МГТУ им. Баумана

Лабораторная работа №7

По курсу: "Анализ алгоритмов"

Алгоритмы поиска подстроки в строке

Работу выполнил: Рязанов М. С., ИУ7-52Б

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

Москва, 2019

Оглавление

Введение

Поиск подстроки в строке - важная задача поиска информации. Применяется в виде встроенной функции в текстовых редакторах, СУБД, поисковых машинах и языках программирования, задачах биоинформатики[2].

Такой поиск приходится проводить довольно часто, поэтому необходимо, чтобы он осуществлялся как можно быстрее. Становится ясно, что наивный алгоритм с полным перебором всех частей строки, длины которых соответсвуют длине шаблона, является не самым эффективным способом решения такой задачи.

Существует немалое количество алгоритмов, справляющихся с поиском подстроки, лучше, чем наивный перебор.

Целью данной лабораторной работы является изучение алгоритмов поиска подстроки в строке. Для достижения поставленной цели необходимо решить следующие задачи:

- изучить алгоритмы поиска подстроки в строке Кнута-Морриса-Пратта и Бойера-Мура;
- реализовать указанные алгоритмы;
- провести тестирования корректности работы реализаций;
- провести сравнительный анализ алгоритмов по времени выполнения;
- описать и обосновать полученные результаты.

1 | Аналитическая часть

1.1 Описание алгоритмов

1.1.1 Алгоритм Кнута-Морриса-Пратта

Алгоритм Кнутта-Морриса-Пратта основан на принципе конечного автомата. 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 приводт нас в следующее состояние автомата, а в случае несовпадения мы попадаем в предыдущий узел, отвечающий образцу. Пример автомата Кнута-Морриса-Пратта для подстроки *ababcb* приведен на 1.1. [3]

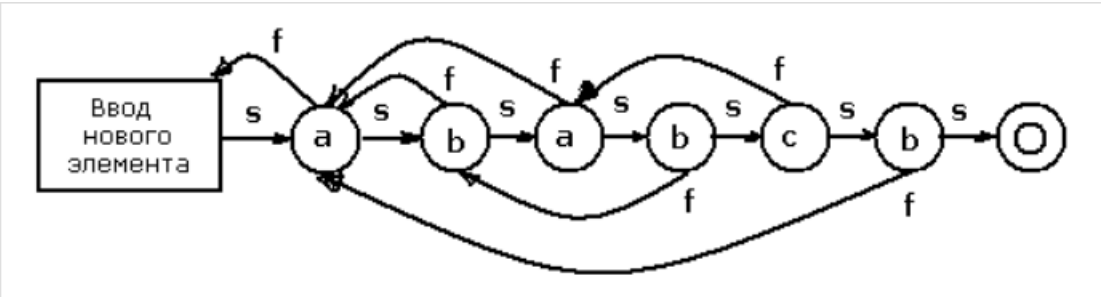

Рис. 1.1: Пример полного автомата Кнута-Морриса-Пратта

приведены примеры работы алгоритмов.

Для построения конечного автомата испольуется префикс-функция, т.е. массив чисел $\pi[0 \dots n-1]$, где $\pi[i]$ определяется следующим образом: это такая наибольшая длина наибольшего собственного суффикса подстроки s[0 . . . i], совпадающего с её префиксом (собственный суффикс — значит не совпадающий со всей строкой). В частности, значение $\pi[0]$

полагается равным нулю.

Математически определение префикс функции записывается следующим образом: $\pi[i] = max_k$

Таким образом для нахождения вхождения строки s в строку t можно построить префикс-функцию для строки $s + +t$, где - разделитель, который не должен встречаться в строках s и t. К тому же возможно оптимизировать данный алгоритм, храня значения префикс-функции только для строки s и для последнего расмотренного символа строки t.

1.1.2 Алгоритм Бойера-Мура

Алгоритм Бойера-Мура осуществляет сравнение с образцом справа налево, а не слева направо, как алгоритм Кнута-Морриса-Пратта. Исследуя искомый образец, можно осуществлять более эффективные прыжки в тексте при обнаружении несовпадения. [3]

Рассмотрим строку there they are и подстроку they. Здесь сначала сравниваются y и r и обнаруживается несовпадение. Поскольку известно, что буква r вообще не входит в образец, можно сдвинуться в тексте на целых четыре буквы(т.е. на длину образца) вправо. Затем сравнивается буква y с h и вновь обнаруживается несовпадение. Однако поскольку на этот раз p входит в образец, есть возможность сдвинуться вправо только на две буквы так, чтобы буквы h совпали. Затем начинается сравнение справа и обнаруживается полное совпадение кусочка текста с образцом. В алгоритме Бойера-Мура делается 6 сравнений вместо 13 сравнений в наивном алгоритме, в котором рассматриваются все подстроки исходной строки, совпадающие по длине с шаблоном. Пример проиллюстрирован в таблице 1.1.

	t	h	e	r	e		t	h	e	y		a	r	e
1	t	h	e	y										
2					t	h	e	y						
3							t	h	e	y				

Таблица 1.1: Поиск образца they в тексте there they are

Выводы:

- Расмотрены алгоритмы поиска подстроки в строке, представляющие более эффективное решение в сравнении с наивным алгоритмом;
- выделены ключевые моменты рассмотренных алгоритмов.

2 | Конструкторская часть

2.1 Разработка алгоритмов

На рисунках 2.1, 2.1, 2.1 представлены схемы разработанных алгоритмов.

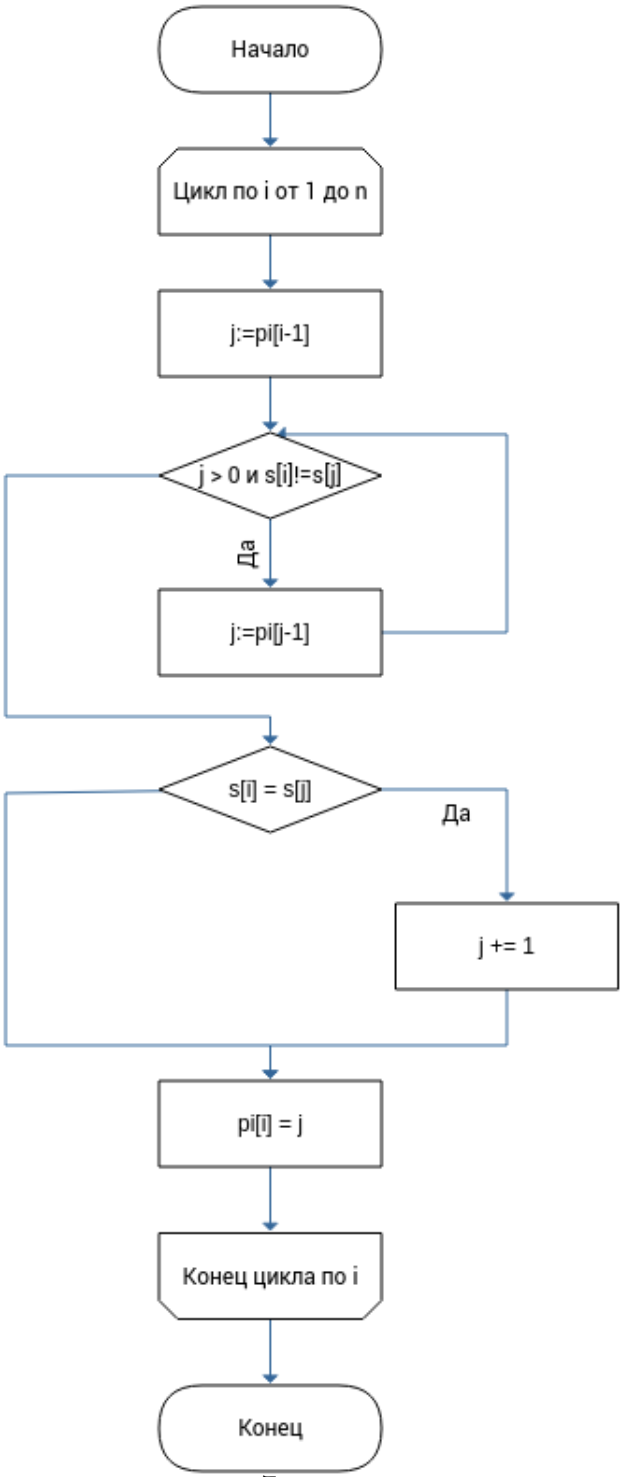

Рис. 2.1: Схема алгоритма подсчета префикс функции

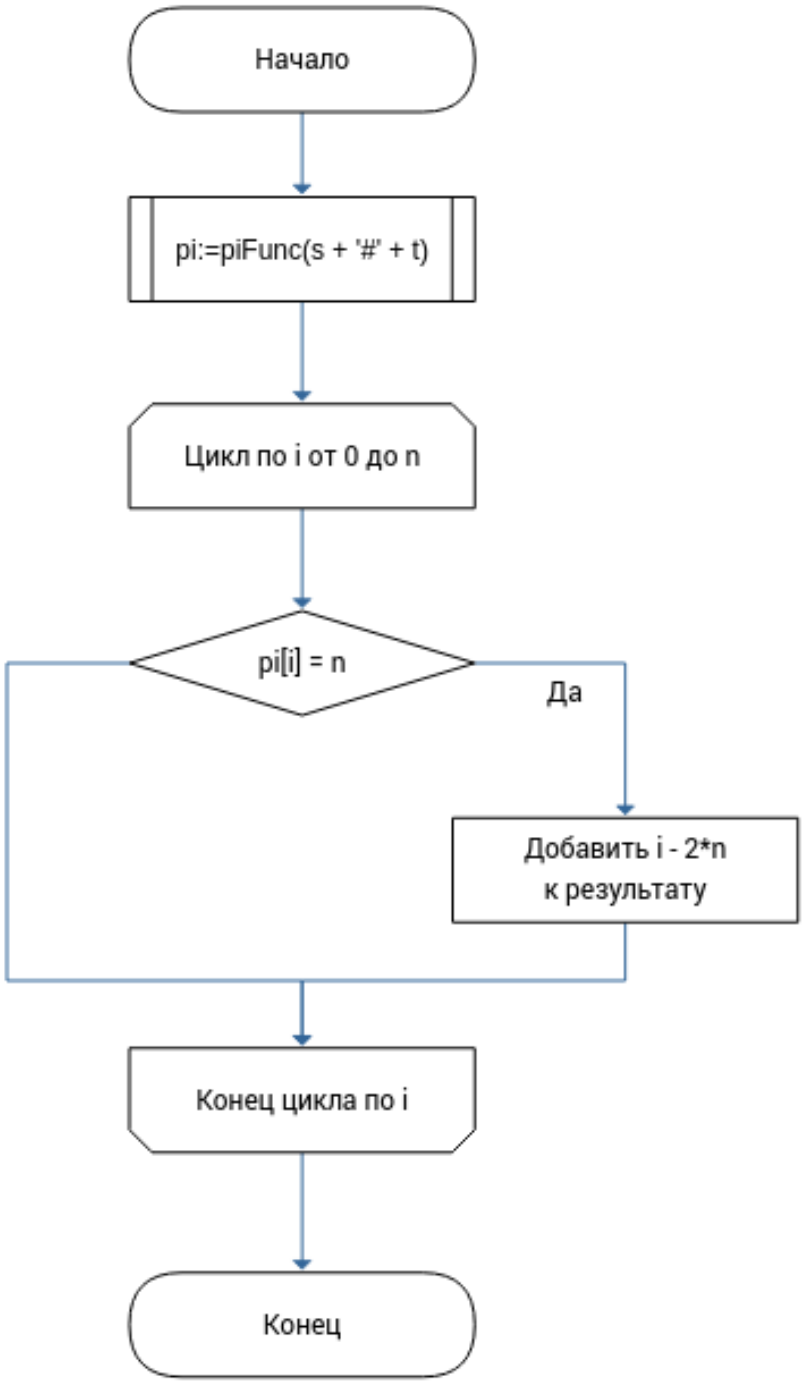

Рис. 2.2: Схема алгоритма Кнута-Морриса-Пратта

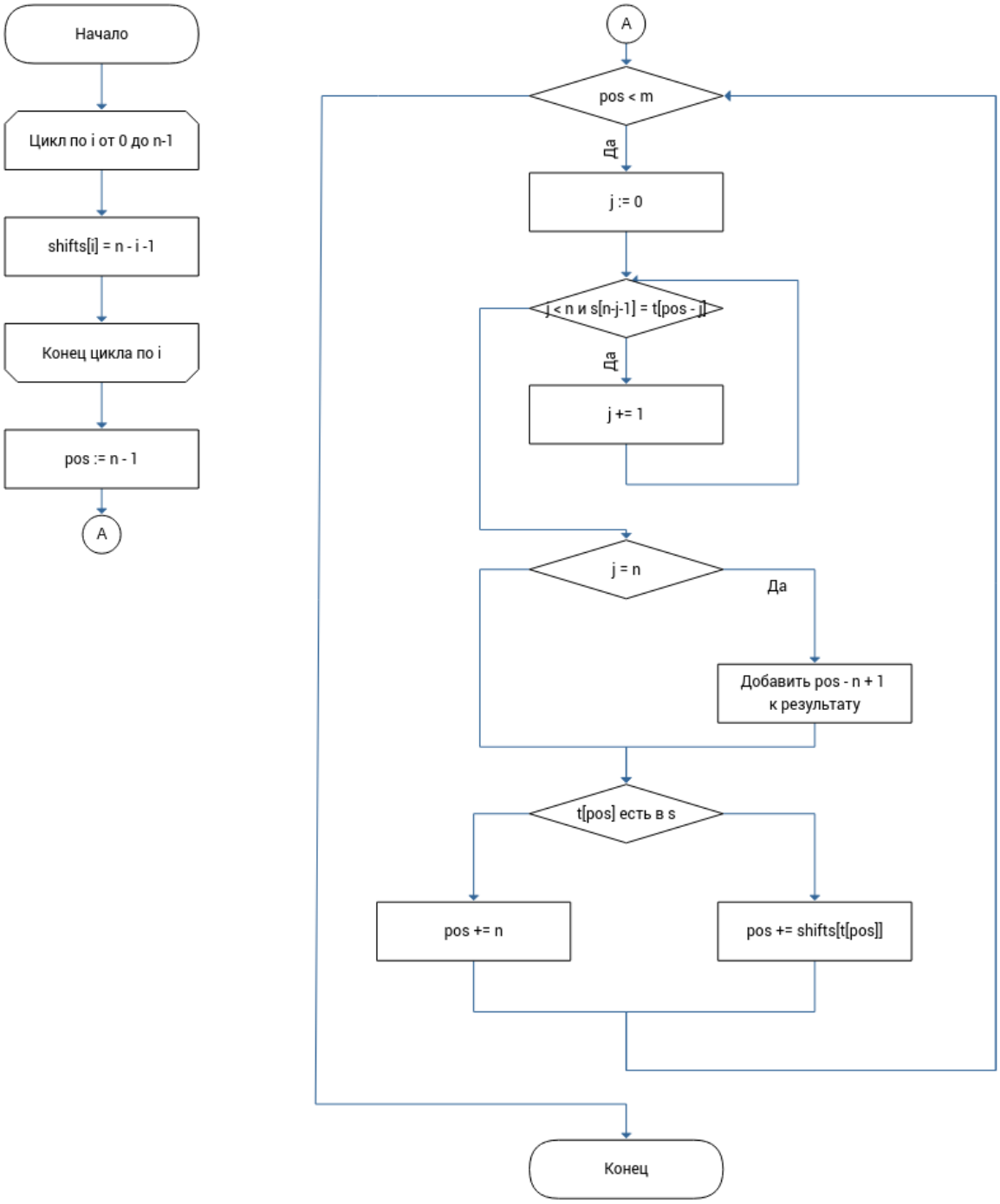

Рис. 2.3: Схема алгоритма Бойера-Мура

2.2 Сравнительный анализ алгоритмов

Рассмотрим пример работы алгоритмов на строке *ababqcababababakababacqw* длиной 22 символа и подстроке *ababac* длиной 6 символов.

2.2.1 Пример работы алгоритма Кнута-Морриса-Пратта

Сперва необходимо построить префикс-функцию для конкатенации образца и строки, в которой осуществляется поиск, через разделить, который не встречается ни в одной из строк. Результирующая префикс-функция приведена в таблице 2.1. В приведенной таблице *курсивом* обазначены совпадающие префиксы, а **жирным** совпадающие суффиксы.

Индекс	Префикс	Значение функции
0	a	0
1	ab	0
2	*a*b**a**	1
3	*ab***ab**	2
4	ababac	0
5	ababac#	0
6	*a*babac#**a**	1
..	....	...
25	*ababa*c#ababqcababababak**ababa**	5
26	*ababac*#ababqcababababak**ababac**	6

Таблица 2.1: Пример работы алгоритма Кнута-Морриса-Пратта

Затем осуществляется поиск позиций в которых значение i префикс-функции равно длине образца, тогда индекс начала вхождения образца будет равен $i - 2 * len(s)$, где $len(s)$ - длина строки образца.

Для приведенного примера это 13.

2.2.2 Пример работы оптимизированного алгоритма Кнута-Морриса-Пратта

В случае оптимизированного алгоритма префикс-фукнция строится только для образца. А затем при прохождении по строке, в которой осуществляется поиск, применяется похожий подход, за тем исключением, что ис-

пользуется только значение префикс-функции для последнего символа и значения вычисленные для образца. Если значение префикс-функции для i-го символа строки равно длине образца, то к массиву позиций вхождений добавляется значение $i - len(s) + 1$, где $len(s)$ - длина образца.

2.2.3 Пример работы алгоритма Бойера-Мура

В алгоритме Бойера-Мура необходимо предварительно найти массив сдвигов. Длина массива сдвигов равна мощности алфавита. Значения массива сдвигов приведены в таблице 2.2

Символ	a	b	a	b	a	c
Смещение	1	2	1	2	1	6

Таблица 2.2: Таблица смещений

Значения смещений для символов, неуказанных в таблице, равны длине строки-образца.

В таблице 2.2

	a	b	a	b	q	c	a	b	a	b	a	b	a	k	a	b	a	b	a	c	q	w
1	a	b	a	b	a	c																
2							a	b	a	b	a	c										
3									a	b	a	b	a	c								
4															a	b	a	b	a	c		

Таблица 2.3: Пример работы алгоритма Бойера-Мура

Выводы:

- Разработаны алгоритмы нахождения подстроки;
- приведены примеры работы алгоритмов.

3 | Технологическая часть

3.1 Выбор ЯП

В качестве языка программирования был выбран go[1] т.к он отличается производительностью и простотой, что позволит произвести наиболее точные измерения скорости работы и сократить время на реализацию алгоритмов. К тому же имеет богатую стандартную библиотеку для работы со строками, в частности модуль strings[].

В листингах 3.1, 3.2, 3.3, 3.4 приведены реализации алгоритмов поиска подстроки в строке

Листинг 3.1: Реализация алгоритма Кнута-Морриса-Пратта

```
1  func Kmp(s, t string) []int {
2    if len(s) == 0 {
3      return []int{}
4    }
5
6      // Нахождение префиксной функции
7    pi := piFunc(s + "#" + t)
8
9    res := make([]int, 0)
10   // Добавление к результату позиций, в которых значение
11   // префиксной функции равно длине образца
12   lens := utf8.RuneCountInString(s)
13   for i, v := range pi {
14     if v == lens {
15       res = append(res, i-2*lens)
16     }
17   }
18   return res
```

```
}
```

Листинг 3.2: Оптимизированная реализация алгоритма Кнута-Морриса-Пратта

```
func KmpOnline(s string, t *strings.Reader) []int {
  if len(s) == 0 {
    return []int{}
  }

    // Вычисление префиксной функции для образца
  s = s + "#"
  pi := piFunc(s)
  sInRune := []rune(s)
  lens := len(sInRune)

  res := make([]int, 0)

  lastIdx := 0
  count := 0

  // Проход по строке и вычисление значения префиксной функции в
   каждой позиции
  for t.Len() > 0 {
    r, _, _ := t.ReadRune()

    j := lastIdx
    for j > 0 && sInRune[j] != r {
      j = pi[j-1]
    }
    if sInRune[j] == r {
      j++
    }
    lastIdx = j

    if j == lens-1 {
      res = append(res, count-lens+2)
    }
    count++
  }
```

```
35    return res
36 }
```

Листинг 3.3: Функция подсчета префиксной функции

```
1  func piFunc(s string) []int {
2    slnRune := []rune(s)
3
4    pi := make([]int, len(slnRune))
5    for i := 1; i < len(slnRune); i++ {
6      j := pi[i-1]
7      for j > 0 && slnRune[i] != slnRune[j] {
8        j = pi[j-1]
9      }
10     if slnRune[i] == slnRune[j] {
11       j++
12     }
13     pi[i] = j
14   }
15   return pi
16 }
```

Листинг 3.4: Реализация алгоритма Бойера-Мура

```
1  func BoyerMur(s, t string) []int {
2    if len(s) == 0 {
3      return []int{}
4    }
5
6    slnRune := []rune(s)
7    tlnRune := []rune(t)
8    shifts := make(map[rune]int)
9
10     // Подсчет сдвигов
11   for i := 0; i < len(slnRune)-1; i++ {
12     shifts[slnRune[i]] = len(slnRune) - i - 1
13   }
14
15   res := make([]int, 0)
```

```
16    pos := len(slnRune) - 1
17
18      // Проход по строке
19    for pos < len(tlnRune) {
20      j := 0
21      // Сравнение образца и строки
22      for ; j < len(slnRune) && slnRune[len(slnRune)-j-1] ==
       tlnRune[pos-j]; j++ {
23      }
24      // Добавление в результат, если совпали
25      if j == len(slnRune) {
26        res = append(res, pos-len(slnRune)+1)
27      }
28      // мещениеC позиции
29      if sh, exist := shifts[tlnRune[pos]]; exist {
30        pos += sh
31      } else {
32        pos += len(slnRune)
33      }
34    }
35
36    return res
37  }
```

3.2 Тестовые случаи

Тестирование было организовано с помощью пакета **testing**.

Каждая из реализованных программ была протестирована на следующих случаях:

- Образец - пустая строка
- Строка в которой организуется поиск - пустая
- Строка образец имеет большую длину чем строка, в которой организуется поиск
- Строки состоят из кириллических символов

- Произвольные строки с несколькими вхождениями образца в строку.

Анализ покрытия тестами, проведенный с помощью утилиты go test -cover показал, что покрытие реализаций алгоритмов тестами составляет 100%.

Выводы:

- В качестве языка программирования выбран Golang, так как предоставляет удобный инструментарий, необходимый в данной задаче
- Алгоритмы реализованы в виде подпрограмм, код которых приведен в листингах : 3.1, 3.2, 3.3, 3.4
- Разработы тестовые случаи и проведено тестирование

4 | Исследовательская часть

Замеры времени работы реализаций были проведены на 3 типах входных данных:

В первом испытании использовалась строка образец длиной 100, состоящая только из букв 'a', строка в которой происходит поиск состоит из повторяющихся букв 'ab'. Проводились замеры времени работы относительно длины строки, в которой осуществлялся поиск.

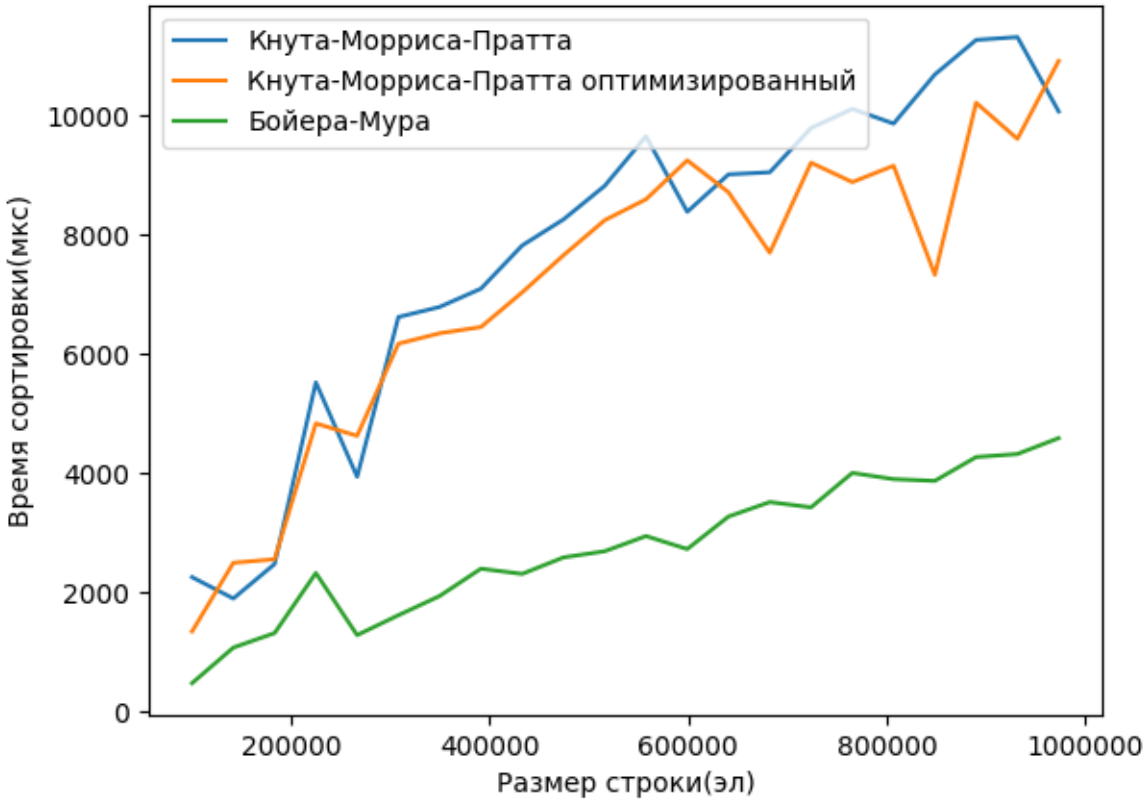

Рис. 4.1: Зависимость времени работы реализаций

Как видно из графика, поиск вхождения подстроки алгоритмов Бойера-Мура выполняется быстрее на этих данных. Это связано с тем, что наличие в строке символа, отсутствующего в образце, позволяет алгоритму сразу сместиться на длину образцу, тем самым сократив число необходимых сравнений. Оптимизированный алгоритм Кнута-Морриса-Пратта работает немного быстрее обычно на больших размерах строк.

В втором испытании использовалась строка образец длиной 100, вида 'baa...', строка в которой осуществлялся поиск состоит только из букв 'a'.

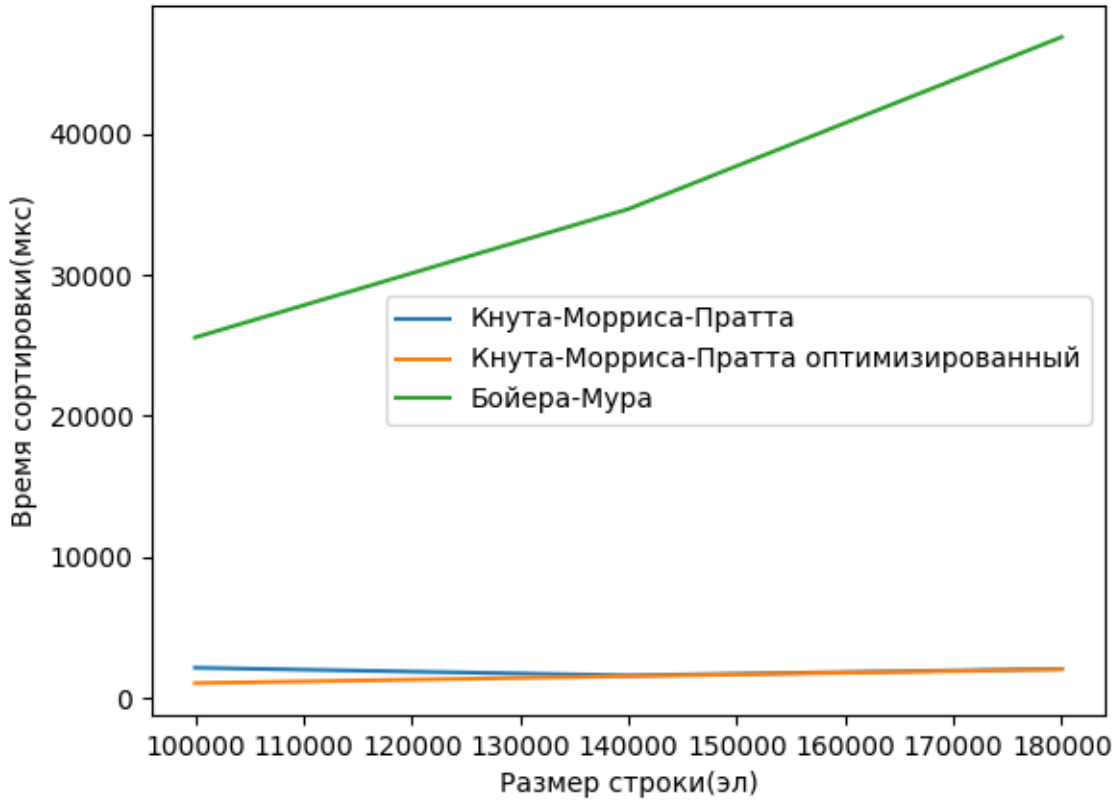

Рис. 4.2: Зависимость времени работы реализаций

Как видно из графика алгоритм Бойера-Мура работает значительнее медленее, обе версии алгоритма Кнута-Морриса-Пратта дают одинаковый результат. Медленная работа алгоритма Бойера-Мура обусловлена тем, что происходит полное сравнение строки образца с строкой, в которой осуществляется поиск, а смещение происходит только на 1 символ. На данных такого типа алгоритм Бойера-Мура имеет трудоемкость $O(n * m)$, где длинны образца и строки поиска. В контексте данного эксперимента трудоемкость линейно зависит от длины строки, в которой происходит поиск, но с большей константой, что обуславливает большее время выполнения.

В третьем испытании использовалась строка образец длиной 100, вида 'abab..', строка в которой осуществлялся поиск состоит только из букв

’a’.

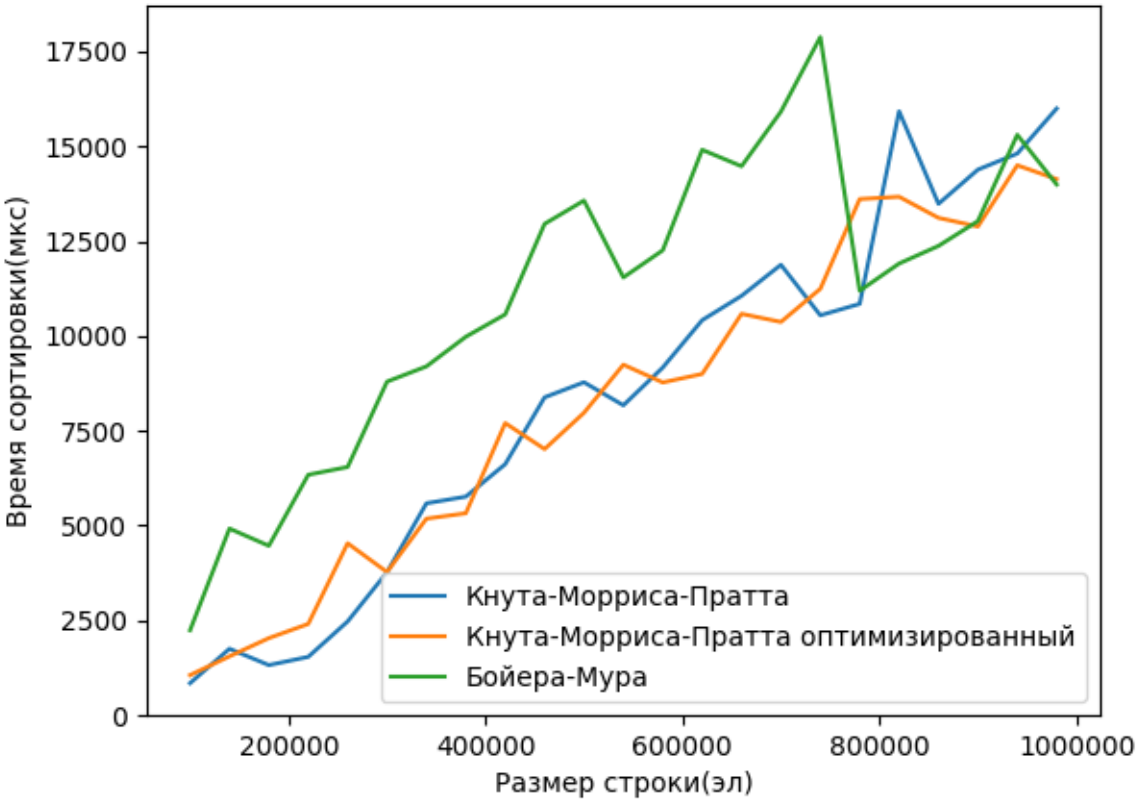

Рис. 4.3: Зависимость времени работы реализаций

Как видно из графика зависимости времени выполнения от длины строки, в которой осуществляется поиск, линейны. Время работы алгоритма Бойера-Мура в среднем на 2500мкс дольше времени работы обеих реализаций алгоритма Кнута-Морриса-Пратта.

Выводы:

- Были проведены замеры времени работы алгоритмов в лучшем, худшем и среднем случаях;
- обе реализации алгоритма Кнута-Морриса-Пратта имеют одинаковое время выполнения, что позволяет использовать оптимизированную версию для сокращения использования памяти;
- алгоритм Бойера-Мура выгоднее использовать в случаях, когда строка в которой осуществляется поиск содержит символы не встречающиеся в образце.

Заключение

Был изучены и реализованы алгоритмы поиска подстроки в строке. Для реализации алгоритмов был выбран язык Golang, так сочетает быстродействие и удобство разработки, имеет библиотеку для удобной работы со строками, предоставляет инструменты для тестирования программ и для анализа пократия тестами.

Экспериментально было подтверждено различие во временной эффективности алгоритмов на разных типах данных, на основе анализа времени работы 3 реализаций на различных строках образах и строках поиска.

В результате исследований пришел к выводу, что использование алгоритма Бойера-Мура оправдано, если алфавит текста больше алфавита строки-образца, например поиск слова в естественном тексте. Алгорит Кнута-Морриса-Пратта будет эффективнее если, образец и текст состоят из одних и тех же символов в различном порядке, например цепочки ДНК.

Список литературы

[1] *Golang documentation*. URL: `https://godoc.org/`. (accessed: 04.12.2019).

[2] Гасфилд. *Строки, деревья и последовательности в алгоритмах.* Информатика и вычислительная биология. Невский Диалект БВХ-Петербург, 2003.

[3] Дж. Макконнелл. *Анализ алгоритмов.Активный обучающий подход.* М.:Техносфера, 2009.