



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 Богаченко А.Е.

Группа ИУ7-56Б

Оценка (баллы) _____

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

Оглавление

Введение	3
1 Аналитическая часть	4
1.1 Задача коммивояжера	4
1.2 Решение полным перебором	4
1.3 Муравьиные алгоритмы	4
2 Конструкторская часть	7
2.1 Требования к программе	7
2.2 Схемы алгоритмов	7
Вывод	10
3 Технологическая часть	11
3.1 Требования к ПО	11
3.2 Структура ПО	11
3.3 Средства реализации	11
3.4 Листинг кода	12
4 Исследовательская часть	16
4.1 Пример работы	16
4.2 Технические характеристики	16
4.3 Время выполнения алгоритмов	17
Заключение	19
Литература	20

Введение

В современной теории алгоритмов существует ряд задач, получение точного ответа в которых возможно только полным перебором всевозможных вариантов множества решений (т.н. NP-полные задачи). Однако зачастую в подобных задачах оптимизации допустимо получение ответа, приближенного к идеальному. В этих целях используются алгоритмы, работающие за приемлемое полиномиальное время, такие как генетические и муравьиные алгоритмы.

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

В ходе лабораторной работы предстоит:

- рассмотреть муравьиный алгоритм и алгоритм полного перебора в задаче коммивояжера;
- реализовать алгоритмы;
- сравнить временную эффективность алгоритмов.

1 Аналитическая часть

1.1 Задача коммивояжера

Коммивояжёр (фр. *commis voyageur*) — бродячий торговец. Задача коммивояжёра — важная задача транспортной логистики, отрасли, занимающейся планированием транспортных перевозок. Коммивояжёру, чтобы распродать нужные и не очень нужные в хозяйстве товары, следует объехать n пунктов и в конце концов вернуться в исходный пункт. Требуется определить наиболее выгодный маршрут объезда. В качестве меры выгодности маршрута (точнее говоря, невыгодности) может служить суммарное время в пути, суммарная стоимость дороги, или, в простейшем случае, длина маршрута [1].

1.2 Решение полным перебором

Наиболее идейно простым алгоритмом решения задачи коммивояжера [2] является полный перебор решений с выбором кратчайшего из полученных путей. Очевидным недостатком данного алгоритма является необходимость перебора значительного числа комбинаций, которое с ростом числа городов быстро выходит за рамки вычислительных мощностей современных компьютеров. Трудоёмкость алгоритма полного перебора — $O(n!)$.

1.3 Муравьиные алгоритмы

Идея муравьиного алгоритма — моделирование поведения муравьёв, связанное с их способностью быстро находить кратчайшие пути и адаптироваться к изменяющимся условиям, т.е. искать новые кратчайшие пути [3]. При своём движении муравей метит свой путь феромоном, и эта информация используется прочими для выбора пути. Таким образом, более короткие пути будут сильнее обогащаться феромоном и станут более предпочтительны для всей колонии. С помощью моделирования испарение феромона,

т.е. отрицательной обратной связи, гарантируется, что найденное локально оптимальное решение не будет единственным – будут предприняты попытки поиска других путей.

Опишем правила поведения муравья применительно к решению задачи коммивояжера [4]:

- муравьи имеют 'память' - запоминают уже посещенные города, чтобы избегать повторений: обозначим через $J_{i,k}$ список городов, которые необходимо пройти муравью k , находящемуся в городе i ;
- муравьи обладают 'зрением' - видимость есть эвристическое желание посетить город j , если муравей находится в городе i . Уместно считать видимость обратно пропорциональной расстоянию между соответствующими городами $\eta_{i,j} = 1/D_{i,j}$;
- муравьи обладают 'обонянием' – могут улавливать след феромона, подтверждающий желание посетить город j из города i на основании опыта других муравьев. Обозначим количество феромона на ребре (i, j) в момент времени t через $\tau_{i,j}(t)$.

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

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

где $\tau_{i,j}$ – количество феромонов на ребре i до j ;

$\eta_{i,j}$ – эвристическое расстояние от i до j ;

α – параметр влияния феромона;

β – параметр влияния расстояния.

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

$$\Delta\tau_{i,j}^k = \begin{cases} Q/L_k & \text{Если } k\text{-ый муравей прошел по ребру } ij; \\ 0 & \text{Иначе} \end{cases} \quad (1.2)$$

где Q - количество феромона, переносимого муравьём;

Тогда

$$\Delta\tau_{i,j} = \tau_{i,j}^0 + \tau_{i,j}^1 + \dots + \tau_{i,j}^k \quad (1.3)$$

На основе приведённого выше описания алгоритма можно оценить его трудоемкость: $O(t_{max} \cdot m \cdot n^2)$, где t_{max} - число итераций алгоритма ('время жизни колонии'), m - число муравьёв, n - число городов.

Вывод

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

2 Конструкторская часть

2.1 Требования к программе

Требования к вводу:

- У ориентированного графа должно быть хотя бы 2 вершины.

Требования к программе:

- Алгоритм полного перебора должен возвращать кратчайший путь в графе.

.

Входные данные - матрица смежности графа.

Выходные данные - самый выгодный путь.

2.2 Схемы алгоритмов

В данном разделе будут приведены схемы алгоритмов для решения задачи коммивояжера: полный перебор 2.1 и муравьиный 2.2



Рисунок 2.1 – Схема алгоритма полного перебора

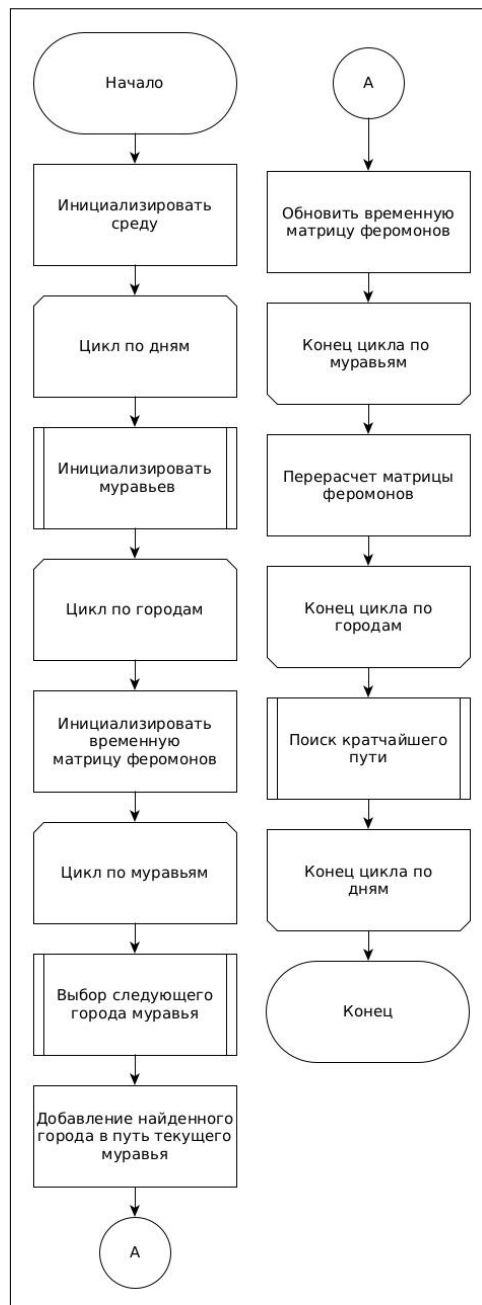


Рисунок 2.2 – Схема муравьиного алгоритма

Вывод

В данном разделе были рассмотрены требования к программе и схемы 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:

- корректная сортировка.

3.2 Структура ПО

В данном разделе будет рассмотрена структура ПО 3.1.

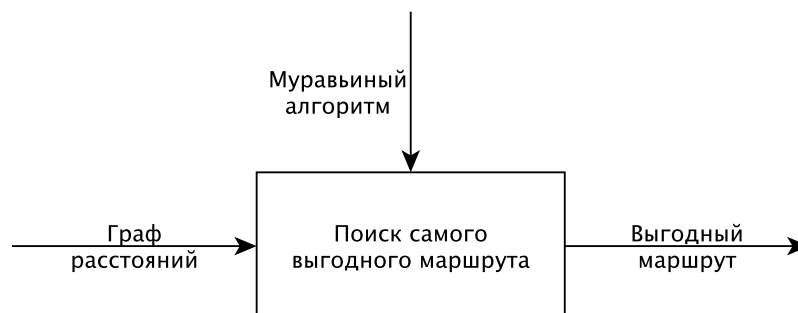


Рисунок 3.1 – Структура ПО

3.3 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран многопоточный язык GO [5]. Данный выбор обусловлен моим желанием расширить свои знания в области применения данного языка. Так же данный язык предоставляет средства тестирования разработанного ПО.

Время работы алгоритмов было замерено с помощью функции `Now()` из библиотеки `Time` [6].

3.4 Листинг кода

В листингах 3.1 – 3.2 приведены реализации алгоритма полного перебора всех решений и муравьиного алгоритма.

Листинг 3.1 – Алгоритм перебора всех возможных вариантов

```
1 // Brute - brute
2 func Brute(f string) []int {
3     weight := getWeights(f)
4     path := make([]int, 0)
5     res := make([]int, len(weight))
6     // for every node
7     for k := 0; k < len(weight); k++ {
8         ways := make([] []int, 0)
9         _ = getRoutes(k, weight, path, &ways)
10        sum := 1000
11        curr := 0
12        ind := 0
13        for i := 0; i < len(ways); i++ {
14            curr = 0
15            for j := 0; j < len(ways[i])-1; j++ {
16                curr += weight[ways[i][j]][ways[i][j+1]]
17            }
18            if curr < sum {
19                sum = curr
20                ind = i
21            }
22        }
23        res[k] = sum
24        ind = 0
25        _ = ind
26    }
27    return res
28 }
29
30 func contains(s []int, e int) bool {
31     for _, a := range s {
32         if a == e {
33             return true
34         }
35     }
36     return false
37 }
38
39 // getRoutes - find all ways
40 func getRoutes(pos int, weight [] []int, path []int, routes *[] []int) []int {
```

```

41 path = append(path, pos)
42 if len(path) < len(weight) {
43     for i := 0; i < len(weight); i++ {
44         if !(contains(path, i)) {
45             _ = getRoutes(i, weight, path, routes)
46         }
47     }
48 } else {
49     *routes = append(*routes, path)
50 }
51 return path
52 }

```

Листинг 3.2 – Муравьиный алгоритм

```

1 // getProbability - probability of path being choosen
2 func (ant *Ant) getProbability() []float64 {
3     p := make([]float64, 0)
4     var sum float64
5     for i, l := range ant.visited[ant.position] {
6         if l != 0 {
7             d := math.Pow((float64(1)/float64(l)), ant.env.alpha) *
9                 math.Pow(ant.env.pheromon[ant.position][i], ant.env.betta)
8             p = append(p, d)
9             sum += d
10        } else {
11            p = append(p, 0)
12        }
13    }
14    for _, l := range p {
15        l = l / sum
16    }
17    return p
18 }
19
20 // pickWay - choose way with probability
21 func pickWay(p []float64) int {
22     var sum float64
23     for _, j := range p {
24         sum += j
25     }
26     r := rand.New(rand.NewSource(time.Now().UnixNano()))
27     rn := r.Float64() * sum
28     sum = 0
29     for i, j := range p {
30         if rn > sum && rn < sum+j {
31             return i
32         }
33         sum += j

```

```

34     }
35     return -1
36 }
37
38 // renewPheromon - renew pheromon after move
39 func (ant *Ant) renewPheromon() {
40     var delta float64
41     delta = 0
42     for k := 0; k < len(ant.env.pheromon); k++ {
43         for i, j := range ant.env.pheromon[k] {
44             if ant.env.weight[k][i] != 0 {
45                 if ant.been[k][i] {
46                     delta = ant.env.q / float64(ant.env.weight[k][i])
47                 } else {
48                     delta = 0
49                 }
50                 ant.env.pheromon[k][i] = (1 - ant.env.p) * (float64(j) + delta)
51             }
52             if ant.env.pheromon[k][i] <= 0 {
53                 ant.env.pheromon[k][i] = 0.1
54             }
55         }
56     }
57 }
58
59 // moveWay - perform move
60 func (ant *Ant) moveWay(path int) {
61     for i := range ant.visited {
62         ant.visited[i][ant.position] = 0
63     }
64     ant.been[ant.position][path] = true
65     ant.position = path
66 }
67
68 // moveAnt - start moving
69 func (ant *Ant) moveAnt() {
70     for {
71         prob := ant.getProbability()
72         way := pickWay(prob)
73         if way == -1 {
74             break
75         }
76         ant.moveWay(way)
77         ant.renewPheromon()
78     }
79 }
80
81 // getDistance - distance travelled

```

```

82 func (ant *Ant) getDistance() int {
83     var distance int
84     for i, j := range ant.been {
85         for k, z := range j {
86             if z {
87                 distance += ant.env.weight[i][k]
88             }
89         }
90     }
91     return distance
92 }
93
94 // StartAnt - ant
95 func StartAnt(env *Env, days int) []int {
96     short := make([]int, len(env.weight))
97     for n := 0; n < days; n++ {
98         for i := 0; i < len(env.weight); i++ {
99             ant := env.newAnt(i)
100             ant.moveAnt()
101             cur := ant.getDistance()
102             if (short[i] == 0) || (cur < short[i]) {
103                 short[i] = cur
104             }
105         }
106     }
107     return short
108 }

```

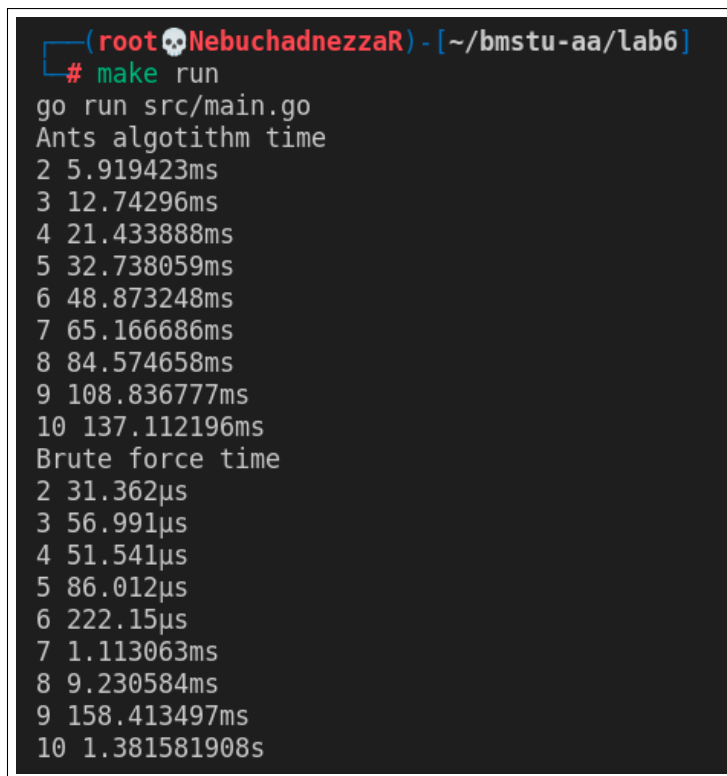
Вывод

В данном разделе была рассмотрена структура ПО и листинги кода программы.

4 Исследовательская часть

4.1 Пример работы

Пример работы программы представлен на рисунке 4.1.



```
(root NebuchadnezzaR) - [~/bmstu-aa/lab6]
# make run
go run src/main.go
Ants algotithm time
2 5.919423ms
3 12.74296ms
4 21.433888ms
5 32.738059ms
6 48.873248ms
7 65.166686ms
8 84.574658ms
9 108.836777ms
10 137.112196ms
Brute force time
2 31.362µs
3 56.991µs
4 51.541µs
5 86.012µs
6 222.15µs
7 1.113063ms
8 9.230584ms
9 158.413497ms
10 1.381581908s
```

Рисунок 4.1 – Демонстрация работы алгоритмов

4.2 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- Операционная система: Kali [7] Linux [8] 5.9.0-kali1-amd64.
- Память: 8 GB.
- Процессор: Intel® Core™ i5-8250U [9] CPU @ 1.60GHz
- Количество логических потоков: 8

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

4.3 Время выполнения алгоритмов

В листинге 4.1 пример реализации функции тестирования.

Листинг 4.1 – Функция тестирования

```
1 // BenchAll - benchmark
2 func BenchAll() {
3     ants := make([]time.Duration, 0)
4     Brutes := make([]time.Duration, 0)
5     for i := 2; i < 11; i++ {
6         _ = os.Remove("data.txt")
7         GenData(i)
8         env := newEnv("data.txt")
9         start := time.Now()
10        shortest := StartAnt(env, 100) //fmt.Print(shortest, " ")
11        end := time.Now()
12        ants = append(ants, end.Sub(start))
13
14        start = time.Now()
15        shortest = Brute("data.txt") //fmt.Print(shortest, "\n")
16        end = time.Now()
17        Brutes = append(Brutes, end.Sub(start))
18        _ = shortest
19    }
20
21    fmt.Println("Ants_algotithm_time")
22    for j := 0; j < len(ants); j++ {
23        fmt.Println(j+2, ants[j])
24    }
25    fmt.Println("Brute_force_time")
26    for j := 0; j < len(ants); j++ {
27        fmt.Println(j+2, Brutes[j])
28    }
29 }
```

На рисунках 4.2 приведён график сравнения производительности конвейеров.

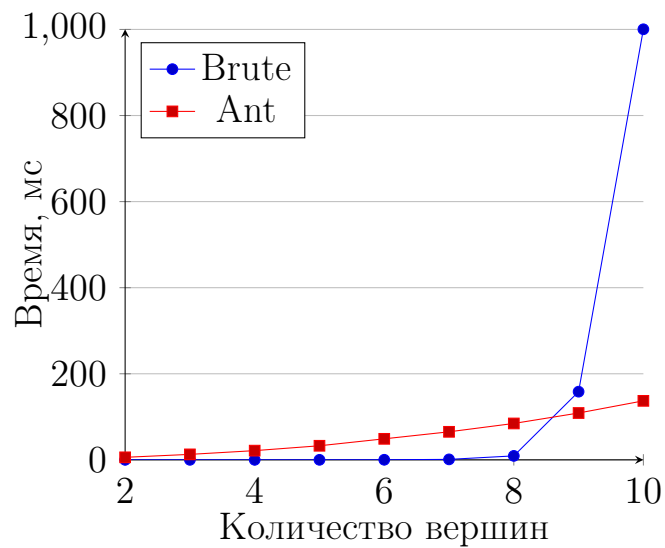


Рисунок 4.2 – Сравнение алгоритмов.

Вывод

Была исследована зависимость времени работы реализованных алгоритмов от размера матрицы смежности графа. По результатам эксперимента на малых размерах графа полный перебор значительно выигрывает муравьиных алгоритм в скорости, однако на размера графа больше 8 сложность полного перебора растет очень быстро, а так как муравьиный алгоритм обладает полиномиальной сложностью, он работает быстрее перебора.

Заключение

В ходе лабораторной работы были изучены и реализованы алгоритмы решения задачи коммивояжера - полный перебор и муравьиный алгоритм.

Было проведено тестирование разработанных алгоритмов, выполнены замеры времени и проведён сравнительный анализ временной эффективности алгоритмов.

Экспериментально были установлены преимущества использования муравьиного алгоритма на графах размерности свыше 8.

Литература

- [1] Товстик Т.М. Жукова Е.В. Алгоритм приближенного решения задачи коммивояжера. 1970.
- [2] Perl problems - Commis Voyageur [Электронный ресурс]. Режим доступа: <http://mech.math.msu.su/~shvetz/54/inf/perl-problems/chCommisVoyageur.xhtml> (дата обращения: 12.12.2020).
- [3] Муравьиные алгоритмы [Электронный ресурс]. Режим доступа: http://www.machinelearning.ru/wiki/index.php?title=%D0%9C%D1%83%D1%80%D0%B0%D0%B2%D1%8C%D0%B8%D0%BD%D1%8B%D0%B5_%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC%D1%8B (дата обращения: 12.12.2020).
- [4] С.Д. Штовба. Муравьиные алгоритмы. 1970.
- [5] The Go Programming Language [Электронный ресурс]. Режим доступа: <https://golang.org/> (дата обращения: 09.09.2020).
- [6] testing – Package time [Электронный ресурс]. Режим доступа: <https://golang.org/pkg/time/> (дата обращения: 18.11.2020).
- [7] Our Most Advanced Penetration Testing Distribution, Ever. [Электронный ресурс]. Режим доступа: <https://kali.org/> (дата обращения: 12.09.2020).
- [8] Linux – Википедия [Электронный ресурс]. Режим доступа: <https://ru.wikipedia.org/wiki/Linux> (дата обращения: 12.09.2020).
- [9] Intel Processors [Электронный ресурс]. Режим доступа: <https://www.intel.com/content/www/us/en/products/processors/core/i5-processors.html> (дата обращения: 12.09.2020).