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»

Отчёт по лабораторной работе №6 по курсу "Анализ алгоритмов"

Тема Решение задачи коммивояжёра

Студент Леонов В.В.

Группа ИУ7-56Б

Оценка (баллы) _____

Преподаватель Волкова Л.Л.

Москва — 2021 г.

Содержание

Введение

Задача коммивояжера - 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 [1] - алгоритм эврестической оптимизации путем подражания муравьиной колонии, использующийся для нахождения приближенных решений задач коммивояжера, а также решения аналогичных NP-трудных задач поиска маршрутов на графах. Важно понимать, что большинство эврестических методов, к которым относится муравьиный алгоритм, не гарантируют точное решение, но являются достаточными для решения задачи за оптимальное время.

Целью данной лабораторной работы являются изучение проблемы коммивояжёра и реализация алгоритмов, решающих данную задачу: алгоритм полного перебора и муравьиный алгоритм.

Для достижения указанной выше цели следует выполнить следующие задачи:

- изучить задачу коммивояжера, а также идеи ее решения с помощью муравьиного алгоритма и алгоритма полного перебора;
- привести схемы изученных алгоритмов;
- описать используемые типы данных;
- описать структуру разрабатываемого ПО;
- реализовать изученные алгоритмы;
- протестировать разработанное ПО;
- выполнить на основе эксперементадьных данных сравнительный анализ временных характеристик каждого из алгоритмов в зависимости от размерности матрицы смежности;

- исследовать влияние параметров муравьиного алгоритма на точность получаемого ответа;
- описание и обоснование полученных результатов в отчете о выполненной лабораторной работе.

1 Аналитическая часть

В данном разделе выполняется формальная постановка задачи коммивояжера и математическое описание каждого из алгоритмов.

1.1 Постановка задачи

Пусть есть N городов, соединенных между собой. Странствующий торговец (т.н. коммивояжер) объезжает эти города. Ему необходимо составить такой маршрут, чтобы он был наикратчайшим, и при этом каждый из N городов должен быть посещен ровно по одному разу. В конце маршрута коммивояжер должен вернуться в город, с которого он начинал маршрут.

Для возможности применения математического аппарата для решения задачи её следует представить в виде математической модели.

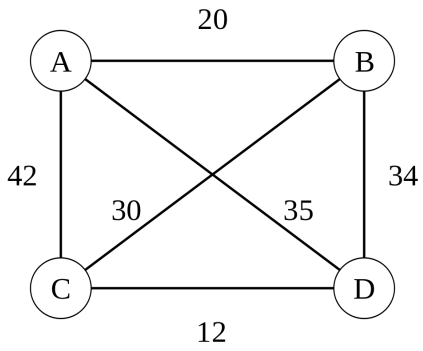

Рисунок 1.1 – Симметричная задача для четырех городов

Задачу коммивояжера можно представить в виде модели на графе. Вершины графа - города, ребра между вершинами - пути между этими городами. Каждому ребру в соответствие ставится его вес, который в данной задаче определяет расстояние между городами, определяемыми вершинами.

Другими словами, задан взвешенный полностью связный граф. Для него необходимо найти гамильтонов цикл минимального веса.

1.2 Алгоритм полного перебора

Задачу коммивояжера можно решить, рассмотрев все возможные комбинации городов. Для каждого из маршрутов высчитывается его суммарная длина. Выбирается такой маршрут, для которого его суммарная длина минимальна.

Такой алгоритм гарантирует точное решение, однако, как известно, алгоритмическая сложность подобных алгоритмов составляет O(n!), поэтому даже при небольшом количестве городов решение за оптимальное время становится невозможным.

1.3 Муравьиный алгоритм

Муравьиный алгоритм основан на особенностях поведения муравьёв в природе. При поиске путей к источникам пищи муравьи помечают пройденный путь специальным веществом — феромоном.

Феромон играет роль непрямой обратной связи: чем больше муравьёв движется по помеченному пути, тем сильнее он привлекает других насекомых. По истечении некоторого времени феромон испаряется.

Таким образом, большая часть муравьёв будет передвигаться от муравейника до найденного источника пищи по одному и тому же пути.

В вершинах описанного ранее графа располагаются муравьи, которые перемещаются по рёбрам графа. Для каждого муравья вводится понятие памяти (или списка запретов) — списка пройденных им за день узлов. Выбирая узел для следующего шага, муравей помнит об уже пройденных узлах и не рассматривает их в качестве возможных для перехода. Также вводится понятие зрения: муравей может оценить длины рёбер до следующих городов.

Пусть имеется N городов и N муравьёв. Вводится матрица смежности D и матрица феромонов T.

Колония помещается в одну из вершин графа, называемую стартовой вершиной. Все рёбра графа помечаются одинаковым фоновым значением феромона $T_{ij} = T_0$. Муравьи каждое утро выходят из муравейника и неза-

висимо друг от друга перемещаются между вершинами.

При выборе следующей вершины для посещения муравьи опираются на значения концентрации феромона на смежных рёбрах, которые не находятся в списке запрета, и на значения длин этих рёбер. К закату муравьи возвращаются в муравейник и обсуждают, кто нашёл самый короткий маршрут. Если новый маршрут лучше прежнего, то его запоминают. Ночью происходит обновление феромонов на рёбрах.

Пусть k-ый муравей находится в i-ой вершине, а его запретный список V_k ещё не до конца заполнен. Тогда вероятность перемещения в j-ую вершину определяется следующей формулой.

$$P_{ij}^k = \begin{cases} \dfrac{T_{ij}^\alpha \cdot \eta_{ij}^\beta}{\sum\limits_{r \in V_{ik}} T_{ir}^\alpha \cdot \eta_{ir}^\beta}, & \text{вершина } j \text{ сегодня не была посещена } k\text{-ым муравьём} \\ 0, & \text{иначе} \end{cases}$$

Здесь α — коэффициент жадности, β — коэффициент стадности, а $\eta_{ij} = \dfrac{1}{D_{ij}}$ — привлекательность ребра. $\alpha + \beta = \text{const}$.

Если $\alpha = 0$, то алгоритм вырождается в чисто стадный (куда стадо, туда и все). Если $\beta = 0$, то алгоритм вырождается в жадный, потому что опыт колонии не учитывается.

Когда все муравьи завершили обход, происходит обновление феромона по следующей формуле.

$$T_{ij}(t+1) = (1-p) \cdot (T_{ij}(t) + \Delta T_{ij}), \text{ где}$$

$$\Delta T_{ij} = \sum_{k=1}^{N} \Delta T_{ij}^k$$

$$\Delta T_{ij}^k = \begin{cases} \dfrac{Q}{L_k}, & \text{если ребро посещено } k\text{-ым муравьём} \\ 0, & \text{иначе} \end{cases}$$

L_k — длина пути k-го муравья, Q — некоторая константа, соразмер-

ная длине лучшего пути (запас феромона), ΔT_{ij}^k — количество феромона, оставленного k-ым муравьём на ij-ом ребре. Таким образом, чем короче путь k-го муравья, тем больше феромона он оставит на пройденных рёбрах.

Описанные действия представляют собой одну итерацию муравьиного алгоритма. Итерации повторяются до тех пор, пока не окажется выполненным какой-либо из критериев останова алгоритма: исчерпано число итераций, достигнута нужная точность, получен единственный путь (алгоритм сошелся к некоторому решению).

Нужно также учитывать случай, когда феромон может обнулиться — тогда обнулится вся вероятность прохода по этому ребру. Чтобы этого не допустить, вводится константа фонового значения феромона. Если значение феромона становится меньше фонового, то ему присваивается фоновое значение.

Чтобы оставить возможность прохода по не самому оптимальному с точки зрения вероятности пути, муравей подбрасывает монетку, получая случайное число от 0 до 1. Если искать только максимум вероятностей — это в своем роде жадное решение.

1.4 Параметризация муравьиного алгоритма

Для эвристических методов проводится параметризация. Параметризация заключается в определении таких параметров или настроек работы метода, при которых для выбранного класса данных задача решается с наилучшим качеством.

Грубо говоря, метод складывается из некоторого набора алгоритмов и некоторой математической модели представления задачи в предметной области. Для эвристических методов вводятся некоторые дополнительные понятия, которые составляют часть модели, которая представляет предметную область.

Здесь вводятся понятия муравья (некоторой части алгоритма, которая соответствует виденью муравья части задачи) и феромона (средства обме-

на информацией).

Для этой задачи могут быть разные классы данных. Предположим, это один вид карт с однотипным разбросом длин рёбер. Нужно подстроить метод под какой-то конкретный класс данных, так как придумать универсальный алгоритм трудно.

У алгоритма есть 3 параметра: α — коэффициент жадности, p — коэффициент испарения феромона и $T_{\max}$ — время жизни колонии.

Чтобы провести параметризацию, необходимо выбрать некоторое ограниченное количество значений исследуемых параметров, затем решить задачу при этих коэффициентах. Полученное решение сравнивается с эталонным значением, определяемым полным перебором. Необходимо отыскать набор параметров, который будет наиболее близок к эталонному решению. Это делается с помощью построения таблицы.

1.5 Вывод

Для ПО, решающего поставленную задачу следует выделить следующие требования:

- входными данными является ориентированный граф, заданной целочисленной матрицей смежности;
- введенный граф является связным (т.е. на нем разрешима задача коммивояжера);
- ПО имеет подсказки ввода;
- ПО должно обеспечить методы логирования полученных эксперементальных данных;
- ПО выводит минимальную длину пути и сам путь.

В данном разделе были рассмотрены два алгоритма решения задачи коммивояжера: алгоритм полного перебора и муравьиный алгоритм. Для каждого из алгоритмов были выделены основные идеи решения поставленной задачи, а также определена асимптотическая сложность.

2 Конструкторская часть

В данном разделе представлены схемы алгоритмов, описанных в аналитическом разделе.

2.1 Схемы алгоритмов

На рисунке 2.1 приведена схема алгоритма полного перебора.

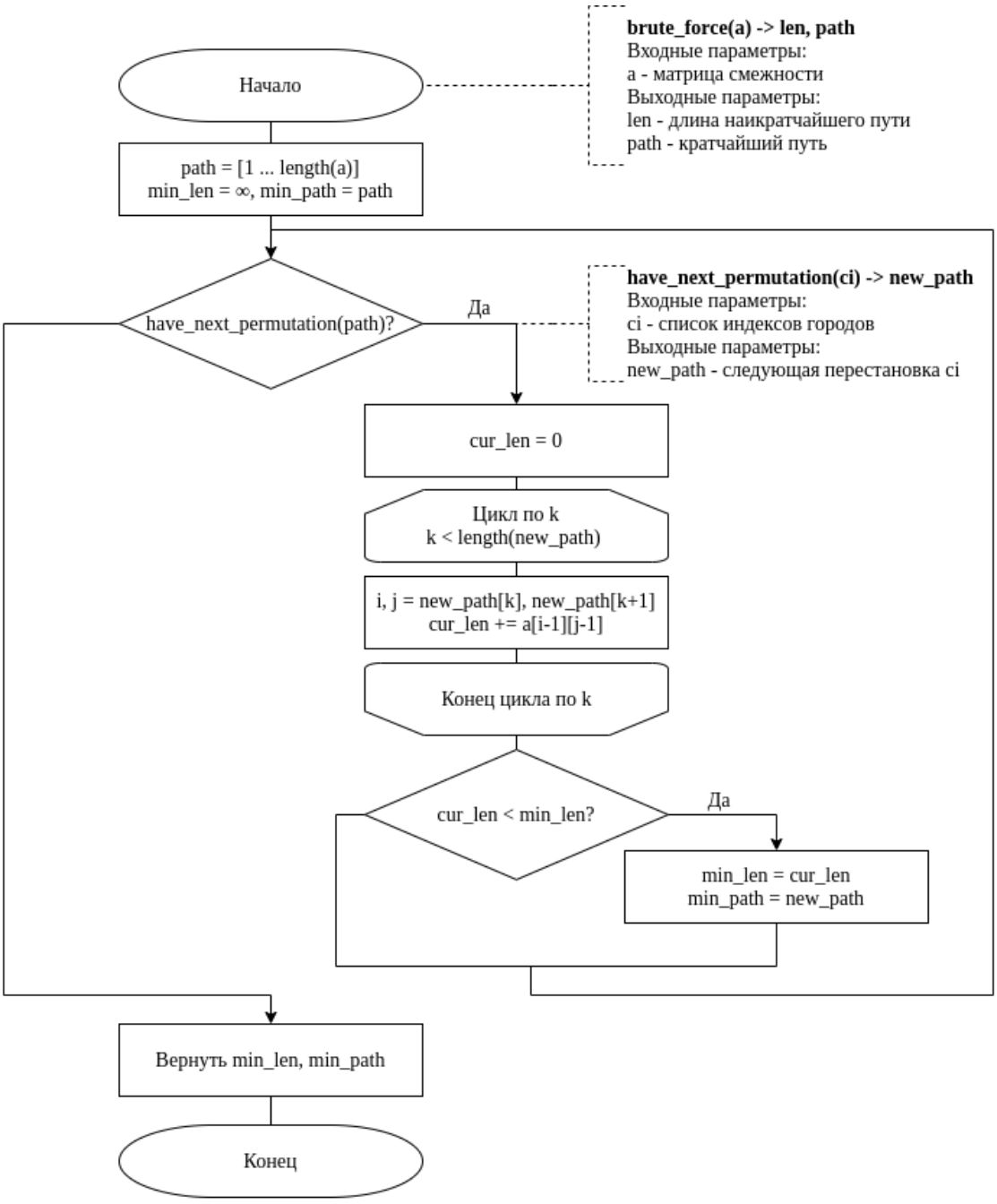

Рисунок 2.1 – Схема алгоритма полного перебора

На рисунке 2.2 приведена схема муравьиного алгоритма.

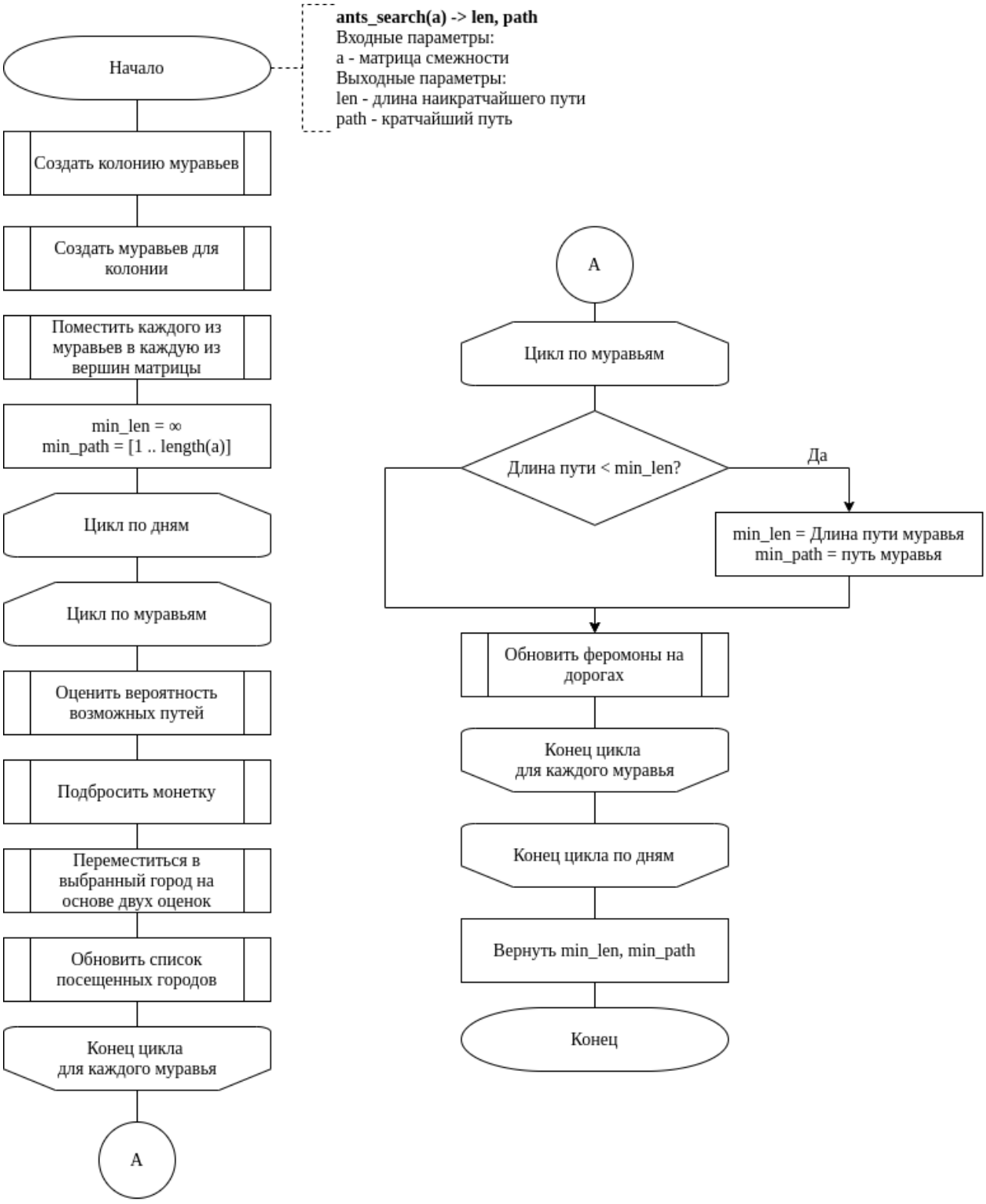

Рисунок 2.2 – Схема муравьиного алгоритма

2.2 Описание структуры ПО

Программа будет будет включать в себя один смысловой модуль называемый *ants*, который будет содержать в себе несколько файлов реализующих процедуры и функции, связанные с реализацией алгоритма полного перебора и муравьиного алгоритма. Независимо от модуля будет существовать файл *main.go*, реализующий взаимодействие между пользователем и модулем, описанным ранее.

2.3 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 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран многопоточный язык GO [2]. Данный выбор обусловлен моим желанием расширить свои знания в области применения данного язкыа. Так же данный язык предоставляет средства тестирования разработанного ПО.

3.2 Листинг кода

В листингах 3.1–?? приведены реализации алгоритмов сортировок, а также вспомогательные функции.

Листинг 3.1 – Основной модуль программы

```
func main() {
    fmt.Println("Ant␣Algorithm␣VS␣Brute␣Force\n")
    var cities int
    fmt.Printf("Cities:")
    fmt.Scanln(&cities)

    citiesMap := ants.GenMap(cities)

    fmt.Printf("The␣Adjacency␣Matrix:\n")
    ants.PrintMap(citiesMap)

    dist, path := ants.AntAlgo(citiesMap, ants.DEFAULT_ALPHA, ants.DEFAULT_BETA,
        ants.DEFAULT_P, ants.DEFAULT_PHEROMONCOEF, ants.DEFAULT_DAYS)
    fmt.Printf("%25s:␣LEN␣=␣%5d,␣PATH␣=␣%v\n", "Ant␣Algorithm", dist, path)

    dist, path = ants.BruteForceAlgo(citiesMap, cities)
    fmt.Printf("%25s:␣LEN␣=␣%5d,␣PATH␣=␣%v\n", "Brute␣Force␣Algorithm", dist, path)

    ants.CompareAlgos()
}
```

Листинг 3.2 – Алгоритм полного перебора - Часть 1

```
func BruteForceAlgo(citiesMap MatrixInt, cities int) (int, []int) {
    path := make([]int, cities)
    for i := 0; i < cities; i++ {
        path[i] = i
    }

    minLen := dist(citiesMap, path)
    minPath := make([]int, cities)

    currPath, state := path, true

    for state {
        currLen := dist(citiesMap, currPath)
        if currLen < minLen {
            minLen = currLen
            copy(minPath, currPath)
        }
        currPath, state = haveNextPermutation(currPath)
    }

    return minLen, minPath
}

func haveNextPermutation(path []int) ([]int, bool) {
    j := len(path) - 2
    for j != -1 && path[j] >= path[j+1] {
        j--
    }
    if j == -1 {
        return path, false
    }

    k := len(path) - 1
    for path[j] >= path[k] {
        k--
    }
    swap(path, j, k)
    l := j + 1
    r := len(path) - 1

    for l < r {
        swap(path, l, r)
        l++
        r--
    }
    return path, true
}
```

Листинг 3.3 – Алгоритм полного перебора - Часть 2

```
func swap(path []int, i int, j int) {
    s := path[i]
    path[i] = path[j]
    path[j] = s
}

func dist(citiesMap MatrixInt, path []int) int {
    dist := 0
    for k := 0; k < (len(path) - 1); k++ {
        dist += citiesMap[path[k]][path[k+1]]
    }
    dist += citiesMap[path[0]][path[len(path)-1]]
    return dist
}
```

Листинг 3.4 – Муравьиный алгоритм - Часть 1

```
func AntAlgo(citiesMap MatrixInt, alpha float64, beta float64, p float64, ph_coef
    float64, days int) (int, []int) {
    params := new(ParamsType)
    params.alpha = alpha
    params.beta = beta
    params.p = p
    params.ph_coef = ph_coef
    params.days = days
    params.q = float64(calcQ(citiesMap))
    colony := createColony(citiesMap, params)
    return live(colony)
}

func calcQ(citiesMap MatrixInt) float64 {
    sum := float64(0)
    for i := 0; i < len(citiesMap); i++ {
        for j := 0; j < len(citiesMap); j++ {
            sum += float64(citiesMap[i][j])
        }
    }
    sum /= float64(len(citiesMap))
    return sum
}
```

Листинг 3.5 – Муравьиный алгоритм - Часть 2

```
func createColony(citiesMap MatrixInt, params *ParamsType) *ColonyType {
    n := len(citiesMap)
    colony := new(ColonyType)
    colony.mtr = citiesMap
    colony.params = *params
    colony.pheromon = make([][]float64, n)
    for i := 0; i < n; i++ {
        colony.pheromon[i] = make([]float64, n)
        for j := 0; j < n; j++ {
            colony.pheromon[i][j] = colony.params.ph_coef
        }
    }
    return colony
}

func live(colony *ColonyType) (int, []int) {
    n := len(colony.mtr)
    min_dist := 0
    min_path := make([]int, n)
    for i := 0; i < colony.params.days; i++ {
        for j := 0; j < n; j++ {
            ant := colony.createAnt(j)
            ant.findPath()
            ant.updatePheromon(colony.params)
            cur_dist := ant.distance()
            if (cur_dist < min_dist) || (min_dist == 0) {
                min_dist = cur_dist
                copy(min_path, ant.path)
            }
        }
    }
    return min_dist, min_path
}

func (ant *AntType) findPath() {
    for {
        p := ant.getProbabilities()
        vertex := chooseVertex(p)
        if vertex != -1 {
            ant.move(vertex)
        } else {
            break
        }
    }
}
```

Листинг 3.6 – Муравьиный алгоритм - Часть 3

```
func (colony *ColonyType) createAnt(vertex int) *AntType {
    n := len(colony.mtr)
    ant := new(AntType)
    ant.colony = colony
    ant.visited = make(MatrixInt, n)
    for i := 0; i < n; i++ {
        ant.visited[i] = make([]int, n)
        for j := 0; j < n; j++ {
            ant.visited[i][j] = colony.mtr[i][j]
        }
    }
    ant.vertex = vertex
    ant.path = make([]int, 0)
    ant.path = append(ant.path, ant.vertex)
    return ant
}

func (ant *AntType) getProbabilities() []float64 {
    n := len(ant.colony.mtr)
    probs := make([]float64, n)
    psum := float64(0)
    var (
        param1 float64
        param2 float64
    )
    for i := 0; i < n; i++ {
        if ant.visited[ant.vertex][i] != 0 {
            param1 = ant.colony.pheromon[ant.vertex][i]
            param2 = (float64(1) /
                float64(ant.colony.mtr[ant.vertex][i]))
            d := math.Pow(param1, ant.colony.params.beta) *
                math.Pow(param2, ant.colony.params.alpha)
            probs[i] = d
            psum += d
        } else {
            probs[i] = 0
        }
    }
    for i := 0; i < n; i++ {
        probs[i] /= psum
    }
    return probs
}
```

Листинг 3.7 – Муравьиный алгоритм - Часть 4

```
func chooseVertex(p []float64) int {
    n := len(p)
    sum := float64(0)
    for i := 0; i < n; i++ {
        sum += p[i]

    }
    randomProbability :=
        rand.New(rand.NewSource(time.Now().UnixNano())).Float64() * sum
    sum = 0
    for i := 0; i < n; i++ {
        if randomProbability < sum+p[i] && randomProbability > sum {
            return i
        }
        sum += p[i]
    }
    return -1
}

func (ant *AntType) move(new_vertex int) {
    n := len(ant.colony.mtr)
    for i := 0; i < n; i++ {
        ant.visited[i][ant.vertex] = 0
    }
    ant.path = append(ant.path, new_vertex)
    ant.vertex = new_vertex
}

func (ant *AntType) distance() int {
    distance := 0
    n := len(ant.path)
    for i := 0; i < n; i++ {
        src := ant.path[i]
        dst := ant.path[(i+1)%len(ant.path)]
        distance += ant.colony.mtr[src][dst]
    }
    return distance
}
```

Листинг 3.8 – Муравьиный алгоритм - Часть 5

```
func (ant *AntType) updatePheromon(params ParamsType) {
    delta := float64(0)
    for i := 0; i < len(ant.colony.pheromon); i++ {
        for j := 0; j < len(ant.colony.pheromon); j++ {
            if ant.colony.mtr[i][j] != 0 {
                f := false
                for k := 0; k < len(ant.path); k++ {
                    src := ant.path[k]
                    dst := ant.path[(k+1)%len(ant.path)]
                    if (src == i && dst == j) || (dst == i && src == j) {
                        f = true
                    }
                }
                if f {
                    delta += ant.colony.params.q /
                        float64(ant.colony.mtr[i][j])
                }

            }
            ant.colony.pheromon[i][j] = (1 - params.p) *
                (float64(ant.colony.pheromon[i][j]) + delta)
            if ant.colony.pheromon[i][j] <= 0 {
                ant.colony.pheromon[i][j] = 0.1
            }
        }

    }
}
```

В таблице 3.1 приведены функциональные тесты для алгоритмов сортировки. Все тесты пройдены успешно (таблица 3.2).

Таблица 3.1 – Ожидаемый результат работы программы

Входная матрица смежности	Полный перебор	Муравьиный алгоритм
$\begin{pmatrix} 0 & 2 & 3 & 26 & 39 \\ 2 & 0 & 29 & 38 & 9 \\ 3 & 29 & 0 & 13 & 34 \\ 26 & 38 & 13 & 0 & 5 \\ 39 & 9 & 34 & 5 & 0 \end{pmatrix}$	32	32
$\begin{pmatrix} 0 & 23 & 49 \\ 23 & 0 & 13 \\ 49 & 13 & 0 \end{pmatrix}$	85	85

Таблица 3.2 – Фактический результат работы программы

Входная матрица смежности	Полный перебор	Муравьиный алгоритм
$\begin{pmatrix} 0 & 2 & 3 & 26 & 39 \\ 2 & 0 & 29 & 38 & 9 \\ 3 & 29 & 0 & 13 & 34 \\ 26 & 38 & 13 & 0 & 5 \\ 39 & 9 & 34 & 5 & 0 \end{pmatrix}$	32	32
$\begin{pmatrix} 0 & 23 & 49 \\ 23 & 0 & 13 \\ 49 & 13 & 0 \end{pmatrix}$	85	85

3.3 Вывод

Были разработаны и протестированы спроектированные алгоритмы: алгоритм полного перебора и муравьиный алгоритм.

4 Исследовательская часть

4.1 Пример работы

Демонстрация работы программы приведена на рисунке 4.1.

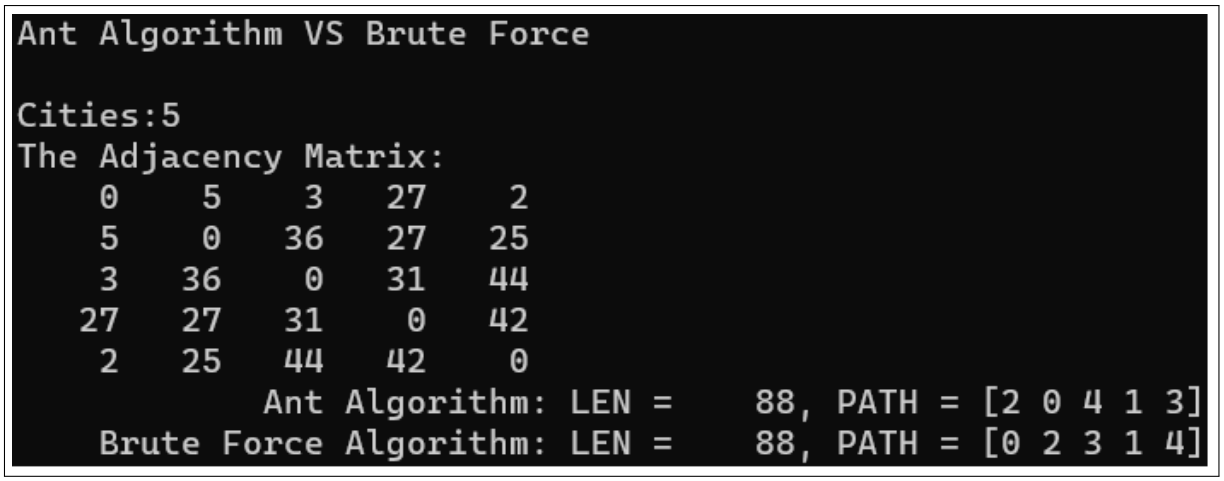

Рисунок 4.1 – Демонстрация работы алгоритмов

4.2 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- Операционная система: Windows 10 64-bit [3].
- Память: 16 GB.
- Процессор: AMD Ryzen 5 4600H [4] @ 3.00 GHz.

Тестирование проводилось на ноутбуке при включённом режиме производительности. Во время тестирования ноутбук был нагружен только системными процессами.

4.3 Время выполнения алгоритмов

Алгоритмы тестировались при помощи написания «бенчмарков» [5], предоставляемых встроенными в Go средствами. Для такого тестирования не нужно самостоятельно указывать количество повторов. В библиотеке для тестирования существует константа N, которая динамически для получения стабильного результата. На рисунке 4.2 приведен график зависимости времени работы алгоритмов для различных наборов данных.

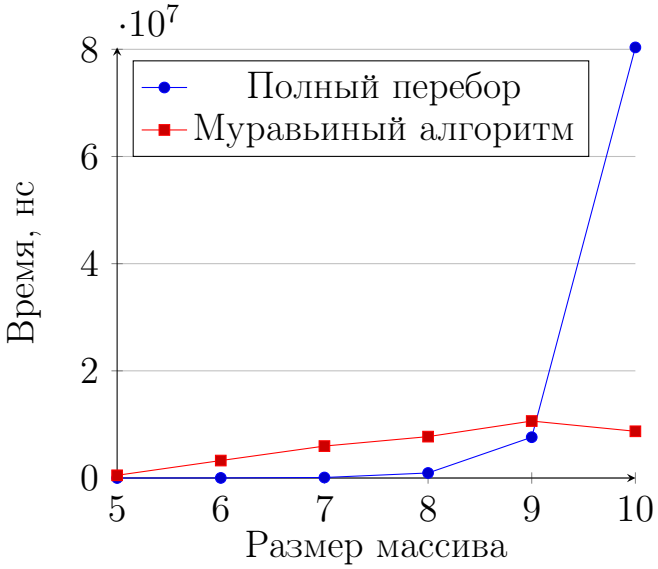

Рисунок 4.2 – Сравнение алгоритмов на случайных матрицах смежности.

4.4 Результаты эксперимента

В таблицах 4.1 - 4.2 представлены результаты эксперимента.

Таблица 4.1 – Результат эксперимента

α	ρ	Кол-во дней	Погрешность
1	0.2	9	18
1	0.2	10	17
1	0.2	11	10
1	0.4	9	17
1	0.4	10	0
1	0.4	11	0

Таблица 4.2 – Результат эксперимента (продолжение)

α	ρ	Кол-во дней	Погрешность
1	0.6	9	18
1	0.6	10	3
1	0.6	11	6
1	0.8	9	18
1	0.8	10	10
1	0.8	11	6
1	1	9	0
1	1	10	0
1	1	11	0
2	0.2	9	3
2	0.2	10	3
2	0.2	11	3
2	0.4	9	6
2	0.4	10	0
2	0.4	11	6
2	0.6	9	6
2	0.6	10	6
2	0.6	11	0
2	0.8	9	0
2	0.8	10	0
2	0.8	11	7
2	1	9	0
2	1	10	3
2	1	11	0
3	0.2	9	0
3	0.2	10	0
3	0.2	11	0
3	0.4	9	6
3	0.4	10	6
3	0.4	11	0
3	0.6	9	3
3	0.6	10	6
3	0.6	11	6
3	0.8	9	3
3	0.8	10	0
3	0.8	11	3
3	1	9	0
3	1	10	0
3	1	11	0

4.5 Вывод

Было произведено сравнение количества затраченного времени полного перебора и муравьиного алгоритма.

По результатам эксперимента можно сделать вывод о том, что при небольших числах алгоритмы работают одинаково. Однако уже на размерности 10×10 алгоритм полного перебора начинает проигрывать, что вполне ожидаемо, так как его алгоритмическая сложность составляет $O(N!)$.

С помощью реализации муравьиного алгоритма удалось добиться как минимум десятикратного выигрыша на небольших размерностях. Чем больше будет размерность матрицы смежности, тем больше будет выигрыш.

Заключение

В ходе выполнения работы были выполнены все поставленные задачи и изучены методы динамического программирования на основе алгоритмов сортировки массивов. Был изучен и реализован эвристический метод решения задачи коммивояжёра — муравьиный алгоритм. Его идея основывается на биологическом принципе поведения муравьёв.

Большинство эвристических методов, к которым относится муравьиный алгоритм, не гарантируют точное решение, но являются достаточными для решения задачи за оптимальное время.

Эвристические методы решают задачу не на универсальном наборе данных, а только на заранее заданном единообразном типе матриц. По этой причине важным этапом реализации эвристических методов является его параметризация — настройка на сооветствующий тип данных.

Параметризация заключается в варьировании параметров, решении задачи с этими параметрами и последующем сравнении полученного значения с эталонным, полученным точным, но долгим методом. В данной лабораторной работе параметризация настраивалась на трёх картах, таким образом суммарная погрешность относительно эталонного решения определялась как максимум из погрешностей для каждой из карт. Выбирая оптимальные настройки, нужно добиться того, чтобы худшая погрешность среди трёх карт была минимальной.

С помощью реализации муравьиного алгоритма удалось добиться как минимум десятикратного выигрыша на небольших размерностях. Чем больше размерность матрицы смежности, тем больше выигрыш.

Список литературы

[1] М.В. Ульянов. Ресурсно-эффективные компьютерные алгоритмы. Разработка и анализ. – М.: ФИЗМАТЛИТ, 2007.

[2] The Go Programming Language [Электронный ресурс]. Режим доступа: `https://golang.org/` (дата обращения: 01.10.2021).

[3] Explore Windows 10. [Электронный ресурс]. Режим доступа: `https://www.microsoft.com/en-us/windows/` (дата обращения: 02.10.2021).

[4] AMD Processors [Электронный ресурс]. Режим доступа: `https://www.amd.com/en/products/apu/amd-ryzen-5-4600h` (дата обращения: 02.10.2021).

[5] Testing – The Go Programming Language [Электронный ресурс]. Режим доступа: `https://golang.org/pkg/testing/` (дата обращения: 08.10.2021).