МГТУ им. Баумана

Лабораторная работа №7

По курсу: "Анализ алгоритмов"

Поиск подстроки в строке

Работу выполнил: Мокеев Даниил, ИУ7-54

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

Москва, 2019

Оглавление

Введение

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

Целью данной лабораторной работы является изучение муравьиных алгоритмов и приобретение навыков параметризации методов на примере муравьиного алгоритма, примененного к задаче коммивояжера.

Задачи данной лабораторной работы:

- рассмотренть муравьиный алгоритм и алгоритм полного перебора в задаче коммивояжера;
- реализовать эти алгоритмы;
- сравнить время работы этих алгоритмов.

1 Аналитическая часть

В данной части будут рассмотрены существующие на данный момент алгоритмические решения проблемы поиска подстроки в строке.

1.1 Общие сведения об алгоритмах поиска подстроки

Поиск подстроки в строке — одна из простейших задач поиска информации. Сферы применения алгоритмов поиска включают в себя:

- Текстовые редакторы;
- СУБД;
- компиляторы;
- программы определения проверки плагиата;
- поисковые системы.

На сегодняшний день существует огромное разнообразие алгоритмов поиска подстроки. Программисту приходится выбирать подходящий в зависимости от таких факторов: длина строки, в которой происходит поиск, необходимость оптимизации, размер алфавита, возможность проиндексировать текст, требуется ли одновременный поиск нескольких строк. В данной лабораторной работе будут рассмотремы два алгоритма сравнения с образцом, алгоритм Кнута-Морриса-Пратта и алгоритм Бойера-Мура.

1.1.1 Стандартный алгоритм

Стандартный алгоритм начинает со сравнения первого символа текста с первым символом подстроки. Если они совпадают, то происходит переход ко второму символу текста и подстроки. При совпадении сравниваются следующие символы. Так продолжается до тех пор, пока не окажется, что подстрока целиком совпала с отрезком текста, или пока не встретятся несовпадающие символы. В первом случае задача решена, во втором мы сдвигаем указатель текущего положения в тексте на один символ и заново начинаем сравнение с подстрокой[2].

1.1.2 Алгоритм Бойера-Мура

Алгоритм Бойера-Мура осуществляет сравнение с образцом справа налево, а не слева направо. Исследуя искомый образец, можно осуществлять более эффективные прыжки в тексте при обнаружении несовпадения. В этом алгоритме кроме таблицы

суффиксов применяется таблица стоп-символов. Она заполняется для каждого сивола в алфавите. Для каждого встречающегося в подстроке символа таблица заполняется по принципу максимальной позиции символа в строке, за исключением последнего символа. При определении сдвига при очередном несовпадении строк, выбирается максимальное значение из таблицы суффиксов и стоп-символов[2].

Таблица 2. Пошаговая работа алгоритма Бойера-Мура.

a	b	a	b	a	c	a	b	a	a
a	b	a	a						
		a	b	a	a				
						a	b	a	a

1.1.3 Алгоритм Кнута-Морриса-Пратта

Алгоритм Кнута-Морриса-Пратта основан на принципе конечного автомата, однако он использует более простой метод обработки неподходящих символов. В этом алгоритме состояния помечаются символами, совпадение с которыми должно в данный момент произойти. Из каждого состояния имеется два перехода: один соответствует успешному сравнению, другой - несовпадению. Успешное сравнение переводит нас в следующий узел автомата, а в случае несовпадения мы попадаем в предыдущий узел, отвечающий образцу. В программной реализации этого алгоритма применяется масив сдвигов, который создается для каждой подстроки, которая ищется в тексте. Для каждого символа из подстроки рассчитывается значение, равное максимальной длине совпадающего префикса и суффикса отсительно конкретного элемента подстроки. Создание этого массива позволяет при несовпадении строки сдвигать ее на расстояние, большее, чем 1 (в отличие от стандартного алгоритма).

Таблица 1. Пошаговая работа алгоритма Кнута-Морриса-Пратта.

a	b	a	b	a	c	a	b	a	a
a	b	a	a						
		a	b	a	a				
				a	b	a	a		
					a	b	a	a	
						a	b	a	a

Вывод

В данном разделе были рассмотрены алгоритмы для решения задачи поиска подстроки в строке.

2 Конструкторская часть

В данном разделе будут рассмотрены основные требования к программе и схемы алгоритмов.

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 комивояжора: полный пребор(Рис.??) и муравьиный (Рис. ??)

Вывод

В данном разделе были рассмотрены требования к программе и схемы алгоритмов.

3 Технологическая часть

3.1 Выбор ЯП

В качестве языка программирования был выбран golang. Время работы алгоритмов было замерено с помощью time.

3.2 Листинг кода алгоритмов

В данном разделе будут приведены листинги кода полного перебора всех решений (Листинг 3.1) и реализации муравьиного алгоритма (Листинг 3.2)

Листинг 3.1: Перебор всех возможных вариантов

```
1
2 func brute(file_name string) []int{
3   weight := get_weights(file_name)
4   path := make([]int, 0)
5   res := make([]int, len(weight))
6
7   for k:=0; k<len(weight);k++{
8     ways := make([][]int, 0)
9     _ = go_route(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      }
21    }
22    res[k] = sum
23  }
24  return res
25 }
26
27 func contains(s []int, e int) bool {
```

```
28    for _, a := range s {
29      if a == e {
30        return true
31      }
32    }
33    return false
34  }
35
36
37  func go_route(pos int, weight [][]int, path []int, routes *[][]int)[]int
        {
38    path = append(path, pos)
39    if len(path) < len(weight){
40      for i:=0; i < len(weight); i++{
41        if !(contains(path, i)){
42          _ = go_route(i, weight,path, routes)
43        }
44      }
45      }else{
46        *routes = append(*routes, path)
47    }
48    return path
49  }
```

Листинг 3.2: Муравьиный алгоритм

```
1
2  func start (env *env, days int) []int{
3    shortest_dist := make([]int, len(env.weight))
4    for n := 0; n < days; n++{
5      for i:= 0; i< len(env.weight); i++{
6        ant := env.new_ant(i)
7        ant.ant_go()
8
9        cur_dist := ant.get_distance()
10       if (shortest_dist[i] == 0) || (cur_dist < shortest_dist[i]){
11         shortest_dist[i] = cur_dist
12       }
13     }
14   }
15   return shortest_dist
16 }
17
18 func (ant *ant) ant_go(){
```

```
19    for{
20      prob := ant.count_probapility()
21      choosen_path := choose_path(prob)
22      if choosen_path == -1{
23        break}
24      ant.go_path(choosen_path)
25      ant.renew_pheromon()
26    }
27  }
28
29  func (ant *ant)count_probapility() []float64{
30    res := make([]float64, 0);
31    var d float64;
32    var sum float64;
33    for i, lenght := range ant.visited[ant.position]{
34      if lenght != 0{
35        d = math.Pow((float64(1)/float64(lenght)), ant.env.alpha) * math.
              Pow(ant.env.pheromon[ant.position][i], ant.env.betta)
36        res = append(res, d)
37        sum += d
38      } else{
39        res = append(res, 0)
40      }
41      }
42      for _, lenght := range res{
43        lenght = lenght / sum
44    }
45    return res
46  }
47
48  func choose_path(probab []float64) int{
49    var sum float64
50    for _, j := range probab{
51      sum += j
52    }
53    r := rand.New(rand.NewSource(time.Now().UnixNano()))
54    random_fl := r.Float64() * sum
55    sum = 0
56    for i , j := range probab{
57      if random_fl > sum && random_fl<sum+j{
58        return i
59      } else{
60      sum+=j
```

```
61      }
62    }
63    return -1
64 }
```

Вывод

В данном разделе были рассмотрены основные сведения о модулях программы и листинг кода алгоритмов.

4 Исследовательская часть

В даннном разделе будет проведен сравнительный временной анализ алгоритмов и рассмотрена параметризация муравьиного алгоритма. Замеры времени были произведены на: Intel Core i5-6200U.

5 Исследование зависимости времени работы алгоритмов от размера графа

В данном разделе будет приведены результаты сравнения времени работы реализованных алгоритмов в зависимоти от размера матрицы смежности (Рис. 5.1). Время измерено в миллисекундах.

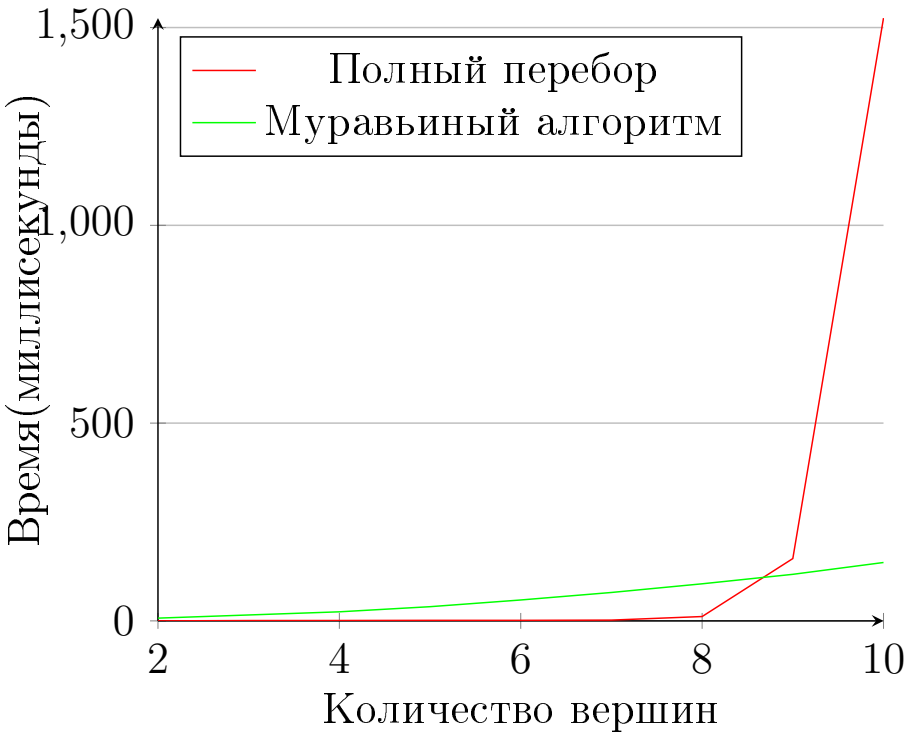

Рис. 5.1: Сравнение параллельного и обычного алгоритмов

6 Выводы исследовательского раздела

Была исследована зависимоть 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 полиноминальной сложностью, он работает быстрее перебора.

Заключение

В ходе лабораторной работы я изучила возможности применения и реализовала алгоритм полного перебора и муравьиный алгоритм.

Временной анализ показал, что неэффективно использовать полный перебор на графе размера больше 8.

Литература

[1] Окулов С. М. Алгоритмы обработки строк. — М.: Бином, 2013. — 255 с.

[2] Дж. Макконнелл. Анализ лгоритмов. Активный обучающий подход

[3] Основные сведения о модульных тестах [Электронный ресурс], - режим доступа: https://docs.microsoft.com/ru-ru/visualstudio/test/unit-test-basics?view=vs-2019