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

по курсу «Анализ Алгоритмов»

на тему: «Задача коммивояжера»

Студент ИУ7-53Б _____ Миронов Г. А.
(Группа) (Подпись, дата) (И. О. Фамилия)

Преподаватель _____ Волкова Л. Л.
(Подпись, дата) (И. О. Фамилия)

2021 г.

СОДЕРЖАНИЕ

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 подходы к решению задачи коммивояжера;
- привести схемы рассматриваемых алгоритмов, а именно:
  - жадный алгоритм решения задачи коммивояжера;
  - муравьиный алгоритм решения задачи коммивояжера;
- описать использующиеся структуры данных;
- описать структуру разрабатываемого програмного обеспечения;
- определить средства программной реализации;
- определить требования к программному обеспечению;
- привести сведения о модулях программы;

- провести тестирование реализованного программного обеспечения;
- провести экспериментальные замеры временных характеристик реализованных алгоритмов.

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 вввести 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, . . . , n - 1.

Задачу коммивояжера можно решить образуя все перестановки первых $n-1$ натуральных 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].

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 избежания 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.

Применение муравьиного алгоритма к решению задачи коммивояжера имеет некоторые особенности:

- муравьи имеют собственную «память». Поскольку каждый город может

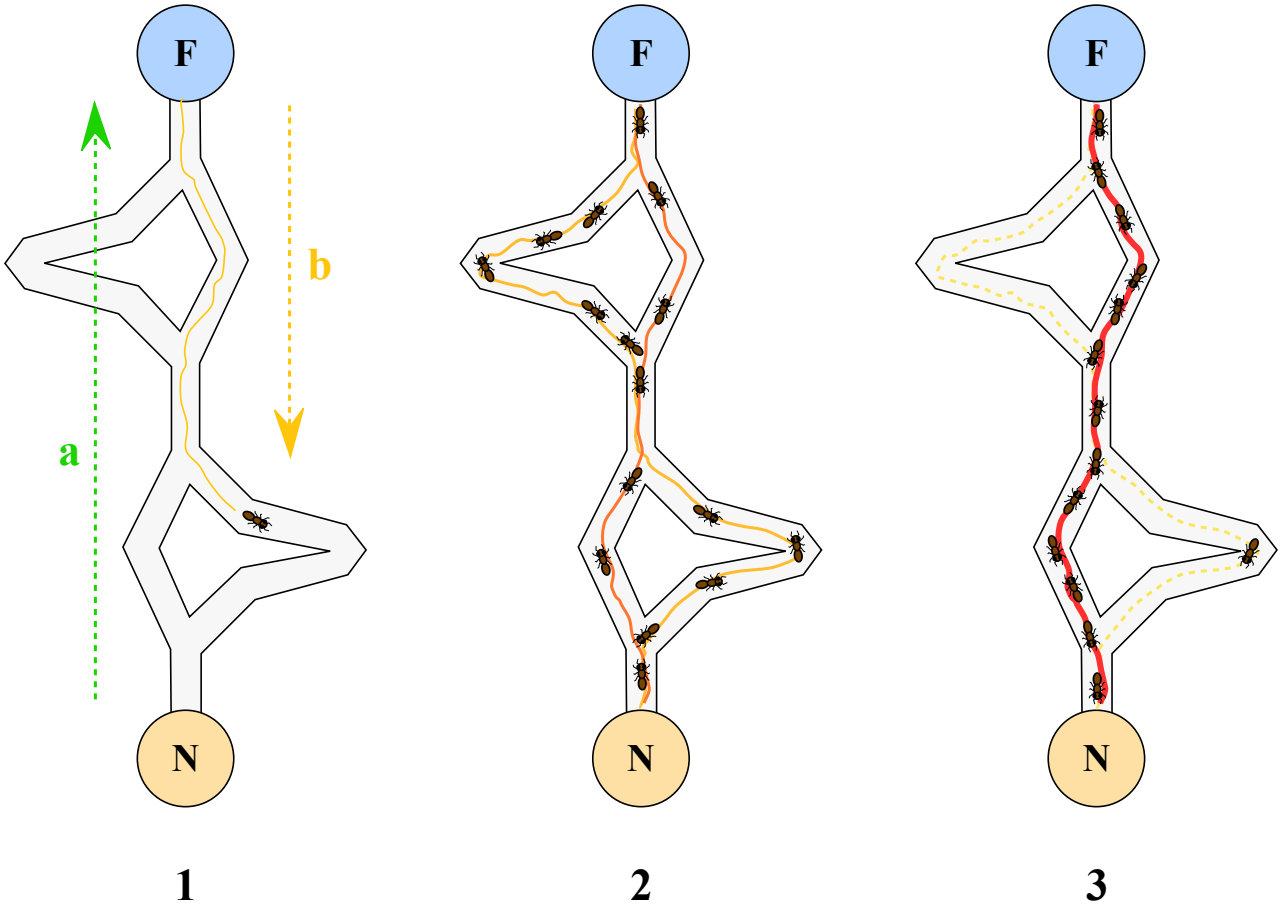

Рисунок 1.1 – Пример работы муравьиного алгоритма

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}

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

- 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:

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

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

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

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

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

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 марш-

рут 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. Так же будут описаны структуры данных, приведены структура программного обеспечения и классы эквивалентности для тестирования реализуемого программного обеспечения.

2.1 Схемы

На рисунке 2.1 представлена схема наивного алгоритма решения задачи коммивояжера.

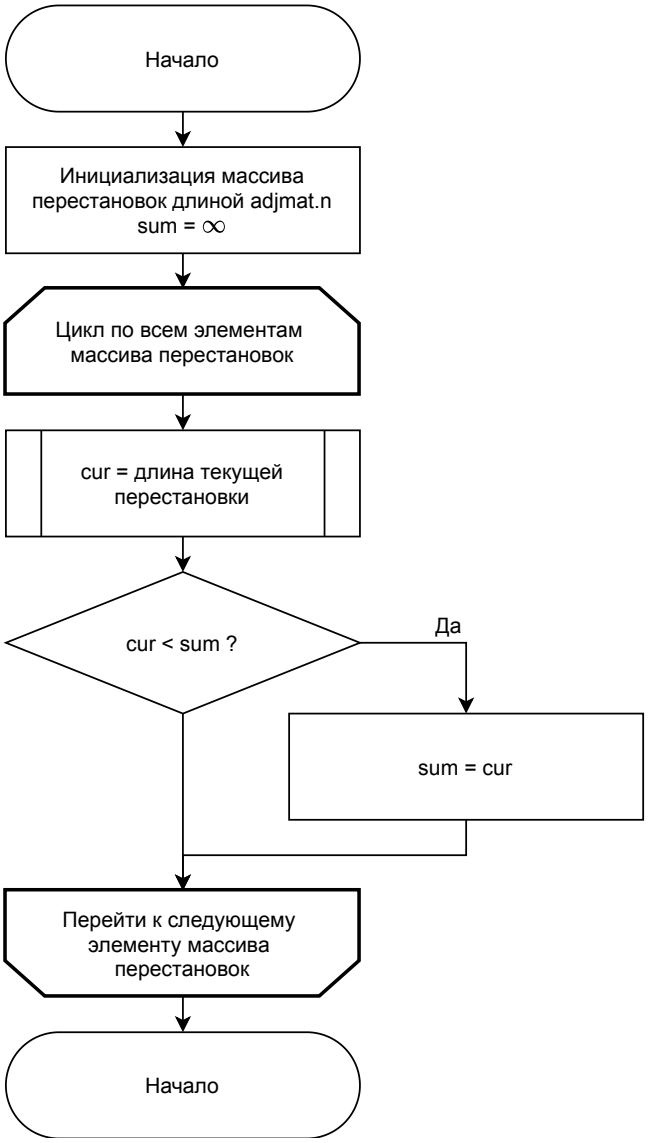

Рисунок 2.1 – Схема наивного алгоритма

На рисунке 2.2 представлена схема наивного алгоритма решения задачи коммивояжера.

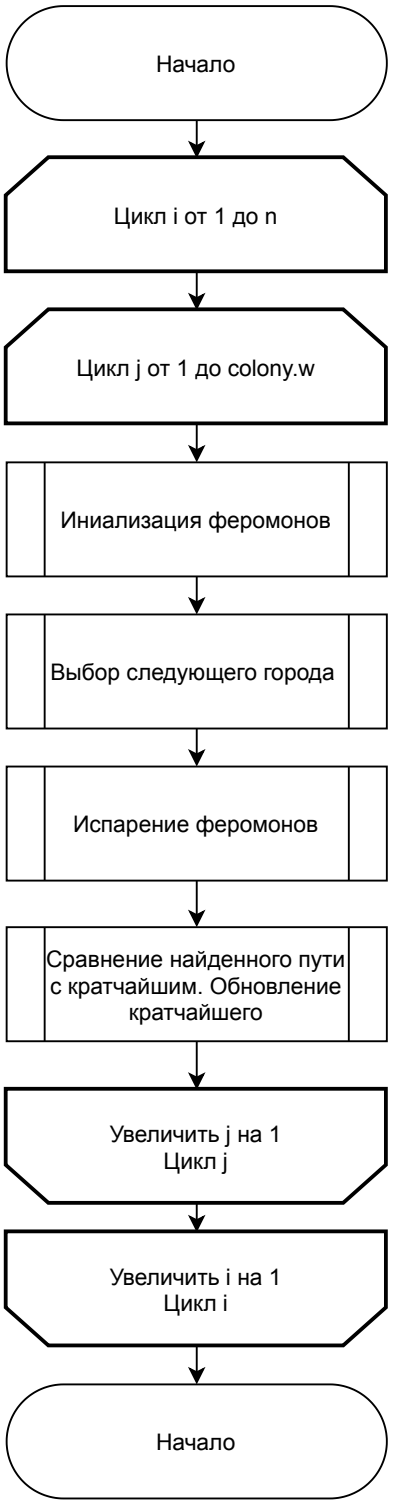

Рисунок 2.2 – Схема муравьиного алгоритма

2.2 Описание структуры программного обеспечения

На Рисунке 2.3 представлена диаграмма классов реализуемого программного обеспечения.

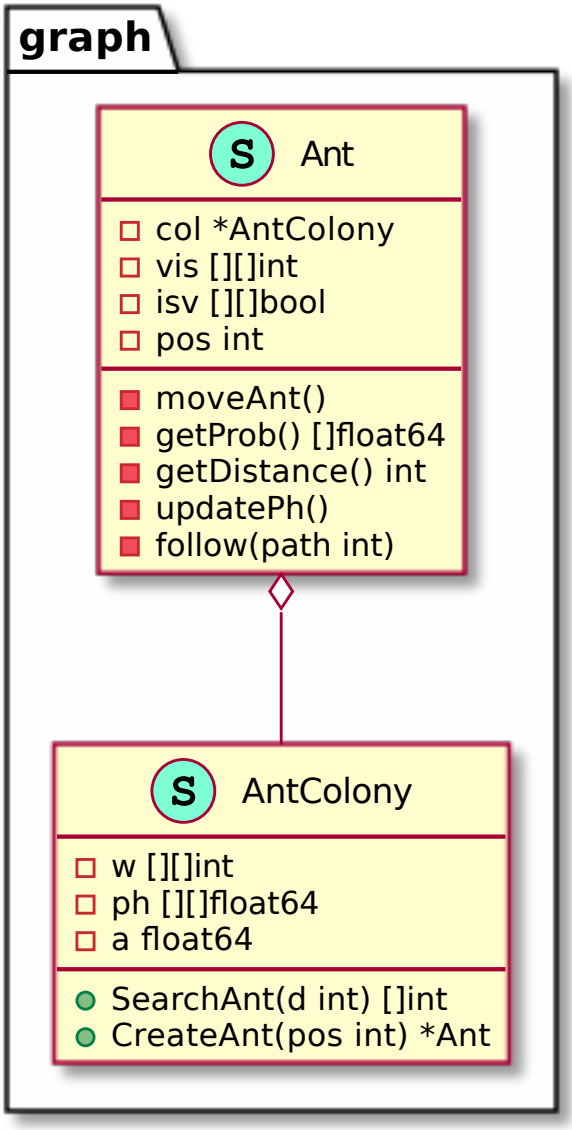

Рисунок 2.3 – Схема муравьиного алгоритма

В связи с тем, что в жадном алгоритме не используются дополнительные параметры, структуры необходимы лишь для реализации муравьиного алгоритма.

2.3 Описание структур данных

Для реализации муравьиного алгоритма введем некоторые пользовательские типы данных:

- AntColony – структура, описывающая колонию муравьев;

- Ant – структура, описывающая конкретного муравьяв колонии.

2.3.1 Описание пользовательского типа данных AntColony

Листинг 2.1 – Описание пользовательского типа данных AntColony

```
1 type AntColony struct {
2     w           [][]int
3     ph          [][]float64
4     a, b, q, p float64
5 }
```

Здесь:

- `w` – рассматриваемый граф, представленный матрицей смежности;
- `ph` – матрица, элемент (i, j) которой соответствует количеству феромона на участке дороги между городами i и j;
- `a` – значение параметра α;
- `b` – значение параметра β;
- `q` – значение параметра Q;
- `p` – значение параметра ρ.

2.3.2 Описание пользовательского типа данных Ant

Листинг 2.2 – Описание пользовательского типа данных Ant

```
1 type Ant struct {
2     col *AntColony
3     vis [][]int
4     isv [][]bool
5     pos int
6 }
```

Здесь:

- `col` – указатель на колонию, которой ринадлежит данный муравей;
- `vis` – рассматриваемый граф, представленный матрицей смежности;

- `isv` – рассматриваемый граф, представленный матрицей признаков посещения (элемент (i, j данной матрицы показывает, был ли посещен город j из города i данным муравьем);
- `pos` – номер города, в котором находится муравей.

2.4 Тестирование

В рамках данной лабораторной работы можно выделить следующие классы эквивалентности:

- входными данными является ациклический ориентированный взвешенный граф;
- входными данными является циклический ориентированный взвешенный граф.

Для проверки корректности работы реализованных алгоритмов будет проводится функциональное тестирование согласно классам эквивалентности.

2.5 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 свпомогательные 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

```
package main

import (
    "fmt"

    "github.com/logrusorgru/aurora"

    "lab_06/graph"
)

func main() {
    fmt.Printf("%v\n\n", aurora.Magenta("Ant algorythm"))

    graph.Compare("data/data.txt")
}
```

Листинг 3.2 – Основная логика приложения

```
func Compare(fn string) {
    ant := make([]time.Duration, 0)
    brute := make([]time.Duration, 0)
    for i := 2; i <= 11; i++ {
        genData(fn, i)
        w := utils.GetWeights(fn)
        col := CreateColony(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))
    }

    utils.LogTime("ANT ALGORITHM", ant)
    utils.LogTime("BRUTE ALGORITHM", brute)
}
```

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

```
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
}
```

Листинг 3.5 – Определения пользовательских типов данных

```
type AntColony struct {
    w           [][]int
    ph          [][]float64
    a, b, q, p float64
}

type Ant struct {
    col *AntColony
    vis [][]int
    isv [][]bool
    pos int
}

func (c *AntColony) CreateAnt(pos int) *Ant {
    a := Ant{
        col: c,
        vis: make([][]int, len(c.w)),
        isv: make([][]bool, len(c.w)),
        pos: pos,
    }
    for i := 0; i < len(c.w); i++ {
        a.vis[i] = make([]int, len(c.w))
        for j := 0; j < len(c.w[i]); j++ {
            a.vis[i][j] = c.w[i][j]
        }
    }
    for i := range a.isv {
        a.isv[i] = make([]bool, len(c.w))
    }
    return &a
}

func CreateColony(w [][]int) *AntColony {
    c := AntColony{
        w:  w,
        ph: make([][]float64, len(w)),
        a:  3.0, b: 7.0, q: 20.0, p: 0.6,
    }
    for i := 0; i < len(c.ph); i++ {
        c.ph[i] = make([]float64, len(c.w[i]))
        for j := range c.ph[i] {
            c.ph[i][j] = 0.5
        }
    }
    return &c
}
```

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

```
func isExist(a []int, v int) bool {
    for _, val := range a {
        if v == val {
            return true
        }
    }

    return false
}
```

Листинг 3.8 – Реализация муравьиного алгоритма. Часть 1

```
func (c *AntColony) 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
}

func (a *Ant) moveAnt() {
    for {
        prob := a.getProb()
        way := getWay(prob)
        if way == -1 {
            break
        }
        a.follow(way)
        a.updatePh()
    }
}
```

Листинг 3.9 – Реализация муравьиного алгоритма. Часть 2

```
1
2  func (a *Ant) getProb() []float64 {
3      var sum float64
4
5      p := make([]float64, 0)
6
7      for i, l := range a.vis[a.pos] {
8          if l == 0 {
9              p = append(p, 0)
10         } else {
11             d := math.Pow((1.0/float64(l)), a.col.a) *
12                 math.Pow(a.col.ph[a.pos][i], a.col.b)
13             p = append(p, d)
14             sum += d
15         }
16     }
17
18     for _, l := range p {
19         l /= sum
20     }
21
22     return p
23 }
24
25 func (a *Ant) getDistance() int {
26     d := 0
27
28     for i, j := range a.isv {
29         for k, z := range j {
30             if z {
31                 d += a.col.w[i][k]
32             }
33         }
34     }
35
36     return d
37 }
```

Листинг 3.10 – Реализация муравьиного алгоритма. Часть 3

```
1
2  func (a *Ant) updatePh() {
3      delta := 0.0
4
5      for i := 0; i < len(a.col.ph); i++ {
6          for j, ph := range a.col.ph[i] {
7              if a.col.w[i][j] != 0 {
8                  if a.isv[i][j] {
9                      delta = a.col.q / float64(a.col.w[i][j])
10                 } else {
11                     delta = 0
12                 }
13                 a.col.ph[i][j] = (1 - a.col.p) * (float64(ph) +
                       delta)
14             }
15
16             if a.col.ph[i][j] <= 0 {
17                 a.col.ph[i][j] = 0.1
18             }
19         }
20     }
21 }
22
23 func (a *Ant) follow(path int) {
24     for i := range a.vis {
25         a.vis[i][a.pos] = 0
26     }
27     a.isv[a.pos][path] = true
28     a.pos = path
29 }
30
31 func getWay(p []float64) int {
32     var sum, rn float64
33     var r *rand.Rand
34
35     for _, j := range p {
36         sum += j
37     }
38
39     r = rand.New(rand.NewSource(time.Now().UnixNano()))
40     rn = r.Float64() * sum
41     sum = 0
```

Листинг 3.11 – Реализация муравьиного алгоритма. Часть 4

```
1
2       for i, val := range p {
3           if rn > sum && rn < sum+val {
4               return i
5           }
6           sum += val
7       }
8
9       return -1
10  }
```

3.4 Тестирование

В таблице 3.1 приведены функциональные тесты для алгоритмов сортировки.

Таблица 3.1 – 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.

3.5 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 Временные характеристики

Для сравнения графы размерностью [2, 3, 4, . . . , 10]. Результаты замеров по результатам экспериментов приведены в Таблице 4.1.

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

Из данных, приведенных в Таблице 4.1, 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.

4.3 Автоматическая параметризация

Рассмотрим результаты автоматической парммметризации на двух различных классах данных.

Таблица 4.1 – Замер времени для графов размером от 2 до 10 узлов

	Время, с	
Размерность графа, эл.	**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

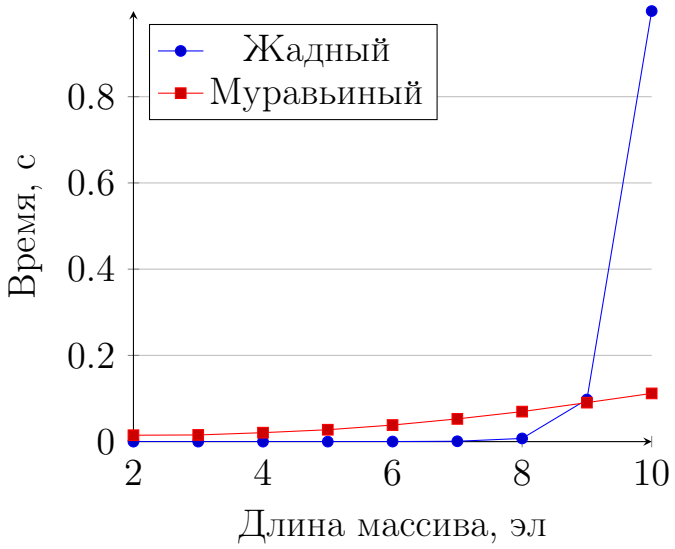

Рисунок 4.1 – Сравнение времени работы алгоритмов

4.3.1 Класс данных 1

$$M = \begin{pmatrix} 0 & 21 & 21 & 4 & 22 & 20 & 6 & 4 & 11 & 23 \\ 21 & 0 & 22 & 6 & 9 & 21 & 16 & 6 & 2 & 7 \\ 21 & 22 & 0 & 18 & 23 & 7 & 19 & 19 & 1 & 14 \\ 4 & 6 & 18 & 0 & 7 & 11 & 16 & 10 & 8 & 19 \\ 22 & 9 & 23 & 7 & 0 & 15 & 2 & 23 & 1 & 22 \\ 20 & 21 & 7 & 11 & 15 & 0 & 6 & 7 & 8 & 21 \\ 6 & 16 & 19 & 16 & 2 & 6 & 0 & 8 & 3 & 14 \\ 4 & 6 & 19 & 10 & 23 & 7 & 8 & 0 & 23 & 15 \\ 11 & 2 & 1 & 8 & 1 & 8 & 3 & 23 & 0 & 9 \\ 23 & 7 & 14 & 19 & 22 & 21 & 14 & 15 & 9 & 0 \end{pmatrix} \quad (4.1)$$

В таблице 4.2 приведены результаты параметризации метода решения задачи коммивояжера на основании муравьиного алгоритма. Количество дней было взято равным 50. Полный перебор определил оптимальную длину пути 52. Два последних стобца таблицы определяют найденный муравьиным алгоритм оптимальный путь и разницу этого пути с оптимальным путём, найденным алгоритмом полного перебора.

Таблица 4.2 – Таблица коэффициентов для класса данных 1.

α	β	ρ	Длина	Разница
0	1	0	52	0
0	1	0.1	52	0
0	1	0.2	52	0
0	1	0.3	52	0
0	1	0.4	53	1
0	1	0.5	52	0
0	1	0.6	52	0
0	1	0.7	52	0
0	1	0.8	52	0
0	1	0.9	52	0
0	1	1	52	0
0.1	0.9	0	52	0
0.1	0.9	0.1	52	0
0.1	0.9	0.2	53	1
0.1	0.9	0.3	52	0
0.1	0.9	0.4	52	0
0.1	0.9	0.5	52	0
0.1	0.9	0.6	53	1
0.1	0.9	0.7	52	0
0.1	0.9	0.8	52	0
0.1	0.9	0.9	52	0
0.1	0.9	1	52	0
0.2	0.8	0	52	0
0.2	0.8	0.1	52	0
0.2	0.8	0.2	52	0
0.2	0.8	0.3	53	1
0.2	0.8	0.4	52	0
0.2	0.8	0.5	52	0
0.2	0.8	0.6	53	1
0.2	0.8	0.7	53	1
0.2	0.8	0.8	52	0
0.2	0.8	0.9	52	0
0.2	0.8	1	52	0
0.3	0.7	0	53	1
0.3	0.7	0.1	52	0
0.3	0.7	0.2	52	0
0.3	0.7	0.3	53	1
0.3	0.7	0.4	52	0
0.3	0.7	0.5	52	0
0.3	0.7	0.6	52	0
0.3	0.7	0.7	52	0
0.3	0.7	0.8	52	0
0.3	0.7	0.9	53	1
0.3	0.7	1	52	0
0.4	0.6	0	53	1
0.4	0.6	0.1	53	1
0.4	0.6	0.2	53	1
0.4	0.6	0.3	52	0
0.4	0.6	0.4	53	1
0.4	0.6	0.5	52	0
0.4	0.6	0.6	52	0
0.4	0.6	0.7	52	0
0.4	0.6	0.8	52	0
0.4	0.6	0.9	52	0
0.4	0.6	1	52	0
0.5	0.5	0	52	0
0.5	0.5	0.1	52	0
0.5	0.5	0.2	52	0
0.5	0.5	0.3	53	1
0.5	0.5	0.4	52	0
0.5	0.5	0.5	52	0
0.5	0.5	0.6	52	0
0.5	0.5	0.7	52	0
0.5	0.5	0.8	52	0
0.5	0.5	0.9	52	0
0.5	0.5	1	52	0
0.6	0.4	0	53	1
0.6	0.4	0.1	53	1
0.6	0.4	0.2	56	4
0.6	0.4	0.3	52	0
0.6	0.4	0.4	52	0
0.6	0.4	0.5	55	3
0.6	0.4	0.6	56	4
0.6	0.4	0.7	52	0
0.6	0.4	0.8	53	1
0.6	0.4	0.9	53	1
0.6	0.4	1	52	0
0.7	0.3	0	52	0
0.7	0.3	0.1	53	1
0.7	0.3	0.2	52	0

α	β	ρ	Длина	Разница
0.7	0.3	0.3	52	0
0.7	0.3	0.4	53	1
0.7	0.3	0.5	53	1
0.7	0.3	0.6	52	0
0.7	0.3	0.7	53	1
0.7	0.3	0.8	57	5
0.7	0.3	0.9	52	0
0.7	0.3	1	52	0
0.8	0.2	0	59	7
0.8	0.2	0.1	53	1
0.8	0.2	0.2	56	4
0.8	0.2	0.3	53	1
0.8	0.2	0.4	52	0
0.8	0.2	0.5	56	4
0.8	0.2	0.6	53	1
0.8	0.2	0.7	52	0
0.8	0.2	0.8	52	0
0.8	0.2	0.9	52	0
0.8	0.2	1	53	1
0.9	0.1	0	56	4
0.9	0.1	0.1	53	1
0.9	0.1	0.2	52	0
0.9	0.1	0.3	56	4
0.9	0.1	0.4	52	0
0.9	0.1	0.5	53	1
0.9	0.1	0.6	56	4
0.9	0.1	0.7	56	4
0.9	0.1	0.8	55	3
0.9	0.1	0.9	53	1
0.9	0.1	1	53	1
1	0	0	71	19
1	0	0.1	61	9
1	0	0.2	53	1
1	0	0.3	59	7
1	0	0.4	59	7
1	0	0.5	60	8
1	0	0.6	60	8
1	0	0.7	74	22
1	0	0.8	60	8
1	0	0.9	57	5
1	0	1	60	8

4.3.2 Класс данных 2

$$M = \begin{pmatrix} 0 & 1790 & 200 & 1900 & 63 & 1659 & 1820 & 1395 & 2382 & 649 \\ 1790 & 0 & 1573 & 2435 & 1515 & 714 & 892 & 2193 & 1590 & 1003 \\ 200 & 1573 & 0 & 833 & 392 & 2404 & 962 & 902 & 141 & 1123 \\ 1900 & 2435 & 833 & 0 & 2283 & 1652 & 2362 & 2262 & 1512 & 2166 \\ 63 & 1515 & 392 & 2283 & 0 & 1322 & 290 & 1305 & 2100 & 969 \\ 1659 & 714 & 2404 & 1652 & 1322 & 0 & 256 & 78 & 2236 & 2041 \\ 1820 & 892 & 962 & 2362 & 290 & 256 & 0 & 1180 & 1547 & 1279 \\ 1395 & 2193 & 902 & 2262 & 1305 & 78 & 1180 & 0 & 1640 & 1161 \\ 2382 & 1590 & 141 & 1512 & 2100 & 2236 & 1547 & 1640 & 0 & 2212 \\ 649 & 1003 & 1123 & 2166 & 969 & 2041 & 1279 & 1161 & 2212 & 0 \end{pmatrix} \quad (4.2)$$

В таблице 4.3 приведены результаты параметризации метода решения задачи коммивояжера на основании муравьиного алгоритма для матрицы с элементами в диапазоне $[0, 2500]$. Количество дней было взято равным 50. Полный перебор определил оптимальную длину пути 6986. Два последних стобца таблицы определяют найденный муравьиным алгоритм оптимальный путь и разницу этого пути с оптимальным путём, найденным алгоритмом полного перебора.

Таблица 4.3 – Таблица коэффициентов для класса данных №2

α	β	ρ	Длина	Разница
0	1	0	6986	0
0	1	0.1	6986	0
0	1	0.2	6986	0
0	1	0.3	6986	0
0	1	0.4	6986	0
0	1	0.5	6986	0
0	1	0.6	6986	0
0	1	0.7	6986	0
0	1	0.8	6986	0
0	1	0.9	6992	6
0	1	1	6986	0
0.1	0.9	0	6986	0
0.1	0.9	0.1	6992	6
0.1	0.9	0.2	6986	0
0.1	0.9	0.3	6986	0
0.1	0.9	0.4	6986	0
0.1	0.9	0.5	6986	0
0.1	0.9	0.6	6986	0
0.1	0.9	0.7	6986	0
0.1	0.9	0.8	6986	0
0.1	0.9	0.9	7165	179
0.1	0.9	1	6986	0
0.2	0.8	0	6986	0
0.2	0.8	0.1	6986	0
0.2	0.8	0.2	6986	0
0.2	0.8	0.3	6992	6
0.2	0.8	0.4	6992	6
0.2	0.8	0.5	6992	6
0.2	0.8	0.6	6986	0
0.2	0.8	0.7	6992	6
0.2	0.8	0.8	6986	0
0.2	0.8	0.9	6986	0
0.2	0.8	1	6986	0
0.3	0.7	0	6986	0
0.3	0.7	0.1	6986	0
0.3	0.7	0.2	7139	153
0.3	0.7	0.3	7139	153
0.3	0.7	0.4	6986	0
0.3	0.7	0.5	6986	0
0.3	0.7	0.6	6986	0
0.3	0.7	0.7	6986	0
0.3	0.7	0.8	6992	6
0.3	0.7	0.9	6992	6
0.3	0.7	1	6986	0
0.4	0.6	0	6986	0
0.4	0.6	0.1	6992	6
0.4	0.6	0.2	6986	0
0.4	0.6	0.3	6986	0
0.4	0.6	0.4	6986	0
0.4	0.6	0.5	6992	6
0.4	0.6	0.6	6992	6
0.4	0.6	0.7	6986	0
0.4	0.6	0.8	7139	153
0.4	0.6	0.9	6986	0
0.4	0.6	1	6992	6
0.5	0.5	0	7139	153
0.5	0.5	0.1	6986	0
0.5	0.5	0.2	6986	0
0.5	0.5	0.3	7139	153
0.5	0.5	0.4	6986	0
0.5	0.5	0.5	6986	0
0.5	0.5	0.6	6986	0
0.5	0.5	0.7	6986	0
0.5	0.5	0.8	6986	0
0.5	0.5	0.9	6986	0
0.5	0.5	1	6986	0
0.6	0.4	0	7139	153
0.6	0.4	0.1	6992	6
0.6	0.4	0.2	6986	0
0.6	0.4	0.3	6986	0
0.6	0.4	0.4	7139	153
0.6	0.4	0.5	6992	6
0.6	0.4	0.6	6986	0
0.6	0.4	0.7	6986	0
0.6	0.4	0.8	6986	0
0.6	0.4	0.9	6992	6
0.6	0.4	1	6986	0
0.7	0.3	0	6986	0
0.7	0.3	0.1	6986	0
0.7	0.3	0.2	6986	0
0.7	0.3	0.3	7139	153

α	β	ρ	Длина	Разница
0.7	0.3	0.4	7165	179
0.7	0.3	0.5	7139	153
0.7	0.3	0.6	6992	6
0.7	0.3	0.7	6992	6
0.7	0.3	0.8	6986	0
0.7	0.3	0.9	6992	6
0.7	0.3	1	6986	0
0.8	0.2	0	7139	153
0.8	0.2	0.1	7562	576
0.8	0.2	0.2	6992	6
0.9	0.1	0.2	6992	6
0.9	0.1	0.3	6986	0
0.9	0.1	0.4	7139	153
0.9	0.1	0.5	7329	343
0.9	0.1	0.6	7217	231
0.9	0.1	0.7	7139	153
0.9	0.1	0.8	7217	231
0.9	0.1	0.9	7376	390
0.9	0.1	1	6986	0
1	0	0	8531	1545
1	0	0.1	8588	1602
1	0	0.2	6986	0
1	0	0.3	7720	734
1	0	0.4	7554	568
1	0	0.5	6992	6
1	0	0.6	7920	934
1	0	0.7	7217	231
1	0	0.8	7874	888
1	0	0.9	7446	460
1	0	1	8119	1133

4.4 Вывод

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

В результате сравнения алгоритма полного перебора и муравьиного алгоритма по времени из таблицы 4.1 были получены следующие результаты:

- при относительно небольших размерах матрицы смежности (а именно от 2 до 8) алгоритм полного перебора работает быстрее (при размере 2 — $\approx$ в 1000 раз, при размере 8 — $\approx$ в 10 раз);

- при размере матрицы смежности равном 9, время работы алгоритма полного перебора становится сопоставимым с временем работы муравьиного алгоритма.

- при размерах матрицы смежности больших 9, время работы алгоритма полного перебора начинает резко возрастать, и становится при размере 10 в 10 раз медленнее муравьиного алгоритма.

Исходя из проведенных исследований, можно сделать вывод, что муравьиный алгоритм решения задачи коммивояжера выигрывает у алгоритма полного перебора при размерностях анализируемых графов равных 9 и больше вершин. В случае, если количество вершин в графе меньше 9, лучше воспользоваться алгоритмом полного перебора.

На основе проведенной параметризации для двух классов данных можно сделать следующие выводы:

- Для класса данных №1, содержащего приблизительно равные значения, наилучшими наборами стали ($\alpha = 0.5, \beta = 0.5, \rho =$ любое), так как они показали наиболее стабильные результаты, равные эталонному значению оптимального пути, равного 52 единицам.

- Для класса данных №2, содержащего различные значения, наилучшими наборами стали ($\alpha = 0.5, \beta = 0.5, \rho =$ любое). При этих параметрах, количество найденных эталонных оптимальных путей составило 8 единиц.

Заключение

В данной работе было рассмотрено два алгоритма решения задачи коммивояжера: жадный и муравьиный. Был описан и реализован каждый 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 3.1).

В рамках выполнения работы решены следующие задачи.

- изучены подходы к решению задачи коммивояжера;
- приведены схемы рассматриваемых алгоритмов, а именно:
  - жаднлшл алгоритма решения задачи коммивояжера;
  - муравьиного алгоритма решения задачи коммивояжера;
- описаны использующиеся структуры данных;
- описана структура разрабатываемого програмного обеспечения;
- определены средства программной реализации;
- определены требования к программному обеспечению;
- приведены сведения о модулях программы;
- проведено тестирование реализованного программного обеспечения;
- проведены экспериментальные замеры временных характеристик реализованных алгоритмов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *С. Г. С. Х.* Введение в разработку и анализ алгоритмов //. — Мир, 1981. — С. 368.
2. *Ульянов М.* 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).