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

ОТЧЕТ

по лабораторной работе №3
по дисциплине «Построение и анализ алгоритмов»
Тема: Редакционное расстояние
Вариант 1.

Студент гр. 3388 _____ Дубровин Д.Н.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2025

Цель работы:

Изучить алгоритм Вагнера-Фишера для нахождения редакционного расстояния Левенштейна. Реализовать задание в соответствии с вариантом.

Задание.

Расстоянием Левенштейна назовём минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую. Разработайте программу, осуществляющую поиск расстояния Левенштейна между двумя строками.

Пример:

Для строк pedestal и stien расстояние Левенштейна равно 7:

- Сначала нужно совершить четыре операции удаления символа: pedestal -> stal.
- Затем необходимо заменить два последних символа: stal -> stie.
- Потом нужно добавить символ в конец строки: stie -> stien.

Параметры входных данных:

Первая строка входных данных содержит строку из строчных латинских букв.(S,$1 \le |S| \le 2550$).

Вторая строка входных данных содержит строку из строчных латинских букв. (T, $1 \le |T| \le 2550$).

Параметры выходных данных:

Одно число L, равное расстоянию Левенштейна между строками S и T.

Sample Input:

pedestal

stien

Sample Output:

7

Реализация

Программа реализует алгоритм Вагнера-Фишера, который использует идею динамического программирования для вычисления редакционного расстояния между двумя строками. Редакционное расстояние — это минимальное количество операций редактирования (вставка, удаление, замена), необходимых для преобразования строки S в строку T.

Описание функций и структур:

Структуры

`OperationCosts`

Назначение: Определяет стоимости операций редактирования для вычисления расстояния Левенштейна.

Поля:

- Replace int: Стоимость замены одного символа на другой.
- Insert int: Стоимость вставки символа.
- Delete int: Стоимость удаления символа.
- SpecialReplace int: Стоимость замены специальной руны.
- SpecialInsert int: Стоимость вставки специальной руны.

`SpecialRunes`

Назначение: Указывает руны, для которых применяются специальные стоимости операций.

Поля:

- Replace rune: Руна, вызывающая специальную стоимость замены.
- Insert rune: Руна, вызывающая специальную стоимость вставки.

Методы

`buildPath(n, m int, opCosts OperationCosts, ops [][]rune, dp [][]int) string`

Назначение: Строит строку пути операций (Match, Replace, Insert, Delete) на основе матрицы операций и динамического программирования.

Параметры:

- n, m: Длины строк s1 и s2.
- opCosts: Указатель на структуру с настройками стоимостей операций.
- ops: Матрица операций.
- dp: Матрица стоимостей (динамическое программирование).

Возвращает: Строку, представляющую последовательность операций.

`minOperation(replaceTotal, insertTotal, deleteTotal int) (rune, int)`

Назначение: Определяет операцию с минимальной стоимостью среди замены, вставки и удаления.

Параметры:

- replaceTotal: Общая стоимость замены.
- insertTotal: Общая стоимость вставки.
- deleteTotal: Общая стоимость удаления.

Возвращает: Руну, соответствующую минимальной операции (Replace, Insert, Delete), и её стоимость.

`FindLevenshteinDistance(s1, s2 string, opCosts *OperationCosts, specRunes SpecialRunes, log logger.Logger) (int, string)`

Назначение: Вычисляет расстояние Левенштейна между строками s1 и s2 с учетом пользовательских стоимостей операций и специальных рун, логируя процесс.

Параметры:

- s1, s2: Входные строки для сравнения.
- opCosts: Указатель на структуру с настройками стоимостей операций.
- specRunes: Указатель на структуру с настройками специальных рун.
- log: Указатель на логгер для записи процесса вычислений.

Возвращает: Расстояние Левенштейна (целое число) и строку пути операций.

Интерфейс

`Logger`

Назначение: Определяет методы для логирования сообщений и матриц.

Методы:

- LogMsg(title, message string): Логирует текстовое сообщение с заголовком.
- LogRuneMatrix(title string, data [][]rune): Логирует матрицу рун.
- LogCostMatrix(title string, data [][]int): Логирует матрицу стоимостей.
- SetDebugMode(): Включает режим отладки.

Оценка сложности алгоритма:

Временная сложность:

- Алгоритм проверяет все комбинации частей S (длина n) и T (длина m), сравнивая символы и выбирая лучший вариант.
- Для каждой пары позиций делается фиксированная работа: сравнение символов и выбор минимума из трёх чисел.
- Начальная подготовка (вставки T и удаления S) занимает n+m.

Итог: O(n·m)

Пространственная сложность

Хранение промежуточных результатов:

- Нужно помнить количество шагов для всех комбинаций частей S и T — n·m значений.

Итог: O(n·m)

Тестирование

Таблица 1. Тестирование.

Входные данные	Выходные данные
Enter the costs (replace, insert, delete): 1 1 1 Enter special runes (replace, insert): a b Enter special runes costs (replace, insert): 4 2 Enter the first string: aboba Enter the second string: abama	Levenshtein distance: 2 Operations sequence: MMRRM
Enter the costs (replace, insert, delete): 1 1 1 Enter special runes (replace, insert): b v Enter special runes costs (replace, insert): 2 3 Enter the first string: moevm Enter the second string: moevm	Levenshtein distance: 0 Operations sequence: MMMMM
Enter the costs (replace, insert, delete): 1 1 1 Enter special runes (replace, insert): & (Enter special runes costs (replace, insert): 5 5 Enter the first string: entrance Enter the second string: reenterable	Levenshtein distance: 5 Operations sequence: RRRRIRRIRRI

Вывод

В ходе лабораторной работы была написана программа, реализующая алгоритм Вагнера-Фишера для поиска редакционного расстояния. В программе был предусмотрен режим для подробного логгирования всех этапов алгоритма.

Исходный код программы см. в ПРИЛОЖЕНИИ А.

ПРИЛОЖЕНИЕ А.
ИСХОДНЫЙ КОД ПРОГРАММЫ

Filename: main.go

```
package main

import (
	"bufio"
	"flag"
	"fmt"
	"os"
	"strconv"
	"strings"

	"lb3_Levenshtein/logger"
	"lb3_Levenshtein/vagner_fisher"
)

func readInputStrings(reader *bufio.Reader, writer *bufio.Writer)
(string, string) {
	fmt.Fprint(writer, "Enter the first string: ")
	writer.Flush()
	s1, _ := reader.ReadString('\n')
	s1 = strings.TrimSpace(s1)

	fmt.Fprint(writer, "Enter the second string: ")
	writer.Flush()
	s2, _ := reader.ReadString('\n')
	s2 = strings.TrimSpace(s2)

	return s1, s2
}

func readInputConfig(reader *bufio.Reader, writer *bufio.Writer)
([]string, []string, []string) {
```

```
	fmt.Fprint(writer, "Enter the costs (replace, insert, delete): ")
	writer.Flush()
	costsInput, _ := reader.ReadString('\n')
	costs := strings.Split(strings.TrimSpace(costsInput), " ")
	if len(costs) != 3 {
		fmt.Fprintln(os.Stderr, "Invalid input. Please enter 3 costs separated by spaces.")
		os.Exit(1)
	}

	fmt.Fprint(writer, "Enter special runes (replace, insert): ")
	writer.Flush()
	specialRunesInput, _ := reader.ReadString('\n')
	specialRunesStrs := strings.Split(strings.TrimSpace(specialRunesInput), " ")
	if len(specialRunesStrs) != 2 {
		fmt.Fprintln(os.Stderr, "Invalid input. Please enter 2 special runes separated by spaces.")
		os.Exit(1)
	}

	fmt.Fprint(writer, "Enter special runes costs (replace, insert): ")
	writer.Flush()
	specialRunesCostsInput, _ := reader.ReadString('\n')
	specialRunesCosts := strings.Split(strings.TrimSpace(specialRunesCostsInput), " ")
	if len(specialRunesCosts) != 2 {
		fmt.Fprintln(os.Stderr, "Invalid input. Please enter 2 special runes costs separated by spaces.")
		os.Exit(1)
	}

	return costs, specialRunesStrs, specialRunesCosts
}

func main() {
	debugMode := flag.Bool("debug", false, "Enable debug mode.")
```

```
	flag.Parse()

	if *debugMode {
		fmt.Println("Debug mode enabled.")
	}

	reader := bufio.NewReader(os.Stdin)
	writer := bufio.NewWriter(os.Stdout)
	defer writer.Flush()

	costs, specialRunesStrs, specialRunesCosts := readInputConfig(reader, writer)
	s1, s2 := readInputStrings(reader, writer)

	parseCost := func(costStr string) int {
		cost, err := strconv.Atoi(costStr)
		if err != nil {
			fmt.Fprintln(os.Stderr, "Error parsing cost:", err)
			os.Exit(1)
		}
		return cost
	}

	parseRunes := func(runeStr string) rune {
		if len(runeStr) != 1 {
			fmt.Fprintln(os.Stderr, "Invalid rune input. Please enter a single character.")
			os.Exit(1)
		}
		return rune(runeStr[0])
	}

	specialRunes := vagner_fisher.SpecialRunes{
		Replace: parseRunes(specialRunesStrs[0]),
		Insert:  parseRunes(specialRunesStrs[1]),
	}
```

```
	opCosts := vagner_fisher.OperationCosts{
		Replace:        parseCost(costs[0]),
		Insert:         parseCost(costs[1]),
		Delete:         parseCost(costs[2]),
		SpecialReplace: parseCost(specialRunesCosts[0]),
		SpecialInsert:  parseCost(specialRunesCosts[1]),
	}

	log := logger.NewLogger(writer)
	if *debugMode {
		log.SetDebugMode()
	}

	distance, operations := vagner_fisher.FindLevenshteinDistance(s1, s2, &opCosts, &specialRunes, log)

	fmt.Fprintln(writer, "\nResults:")
	fmt.Fprintln(writer, "Levenshtein distance: "+strconv.Itoa(distance))
	fmt.Fprintln(writer, "Operations sequence: "+operations)
}
```

Filename: vagner_fisher.go

```
package vagner_fisher

import (
	"fmt"

	"lb3_Levenshtein/logger"
)

var debug bool = false

type Logger interface {
	LogMsg(title, message string)
```

```
        LogRuneMatrix(title string, data [][]rune)
        LogCostMatrix(title string, data [][]int)
        SetDebugMode()
}

const (
        Match   = 'M'
        Replace = 'R'
        Insert  = 'I'
        Delete  = 'D'
)

type OperationCosts struct {
        Replace        int
        Insert         int
        Delete         int
        SpecialReplace int
        SpecialInsert  int
}

type SpecialRunes struct {
        Replace rune
        Insert  rune
}

func buildPath(n, m int, opCosts *OperationCosts, ops [][]rune, dp [
[]int) string {
        var path []rune
        i, j := 0, 0
        for i < n || j < m {
                if i < n && j < m && ops[i+1][j+1] == Match {
                        path = append(path, Match)
                        i++
                        j++
                            } else if j < m && (i == n || dp[i]
[j+1]+opCosts.Insert == dp[i][j]) {
                        path = append(path, Insert)
                        j++
```

```
                        } else if i < n && (j == m || dp[i+1][j]
+opCosts.Delete == dp[i][j]) {
                                path = append(path, Delete)
                                i++
                        } else {
                                path = append(path, Replace)
                                i++
                                j++
                        }
                }

                return string(path)
}

func minOperation(replaceTotal, insertTotal, deleteTotal int) (rune,
int) {
                minCost := replaceTotal
                minOp := Replace

                if insertTotal < minCost {
                        minCost = insertTotal
                        minOp = Insert
                }
                if deleteTotal < minCost {
                        minCost = deleteTotal
                        minOp = Delete
                }

                return minOp, minCost
}

func FindLevenshteinDistance(s1, s2 string, opCosts *OperationCosts,
specRunes *SpecialRunes, log *logger.Logger) (int, string) {
                n, m := len(s1), len(s2)

                dp := make([][]int, n+1)
                ops := make([][]rune, n+1)
```

```
        for i := range dp {
                dp[i] = make([]int, m+1)
                ops[i] = make([]rune, m+1)
        }

          log.LogMsg("Init", fmt.Sprintf("Calculating distance between
'%s' (%d) and '%s' (%d)", s1, n, s2, m),
                logger.ColorCyan)
            log.LogMsg("Costs", fmt.Sprintf("Replace: %d, Insert: %d,
Delete: %d, SpecialReplace: %d, SpecialInsert: %d",
                    opCosts.Replace, opCosts.Insert, opCosts.Delete,
opCosts.SpecialReplace, opCosts.SpecialInsert),
                logger.ColorCyan)
          log.LogMsg("SpecialRunes", fmt.Sprintf("Replace: %c, Insert:
%c", specRunes.Replace, specRunes.Insert),
                logger.ColorCyan)

        dp[0][0] = 0
        ops[0][0] = Match

        for j := 1; j <= m; j++ {
                dp[0][j] = dp[0][j-1] + opCosts.Insert
                ops[0][j] = Insert
        }

        for i := 1; i <= n; i++ {
                dp[i][0] = dp[i-1][0] + opCosts.Delete
                ops[i][0] = Delete
        }

        log.LogCostMatrix("Initial DP", dp, logger.ColorRed)
        log.LogRuneMatrix("Initial Ops", ops, logger.ColorBlue)

        for i := 1; i <= n; i++ {
                for j := 1; j <= m; j++ {
                        if s1[i-1] == s2[j-1] {
                                dp[i][j] = dp[i-1][j-1]
```

```
                                ops[i][j] = Match
                                                log.LogMsg("Match",
fmt.Sprintf("Characters match at (%d,%d): %c", i, j, s1[i-1]),
                                        logger.ColorGreen)
                        } else {
                                replaceCost := opCosts.Replace
                                  if rune(s1[i-1]) == specRunes.Replace
{
                                                        replaceCost =
opCosts.SpecialReplace
                                            log.LogMsg("SpecialReplace",
fmt.Sprintf("Special replace at (%d,%d): %c", i, j, s1[i-1]),
                                                    logger.ColorPurple)
                                }

                                insertCost := opCosts.Insert
                                if rune(s2[j-1]) == specRunes.Insert {
                                                         insertCost =
opCosts.SpecialInsert
                                             log.LogMsg("SpecialInsert",
fmt.Sprintf("Special insert at (%d,%d): %c", i, j, s2[j-1]),
                                                    logger.ColorPurple)
                                }

                                        replaceTotal := dp[i-1][j-1] +
replaceCost
                                insertTotal := dp[i][j-1] + insertCost
                                           deleteTotal := dp[i-1][j] +
opCosts.Delete

                                                    minOp, minCost :=
minOperation(replaceTotal, insertTotal, deleteTotal)

                                ops[i][j] = minOp
                                dp[i][j] = minCost
```

```
                                                log.LogMsg("Operation",
fmt.Sprintf("Cell (%d,%d): chose %c with cost %d (replace=%d, insert=
%d, delete=%d)",
                                                  i, j, minOp, minCost,
replaceTotal, insertTotal, deleteTotal),
                                            logger.ColorYellow)
                        }
                }
        }

        log.LogCostMatrix("Final DP", dp, logger.ColorRed)
        log.LogRuneMatrix("Final Ops", ops, logger.ColorBlue)

        path := buildPath(n, m, opCosts, ops, dp)
           log.LogMsg("Result", fmt.Sprintf("Final distance: %d, Path:
%s", dp[n][m], path),
                logger.ColorGreen)

        return dp[n][m], path
}
```

Filename: logger.go

```
package logger

import (
        "bufio"
        "fmt"
)

const (
        ColorReset  = "\033[0m"
        ColorRed    = "\033[31m"
        ColorGreen  = "\033[32m"
        ColorYellow = "\033[33m"
        ColorBlue   = "\033[34m"
        ColorPurple = "\033[35m"
```

```
        ColorCyan   = "\033[36m"
        ColorWhite  = "\033[37m"
)

type Logger struct {
        Writer *bufio.Writer
        Debug  bool
}

func NewLogger(writer *bufio.Writer) *Logger {
        return &Logger{
                Writer: writer,
                Debug:  false,
        }
}

func (l *Logger) SetDebugMode() {
        l.Debug = true
}

func (l *Logger) LogMsg(title, message, color string) {
        if l.Debug {
                  fmt.Fprintf(l.Writer, "%s[%s]:%s %s\n", color, title,
ColorReset, message)
                l.Writer.Flush()
        }
}

func (l *Logger) LogRuneMatrix(title string, data [][]rune, color
string) {
        if l.Debug {
                   fmt.Fprintf(l.Writer, "%s[%s]:%s \n", color, title,
ColorReset)
                for _, row := range data {
                        for _, val := range row {
                                fmt.Fprint(l.Writer, string(val), " ")
                        }
```

```
                        fmt.Fprint(l.Writer, "\n")
                }
                l.Writer.Flush()
        }
}

func (l *Logger) LogCostMatrix(title string, data [][]int, color
string) {
        if l.Debug {
                   fmt.Fprintf(l.Writer, "%s[%s]:%s \n", color, title,
ColorReset)
                for _, row := range data {
                        for _, val := range row {
                                fmt.Fprint(l.Writer, val, " ")
                        }
                        fmt.Fprint(l.Writer, "\n")
                }
                l.Writer.Flush()
        }
}
```