МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №5
по дисциплине «Построение и анализ алгоритмов»
Тема: Ахо-Корасик
Вариант 6: подготовка к распараллеливанию

Студент гр. 1303 _____ Беззубов Д.В.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2023

Цель работы.

Ознакомиться с алгоритмом Ахо-Корасик поиска набора образцов в строке, применить его в решении поставленных задач.

Задание.

Задание 1.

Разработайте программу, решающую задачу точного поиска набора образцов.

Вход:

Первая строка содержит текст (T, $1 \le |T| \le 100000$).

Вторая - число n ($1 \le n \le 3000$), каждая следующая из n строк содержит шаблон из набора $P = \{p_1, \ldots, p_n\}$ $1 \le |p_i| \le 75$.

Все строки содержат символы из алфавита $\{A, C, G, T, N\}$.

Выход:

Все вхождения образцов из P в T.

Каждое вхождение образца в текст представить в виде двух чисел - i p.

Где i - позиция в тексте (нумерация начинается с 1), с которой начинается вхождение образца с номером p (нумерация образцов начинается с 1).

Строки выхода должны быть отсортированы по возрастанию, сначала номера позиции, затем номера шаблона.

Задание 2.

Используя реализацию точного множественного поиска, решите задачу точного поиска для одного образца с джокером.

В шаблоне встречается специальный символ, именуемый джокером (*wild card*), который "совпадает" с любым символом. По заданному содержащему шаблоны образцу P необходимо найти все вхождения P в текст T.

Например, образец *ab??c?* с джокером *?* встречается дважды в тексте *xabvccbababcax*.

Символ джокер не входит в алфавит, символы которого используются в T. Каждый джокер соответствует одному символу, а не подстроке неопределённой

длины. В шаблон входит хотя бы один символ не джокер, т.е. шаблоны вида *???* недопустимы. Все строки содержат символы из алфавита {*A, C, G, T, N*}.

Вход:

Текст (T,$1 \le |T| \le 100000$)

Шаблон (P,$1 \le |P| \le 40$)

Символ джокера

Выход:

Строки с номерами позиций вхождений шаблона (каждая строка содержит только один номер).

Номера должны выводиться в порядке возрастания.

Основные теоретические положения.

Алгоритм Ахо-Корасик (АК) - классическое решение задачи точного сопоставления множеств.

АК основан на структуре данных "дерево ключевых слов".

Поиск строки S в бору: начинаем в корне, идем по ребрам, отмеченным символами S, пока возможно.

Если с последним символом S мы приходим в вершину с сохраненным идентификатором, то S - слово из словаря.

Если в какой-то момент ребра, отмеченного нужным символом, не находится, то строки S в словаре нет.

Ясно, что это занимает O (|S|) времени. Таким образом, бор - это эффективный способ хранить словарь и искать в нем слова.

Теперь перейдем от бора к автомату, чтобы добиться поиска шаблонов в тексте за линейное время.

Действия автомата определяются тремя функциями, определенными для всех состояний:

Функция goto g(s, a) указывает, в какое состояние переходить из данного состояния s при просмотре символа a.

Функция неудачи f(s) указывает, в какое состояние переходить при просмотре неподходящего символа.

Выходная функция out(s) выдает множество шаблонов, которые обнаруживаются при переходе в состояние s.

Итоговая сложность алгоритма Ахо-Корасика — расход памяти O(n*V), вычислительная сложность O(n*V + H + k), где H — длина текста, в котором производится поиск, n — общая длина всех слов в словаре, V — размер алфавита, k — общая длина всех совпадений.

Выполнение работы.

Для решения поставленных задач определения вхождения подстроки в строку был реализован алгоритм Ахо-Корасик.

Для реализации алгоритма был создан вспомогательный класс, описывающий узлы дерева ключевых слов – бора.

type Node struct. Как описано раннее, данная структура описывает узлы бора. Представлен полями: *out, move, suf_link*. Словарь *move* – указывает, в какой узел переходить из узла по ключу при просмотре символа-ключа. Список *out* – множество шаблонов, которые обнаруживаются при переходе по символу. Переменная *suf_link* – узел для перехода при просмотре неподходящего символа.

Основные функции:

1) *func (trie *Trie) AddStrings(s []string)*. Функция, создающая бор – дерево паттернов. На вход подается набор шаблонов, по которым будет строиться дерево. С помощью вспомогательного класса, описанного выше, создается узел для бора. Соответствующие поля класса обновляются для каждого узла: в словарь *move* добавляются символы из шаблонов, по которым будут храниться другие узлы для перехода, в список *out* добавляется номер обрабатываемого на данном этапе шаблона. Функция ничего не возвращает, при этом заполняет узлы дерева.

2) *func (tree *Trie) GetSuffixLinks()*. Функция, создающая автомат Ахо-Корасик, а именно вычисляющая суффиксные ссылки для перехода при

просмотре неподходящих символов. Дерево обходится в ширину. Переменная *suf_link*, узел для перехода при просмотре неподходящего символа, и список *out* с множеством шаблонов вычисляются для всех вершин в порядке обхода. Данная функция находит указатель на строку, которая есть в боре, но при этом являющаяся самым длинным суффиксом для текущей обрабатываемой подстроки.

3) *func (tree *Trie) GetIntervals(content string, count int) (intervals []IntervalIndices).* На вход получает исходную строку, вычисляет размер одного чанка и делит строку на пересекающиеся подстроки, возвращает их список – требуется для подготовки к параллельному выполнению

4) *func (tree *Trie) FindAll(content string, count int) (results []SearchResult).* Функция поиска вхождения всех шаблонов, хранящихся в боре в строку. На вход получает обрабатываемый текст, а так же количество потоков для выполнения. Запускает горутины, которые параллельно выполняют вычисления на разбитых ранее интервалах, найденные результаты записываются в общий поток, после чего поток закрывается, а хранимые в нем значения извлекаются, удаляются повторяющиеся, и сортируются

5) func CreatePatterns(pattern string, joker string) ([]string, []int). Функция принимающая на вход строку шаблона с маской и символ-джокер, разделяет строку и вычисляет вхождения подстрок в исходный шаблон, возвращает слайсы – подстроки и индексы вхождений.

6) *func CalcSubstrings(text string, indices []int, substrings []SearchResult) []int.* Функция, считающая, количество безмасочных подстрок, встретившихся в тексте. Принимает на вход текст, начальные индексы шаблонов в едином шаблоне, слайс результатов поиска подстрок в тексте. Для алгоритма понадобится список *amount*, где *amount[i]* – количество встретившихся в тексте безмасочных подстрок шаблона, который начинается в тексте на позиции *i*. Появление *i*-ой подстроки в тексте на позиции *j* будет означать возможное появление единого шаблона *pattern* на позиции *j-indices[pattern]*. То есть при нахождении *i*-ой подстроки в тексте на позиции *j* увеличиваем на единицу

список, объявленный раннее в данном пункте, по индексу, равному позиции возможного появления единого шаблона. Функция возвращает список, хранящий по индексу количество встретившихся подстрок.

*7) func (tree *Trie) FindMaskPattern(text string, pattern string, joker string, n int).* Функция, выполняющая подготовку строки с шаблоном для распараллеливания и дальнейшего поиска, запускающая поиск подстроки с шаблоном в строке.

Разработанный программный код смотреть в приложении А.

Тестирование.

Задание 1.

Входные данные	**Вывод**	**Ожидание**
NTAG 3 TAGT TAG T 2	FOUND THE RESULT 2 2 2 3	**2 2** **2 3**
CAGA 3 CAG AGA G 2	FOUND THE RESULT 1 1 2 2 3 3	**1 1** **2 2** **3 3**
CAGA 3 A AAA AA 10	FOUND THE RESULT 2 1 4 1	**2 1** **4 1**

Задание 2.

Входные данные	**Вывод**	**Ожидание**
ABBABABA A**A * 5	FOUND THE RESULT 1	**1**
ACTANCA A$$A$ $ 5	FOUND THE RESULT 1	**1**

AABBBAA NNNN N 5	FOUND THE RESULT 1 2 3 4	**1** **2** **3** **4**

Выводы.

В ходе лабораторной работы был изучен алгоритм Ахо-Корасик. Разработан программный код, позволяющий решить следующие задачи с помощью данного алгоритма: точный поиск набора образцов в тексте и поиск для одного образца с джокером. На языке программирования *Go* реализованы функции, представляющие собой решение поставленных задач.

Для работы алгоритма Ахо-Корасик изучена структура данных «дерево ключевых слов» или бор. Реализован класс, описывающий узел такого дерева, и функция, строящее такое дерево. Для нахождения шаблонов в тексте реализован автомат Ахо-Корасик, состояниями которого являются узлы бора.

Для решения задачи поиска одного образца с джокером была осуществлена предобработка образца перед началом алгоритма. Образец делился на подстроки по разделителю, которым являлся символ джокера, а также для получившихся подстрок сохранялись их стартовые позиции в образце. Это позволяет решать данную задачу по такому же принципу, как и первую.

Разработанный программный код для решения поставленных задач успешно прошел тестирование на онлайн платформе *Stepik*.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: aho.go

```
package aho

import (
	"fmt"
	"sort"
	"strings"
	"sync"
)

// Структура узла бора
type Node struct {
	out      []int
	move     map[byte]*Node
	suf_link *Node
}

// Конструктор ноды
func NewNode() *Node {
	return &Node{
		out:      make([]int, 0, 0),
		move:     make(map[byte]*Node),
		suf_link: nil,
	}
}

// структура бора
type Trie struct {
	root     *Node
	patterns []string
}

// конструктор бора
func NewTrie() *Trie {
```

```
	return &Trie{
		root:     NewNode(),
		patterns: make([]string, 0, 0),
	}
}

// функция добавляющая шаблоны в бор
// идет по узлам, если нет перехода по очередному символу - создает
ноду
// и переходит туда
// добавляет шаблоны в ноды
func (trie *Trie) AddStrings(s []string) {
	fmt.Printf("Creating tree:\n")
	for i, elem := range s {
		x := trie.root
		for _, sym := range []byte(elem) {
			if _, ok := x.move[sym]; !ok {
				x.move[sym] = NewNode()
				fmt.Printf("added symbol: %s\n", string(sym))
			}
			x = x.move[sym]
		}
		x.out = append(x.out, i)
		fmt.Printf("To vertex %+v added pattern: %s\n", x, elem)
		trie.patterns = append(trie.patterns, elem)
	}

}

//Функция вычисляет суффиксные ссылки для каждой ноды
// Ищем самую длинную суффиксную ссылку и записываем ее в
// соответствующее поле ноды
func (tree *Trie) GetSuffixLinks() {
	fmt.Printf("\nGetting suffix links\n")
	queue := make([]*Node, 0, 0)

	for _, child := range tree.root.move {
		child.suf_link = tree.root
		child.out = append(child.out, child.suf_link.out...)
```

```
                queue = append(queue, child)
        }

        for len(queue) != 0 {

                parent := queue[0]
                queue = queue[1:]

                for key, unode := range parent.move {
                        queue = append(queue, unode)
                        fnode := parent.suf_link
                        _, ok := fnode.move[key]

                        for fnode != nil && !ok {
                                fnode = fnode.suf_link
                                if fnode != nil {
                                        _, ok = fnode.move[key]
                                }
                        }

                        if ok {
                                unode.suf_link = fnode.move[key]
                                fmt.Printf("Suffix link from %+v to %+v\n",
unode, fnode.move[key])
                        } else {
                                unode.suf_link = tree.root
                                fmt.Printf("Suffix link from %+v to root\n",
unode)
                        }
                       unode.out = append(unode.out, unode.suf_link.out...)
                        fmt.Printf("Into %+v added patterns: %v\n", unode,
unode.suf_link.out)
                }
        }
    }

    // структура результата поиска
    type SearchResult struct {
        Start, Pattern int
```

```
	}

	// структура хранящая границы интервалов
	// при подготовке к распараллеливанию
	type IntervalIndices struct {
		Start, Finish int
	}

	// разбивает строку на n пересекающихся частей для параллельной обработки
	// возвращает список границ интервалов
	func (tree *Trie) GetIntervals(content string, count int) (intervals []IntervalIndices) {
		fmt.Printf("\nGetting Intervals\n")
		intervals = make([]IntervalIndices, 0)

		GetMaxLen := func() (res int) {
			for _, elem := range tree.patterns {
				if len(elem) > res {
					res = len(elem)
				}
			}
			fmt.Printf("Max Length of pattern: %d\n", res)
			return
		}

		maxLen := GetMaxLen()
		partSize := int(len(content) / count)
		fmt.Printf("Size of slice of the text: %d + %d - 1 = %d\n", partSize, maxLen, partSize+maxLen-1)

		i := 0
		for i < count-1 {

			if len(content) < i*partSize+partSize+maxLen-1 {
				break
			}

			intervals = append(intervals,
```

```
                    IntervalIndices{
                        Start:  i * partSize,
                        Finish: i*partSize + partSize + maxLen - 1,
                    })
                i++
            }
            intervals = append(intervals, IntervalIndices{Start: i *
partSize, Finish: len(content)})
            fmt.Printf("Found indices of substring:\n")
            for _, elem := range intervals {
                fmt.Printf("%+v\n", elem)
            }
            return
        }

        // осуществление поиска, запуск горутин, обработка ответа
        func (tree *Trie) FindAll(content string, count int) (results
[]SearchResult) {

            // получили интервал
            intervals := tree.GetIntervals(content, count)
            // создали канал
            res := make(chan SearchResult, len(content)*count)

            var wg sync.WaitGroup //  инициализируем счётчик
            wg.Add(len(intervals))

            fmt.Printf("\nStart %d goroutines\n", len(intervals))
            // запускаем горутины
            for _, indices := range intervals {
                go func(indices IntervalIndices) {
                    fmt.Printf("Finding in the interval %+v started\n",
indices)
                    iter := tree.root

                    i := indices.Start
                    for _, c := range
[]byte(content[indices.Start:indices.Finish]) {
                        _, ok := iter.move[c]
```

```
                        for !ok && iter != tree.root {
                              iter = iter.suf_link
                              _, ok = iter.move[c]
                        }

                        if !ok && iter == tree.root {
                              i++
                              continue
                        }

                        iter = iter.move[c]
                        // сохраняем те шаблоны, которые хранятся в
ноде
                        for _, pattern := range iter.out {
                            fmt.Printf("In interval %+v found %+v\n",
indices, SearchResult{Start: i - len(tree.patterns[pattern]) + 2, Pattern:
pattern + 1})
                              res    <-    SearchResult{Start:    i    -
len(tree.patterns[pattern]) + 2, Pattern: pattern + 1}
                        }
                        i++
                  }
                  wg.Done()
            }(indices)
      }
      // ожидаем все горутины и закрываем канал
      wg.Wait()
      close(res)
      fmt.Printf("Goroutines finished, channel closed\n")

      unique := make(map[SearchResult]bool)
      // сохраняем только уникальные вхождения
      fmt.Printf("Delete doublicated inclusions:\n")
      for elem := range res {
            if !unique[elem] {
                  unique[elem] = true
                  results = append(results, elem)
            } else {
                  fmt.Printf("%+v already exists in result\n", elem)
```

```
                }
        }
        // сортируем результаты
        sort.Slice(results, func(i, j int) bool {
                if results[i].Start == results[j].Start {
                        return results[i].Pattern < results[j].Pattern
                }
                return results[i].Start < results[j].Start
        })

        return
}

// разбиваем строку по джокеру и вычисляем
// вхождения каждой подстроки в исходный шаблон
func CreatePatterns(pattern string, joker string) ([]string, []int)
{

        preproc := strings.Split(pattern, joker)

        patterns := make([]string, 0)
        for _, elem := range preproc {
                if elem != "" {
                        patterns = append(patterns, elem)
                }
        }
        fmt.Printf("substrings without joker: %+v\n", patterns)

        patternIndices := make([]int, 0)

        startIndex := 0

        fmt.Printf("Calculation of indices of substrings in
pattern:\n")

        for _, string := range patterns {
                fmt.Printf("Current substring: %s\n", string)
                curSection := pattern[startIndex:]
                fmt.Printf("Finding in %+v section\n", curSection)
```

```
                curIndex := strings.Index(curSection, string)
                patternIndices = append(patternIndices,
curIndex+(len(pattern)-len((curSection))))
                fmt.Printf("Inclusion index of %s - %d\n", string,
curIndex+(len(pattern)-len((curSection))))
                startIndex = curIndex + len(string) + (len(pattern) -
len((curSection)))
                fmt.Printf("Next start index: %d\n", startIndex)
          }

          fmt.Printf("Patterns: %v, Indices: %v\n", patterns,
patternIndices)
          return patterns, patternIndices
     }

     // вычисляем кол-во вхождений i-го шаблона на j-ой позиции
     func CalcSubstrings(text string, indices []int, substrings
[]SearchResult) []int {
          amount := make([]int, len(text))
          for _, inclusion := range substrings {
                curIndex := inclusion.Start - indices[inclusion.Pattern-
1] - 1
                if curIndex >= 0 && curIndex < len(amount) {
                     amount[curIndex]++
                }
          }
          return amount
     }

     func (tree *Trie) FindMaskPattern(text string, pattern string,
joker string, n int) {
          // получаем шаблоны
          subpatterns, indices := CreatePatterns(pattern, joker)
          // строим автомат
          tree.AddStrings(subpatterns)
          tree.GetSuffixLinks()
          // ищем вхождения подстрок шаблона
          substrings := tree.FindAll(text, n)
```

```
            subpatternsAmount := CalcSubstrings(text, indices, substrings)

            func(amount []int, pattern string, subpatterns []string) {
                  fmt.Printf("FOUND THE RESULT\n")
                  for i := 0; i < len(amount)-len(pattern)+1; i++ {
                        if amount[i] == len(subpatterns) {
                              fmt.Println(i + 1)
                        }
                  }
            }(subpatternsAmount, pattern, subpatterns)
      }
```

Файл: prime.go

```
package main

import (
      "aho_corasik/aho"
      "fmt"
)

func main() {
      var num int
      var text string
      var n int

      fmt.Scanf("%s", &text)
      fmt.Scanf("%d", &num)

      patterns := make([]string, num, num)

      for i := 0; i < num; i++ {
            fmt.Scanf("%s", &patterns[i])
      }
      fmt.Scanf("%d", &n)

      tree := aho.NewTrie()

      tree.AddStrings(patterns)

      tree.GetSuffixLinks()
      res := tree.FindAll(text, n)

      fmt.Printf("FOUND THE RESULT\n")
      for _, elem := range res {
            fmt.Println(elem.Start, elem.Pattern)
      }
}
```

Файл: mask.go

```
package main

import (
	"aho_corasik/aho"
	"fmt"
)

func main() {
	var mask string
	var text string
	var joker string
	var n int

	fmt.Printf("Program started, please input data\n")
	fmt.Printf("text: ")
	fmt.Scanf("%s", &text)
	fmt.Printf("mask: ")
	fmt.Scanf("%s", &mask)
	fmt.Printf("joker: ")
	fmt.Scanf("%s", &joker)
	fmt.Printf("count of goroutines: ")
	fmt.Scanf("%d", &n)

	tree := aho.NewTrie()
	tree.FindMaskPattern(text, mask, joker, n)

}
```