12        if err == io.EOF {
13            break
14        }
15        str = strings.TrimSuffix(str, "\n")
16        str = strings.TrimSuffix(str, "\r")
17        str = strings.TrimRight(str, " ")
18        cur := strings.Split(str, " ")
19
20        path := make([]int, 0)
21        for _, i := range cur {
22            i, err := strconv.Atoi(i)
23            if err != nil {
24                fmt.Println(err)
25            }
26            path = append(path, i)
27        }
28        w = append(w, path)
29    }
30
31    return w
32 }

```

Листинг 3.5 – Определения пользовательских типов данных

```

1 type AntColony struct {
2     w          [][]int
3     ph          [][]float64
4     a, b, q, p float64
5 }
6
7 type Ant struct {
8     col *AntColony
9     vis [][]int
10    isv [][]bool
11    pos int
12 }
13
14 func (c *AntColony) CreateAnt(pos int) *Ant {
15     a := Ant{
16         col: c,
17         vis: make([][]int, len(c.w)),
18         isv: make([][]bool, len(c.w)),
19         pos: pos,
20     }
21     for i := 0; i < len(c.w); i++ {
22         a.vis[i] = make([]int, len(c.w))
23         for j := 0; j < len(c.w[i]); j++ {
24             a.vis[i][j] = c.w[i][j]
25         }
26     }
27     for i := range a.isv {
28         a.isv[i] = make([]bool, len(c.w))
29     }
30     return &a
31 }
32
33 func CreateColony(w [][]int) *AntColony {
34     c := AntColony{
35         w: w,
36         ph: make([][]float64, len(w)),
37         a: 3.0, b: 7.0, q: 20.0, p: 0.6,
38     }
39     for i := 0; i < len(c.ph); i++ {
40         c.ph[i] = make([]float64, len(c.w[i]))
41         for j := range c.ph[i] {
42             c.ph[i][j] = 0.5
43         }
44     }
45     return &c
46 }

```

Листинг 3.6 – Реализация жадного алгоритма. Часть 1

```

1 func SearchBrute(w [][]int) []int {
2     var (
3         path = make([]int, 0)
4         r     = make([]int, len(w))
5     )
6
7     for i := 0; i < len(w); i++ {
8         var (
9             rts = make([][]int, 0)
10            sum = math.MaxInt64
11            curr = 0
12        )
13        getRoutes(i, w, path, &rts)
14
15        for j := 0; j < len(rts); j++ {
16            curr = 0
17
18            for k := 0; k < len(rts[j])-1; k++ {
19                curr += w[rts[j][k]][rts[j][k+1]]
20            }
21
22            if curr < sum {
23                sum = curr
24            }
25        }
26
27        r[i] = sum
28    }
29
30    return r
31 }
32
33 func getRoutes(pos int, w [][]int, path []int, rts *[][]int) {
34     path = append(path, pos)
35
36     if len(path) < len(w) {
37         for i := 0; i < len(w); i++ {
38             if !isExist(path, i) {
39                 getRoutes(i, w, path, rts)
40             }
41         }
42     } else {
43         *rts = append(*rts, path)
44     }
45 }

```

Листинг 3.7 – Реализация жадного алгоритма. Часть 2

```

46
47 func isExist(a []int, v int) bool {
48     for _, val := range a {
49         if v == val {
50             return true
51         }
52     }
53
54     return false
55 }

```

Листинг 3.8 – Реализация муравьиного алгоритма. Часть 1

```

1 func (c *AntColony) 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 func (a *Ant) moveAnt() {
20     for {
21         prob := a.getProb()
22         way := getWay(prob)
23         if way == -1 {
24             break
25         }
26         a.follow(way)
27         a.updatePh()
28     }
29 }

```


Листинг 3.9 – Реализация муравьиного алгоритма. Часть 2

```

30
31 func (a *Ant) getProb() []float64 {
32     var sum float64
33
34     p := make([]float64, 0)
35
36     for i, l := range a.vis[a.pos] {
37         if l == 0 {
38             p = append(p, 0)
39         } else {
40             d := math.Pow((1.0/float64(l)), a.col.a) *
41                 math.Pow(a.col.ph[a.pos][i], a.col.b)
42             p = append(p, d)
43             sum += d
44         }
45     }
46
47     for _, l := range p {
48         l /= sum
49     }
50
51     return p
52 }
53
54 func (a *Ant) getDistance() int {
55     d := 0
56
57     for i, j := range a.isv {
58         for k, z := range j {
59             if z {
60                 d += a.col.w[i][k]
61             }
62         }
63     }
64
65     return d
66 }

```

Листинг 3.10 – Реализация муравьиного алгоритма. Часть 3

```

67
68 func (a *Ant) updatePh() {
69     delta := 0.0
70
71     for i := 0; i < len(a.col.ph); i++ {
72         for j, ph := range a.col.ph[i] {
73             if a.col.w[i][j] != 0 {
74                 if a.isv[i][j] {
75                     delta = a.col.q / float64(a.col.w[i][j])
76                 } else {
77                     delta = 0
78                 }
79                 a.col.ph[i][j] = (1 - a.col.p) * (float64(ph) + delta)
80             }
81
82             if a.col.ph[i][j] <= 0 {
83                 a.col.ph[i][j] = 0.1
84             }
85         }
86     }
87 }
88
89 func (a *Ant) follow(path int) {
90     for i := range a.vis {
91         a.vis[i][a.pos] = 0
92     }
93     a.isv[a.pos][path] = true
94     a.pos = path
95 }
96
97 func getWay(p []float64) int {
98     var sum, rn float64
99     var r *rand.Rand
100
101     for _, j := range p {
102         sum += j
103     }
104
105     r = rand.New(rand.NewSource(time.Now().UnixNano()))
106     rn = r.Float64() * sum
107     sum = 0

```

Листинг 3.11 – Реализация муравьиного алгоритма. Часть 4

```
108
109     for i, val := range p {
110         if rn > sum && rn < sum+val {
111             return i
112         }
113         sum += val
114     }
115
116     return -1
117 }
```

3.4 Вывод

Были реализованы спроектированные алгоритмы: жадный алгоритм и муравьиный алгоритм.

4 Экспериментальная часть

В данном разделе будет проведено функциональное тестирование разработанного программного обеспечения. Так же будет произведено измерение временных характеристик каждого из реализованных алгоритмов.

Для проведения подобных экспериментов на языке программирования Golang [3], используется специальный пакет `time` [4], позволяющий замерить процессорное время при помощи функции `time.Time`.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось исследование:

- процессор: Intel Core™ i5-8250U [5] CPU @ 1.60GHz;
- память: 32 GiB;
- Операционная система: Manjaro [6] Linux [7] 21.1.4 64-bit.

Исследование 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.

4.2 Тестирование

В таблице 4.1 приведены функциональные тесты для алгоритмов сортировки.

Таблица 4.1 – Таблица тестов

Первая матрица	Ожидаемый результат	Жадный	Муравьиный
$\begin{bmatrix} 0 & 3 & 1 & 6 & 8 \\ 3 & 0 & 4 & 1 & 0 \\ 1 & 4 & 0 & 5 & 0 \\ 6 & 1 & 5 & 6 & 1 \\ 8 & 0 & 0 & 1 & 1 \end{bmatrix}$	15	15	15
$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 10 & 0 & 35 & 25 \\ 15 & 35 & 0 & 30 \\ 20 & 25 & 30 & 0 \end{bmatrix}$	80	80	80

При проведении функционального тестирования, полученные результаты работы программы совпали с ожидаемыми. Таким образом, функциональное тестирование пройдено успешно.

4.3 Временные характеристики

Для сравнения графы размерностью [2, 3, 4, ..., 10]. Результаты замеров по результатам экспериментов приведены в Таблице 4.2.

Таблица 4.2 – Замер времени для графов размером от 2 до 10 узлов

Размерность графа, эл.	Время, с	
	Наивный	Муравьиный
2	1.55e-06	14.84e-03
3	3.76e-06	15.46e-03
4	9.18e-06	20.77e-03
5	28.39e-06	27.56e-03
6	137.93e-06	38.42e-03
7	834.643e-06	52.72e-03
8	7.36e-03	69.62e-03
9	97.64e-03	90.43e-03
10	998.64e-03	111.80e-03

На рисунке 4.1 приведены результаты сравнения на основе данных, представленных в Таблице 4.2.

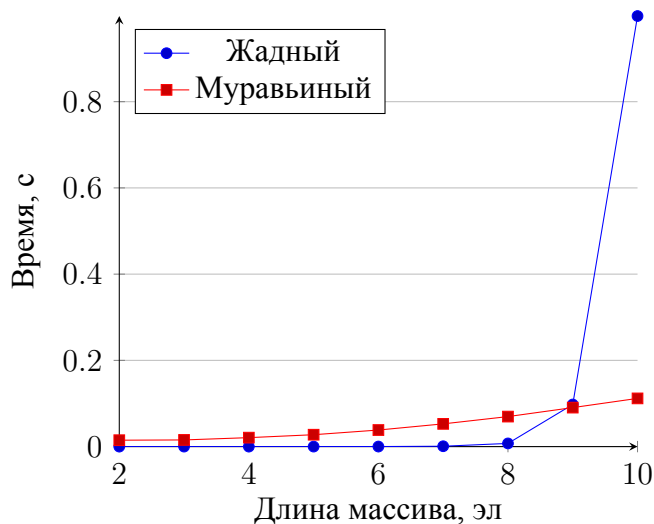


Рисунок 4.1 – Сравнение времени работы алгоритмов

Из данных, приведенных в Таблице 4.2, видно, что муравьиный алгоритм становится эффективнее жадного алгоритма при размере графа, не меньше 9 элементов. В связи с этим, при больших размерностях графа муравьиный алгоритм является более предпочтительным для использования.

4.4 Автоматическая параметризация

Таблица 4.3 – Выборка из параметризации для графа размером в 10 узлов

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	0.3	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	135	5
0.7	0.3	0.6	133	3
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	15
1.0	0.0	0.4	133	3
1.0	0.0	0.7	142	12

В таблице 4.3 приведена выборка результатов параметризации для матрицы смежности размером 10×10 . Количество дней принято равным 100. Полным перебором был посчитан оптимальный путь – он составил 130.

4.5 Вывод

В данном разделе было произведено сравнение количества затраченного времени вышеизложенных алгоритмов.

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 данной работе было рассмотрено два алгоритма решения задачи коммивояжера: жадный и муравьиный. Был описан и реализован каждый алгоритм (листинги 3.1 – 3.11). Также были показаны схемы работы алгоритмов (рисунки 2.1, 2.2) Были выбраны и обоснованы средства реализации. А также приведены тесты (таблица 4.1).

В рамках выполнения работы решены следующие задачи.

- изучение основных методов решения задачи коммивояжера;
- реализация каждого из указанных алгоритмов;
- проведено сравнение временных характеристик реализованных алгоритмов экспериментально.

Литература

- [1] С. Гудман С. Хидетниemi. Введение в разработку и анализ алгоритмов. Мир, 1981. с. 368.
- [2] Ульянов М.В. 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: 24.10.2021).
- [4] time – The Go Programming Language [Электронный ресурс]. Режим доступа: <https://pkg.go.dev/time> (дата обращения: 24.10.2021).
- [5] Процессор Intel® Core™ i5-8250U [Электронный ресурс]. Режим доступа: <https://ark.intel.com/content/www/ru/ru/ark/products/124967/intel-core-i5-8250u-processor-6m-cache-up-to-3-40-ghz.html> (дата обращения: 24.10.2021).
- [6] Manjaro - enjoy the simplicity [Электронный ресурс]. Режим доступа: <https://manjaro.org/> (дата обращения: 24.10.2021).
- [7] LINUX.ORG.RU – Русская информация об ОС Linux [Электронный ресурс]. Режим доступа: <https://www.linux.org.ru/> (дата обращения: 24.10.2021).