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 №2 по курсу "Анализ алгоритмов"

Тема Алгоритм Копперсмита-Винограда

Студент Якуба Д. В.

Группа ИУ7-53Б

Оценка (баллы) _____

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

Москва — 2020 г.

Оглавление

Введение

Цели лабораторной работы

1. изучение алгоритмов умножения матриц: классического, Копперсмита-Винограда и модифицированного Копперсмита-Винограда;
2. реализация алгоритмов умножения матриц: классического, Копперсмита-Винограда и модифицированного Копперсмита-Винограда;
3. проведение сравнительного анализа трудоёмкости алгоритмов на основе теоретических расчетов и выбранной модели вычислений;
4. сравнительный анализ алгоритмов на основе экспериментальных данных;
5. подготовка отчёта по лабораторной работе.

Определение

Алгоритм Копперсмита-Винограда = это алгоритм умножения квадратных матриц, предложенный в 1987 году Д. Копперсмитом и Ш. Виноградом [1]. В исходной весрии асимптотическая сложность алгоритма составляла $O(n^{2.3755})$, где n - это размер стороны матрицы. Алгоритм Копперсмита-Винограда с учётом усерии улучшений и доработок в последующие годы, обладает лучшей асимптотикой среди известных алгоритмов умножения матриц.

На практике алгоритм Копперсмита — Винограда не используется, так как он имеет очень большую константу пропорциональности и начинает выигрывать в быстродействии у других известных алгоритмов

только для матриц, размер которых превышает память современных компьютеров [2]. Поэтому пользуются алгоритмом Штрассена по причинам простоты реализации и меньшей константе в оценке трудоемкости.

1 Аналитическая часть

1.1 Классический алгоритм умножения матриц

Пусть даны две прямоугольные матрицы

$$A_{lm} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{l1} & a_{l2} & \dots & a_{lm} \end{pmatrix}, \quad B_{mn} = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \dots & b_{mn} \end{pmatrix}, \tag{1.1}$$

тогда матрица C

$$C_{ln} = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{l1} & c_{l2} & \dots & c_{ln} \end{pmatrix}, \tag{1.2}$$

где

$$c_{ij} = \sum_{r=1}^{m} a_{ir} b_{rj} \quad (i = \overline{1,l}; j = \overline{1,n}) \tag{1.3}$$

будет называться произведением матриц A и B.

Реализация классического алгоритма умножения двух матриц заключается в реализации вычисления элементов итоговой матрицы по формуле 1.3

1.2 Алгоритм Копперсмита-Винограда умножения матриц

Если посмотреть на результат умножения двух матриц, то видно, что каждый элемент в нем представляет собой скалярное произведение соответствующих строки и столбца исходных матриц. Можно заметить также, что такое умножение допускает предварительную обработку, позволяющую часть работы выполнить заранее.

Рассмотрим два вектора $V = (v_1, v_2, v_3, v_4)$ и $W = (w_1, w_2, w_3, w_4)$. Их скалярное произведение равно: $V \cdot W = v_1w_1 + v_2w_2 + v_3w_3 + v_4w_4$, что эквивалентно (1.4):

$$V \cdot W = (v_1+w_2)(v_2+w_1)+(v_3+w_4)(v_4+w_3)-v_1v_2-v_3v_4-w_1w_2-w_3w_4. \quad (1.4)$$

Несмотря на то, что второе выражение требует вычисления большего количества операций, чем стандартный алгоритм: вместо четырех умножений - шесть, а вместо трех сложений - десять, выражение в правой части последнего равенства допускает предварительную обработку: его части можно вычислить заранее и запомнить для каждой строки первой матрицы и для каждого столбца второй, что позволит для каждого элемента выполнять лишь два умножения и пять сложений, складывая затем только лишь с 2 предварительно посчитанными суммами соседних элементов текущих строк и столбцов. Из-за того, что операция сложения быстрее операции умножения в ЭВМ, на практике алгоритм должен работать быстрее стандартного [3].

Вывод

Были рассмотрены классический алгоритм множения матриц и алгоритм Копперсмита-Винограда умножения матриц. Основное отличие данных алгоритмов заключается в наличии предварительной обработки данных и количестве проводящихся операций умножения.

2 | Конструкторская часть

2.1 Блок-схема рекурсивного алгоритма Левенштейна

2.2 Модель вычислений

Для последующего вычисления трудоемкости необходимо ввести модель вычислений:

1. операции из списка (2.1) имеют трудоемкость 1;

$$+, -, /, \%, ==, !=, <, >, <=, >=, [], ++, --, +=, -= \tag{2.1}$$

2. трудоемкость оператора выбора if условие then A else B рассчитывается, как (2.2);

$$f_{if} = f_{\text{условия}} + \begin{cases} f_A, & \text{если условие выполняется,} \\ f_B, & \text{иначе.} \end{cases} \tag{2.2}$$

3. трудоемкость цикла рассчитывается, как (2.3);

$$f_{for} = f_{\text{инициализации}} + f_{\text{сравнения}} + N(f_{\text{тела}} + f_{\text{инкремента}} + f_{\text{сравнения}}) \tag{2.3}$$

4. трудоемкость вызова функции равна 0.

3 | Технологическая часть

3.1 Требования к программному обеспечению

3.2 Средства реализации программного обеспечения

При написании программного продукта был использован язык программирования Kotlin [4].

Данный выбор обусловлен следующими факторами:

- Высокая вычислительная производительность;.

Для тестирования производительности реализаций алгоритмов использовалась утилита measureNanoTime.

При написаннии программного продукта использовалась среда разработки IntelliJ IDEA.

Данный выбор обусловлен тем, что язык программирования Kotlin - это разработка компании JetBrains, поставляющей данную среду разработки;

3.3 Листинг кода

В листингах 3.1 - 3.4 предоставлены реализации рассматриваемых алгоритмов.

Листинг 3.1: Функция реализации рекурсивного алгоритма Левенштейна

```
1 s
```

Листинг 3.2: Функция реализации рекурсивного алгоритма Левенштейна с использованием матрицы расстояний

Листинг 3.3: Функция реализации итеративного алгоритма Левенштейна

Листинг 3.4: Функция реализации алгоритма Дамерау-Левенштейна

3.4 Тестирование программного продукта

В таблице 3.1 приведены тестовые данные и вывод программы для алгоритмов вычисления расстояния Левенштейна и Дамерау-Левенштейна. Тесты пройдены успешно.

Таблица 3.1: Тесты

		Ожидаемый результат	
Строка 1	Строка 2	Алг. Левенштейна	Алг. Дамерау-Левенштейна
table	tumbler	3	3
hell	help	1	1
KillUsAll	KlilUsAll	2	1
smoke	mssql	5	4
OfMiceAndMen	OfMonstersAndMen	6	6
roofer	killer	4	4
orange	orangina	3	3
prolifer	profiler	2	2
cat	dog	3	3

Вывод

Спроектированные алгоритмы вычисления расстояния Левенштейна рекурсивно, рекурсивно с использованием матрицы расстояний, итеративно с использованием матрицы расстояний, а также алгоритм вычисления расстояния Дамерау-Левенштейна итеративно с использованием матрицы были реализованы и протестированы.

4 | Исследовательская часть

4.1 Пример работы программного обеспечения

4.2 Технические характеристики

Технические характеристики ЭВМ, на котором выполнялись исследования:

- ОС: Manjaro Linux 20.1.1 Mikah
- Оперативная память: 16 Гб
- Процессор: Intel Core i7-10510U

  При проведении замеров времени ноутбук был подключен к сети электропитания.

4.3 Время выполнения алгоритмов

Алгоритмы тестировались на данных, сгенерированных случайным образом один раз.

Тестовые данные:

- 5 символов:
  VxgtU (строка 1),
  jRMFA (строка 2)
- 7 символов:
  VxgtUsx (строка 1),
  jRMFAyC (строка 2)

- 10 символов:
  VxgtUsx2u3 (строка 1),
  jRMFAyCfiV (строка 2)

- 20 символов:
  VxgtUsx2u39dtX81sxy8 (строка 1),
  jRMFAyCfiVxyhmILtGMG (строка 2)

- 30 символов:
  VxgtUsx2u39dtX81sxy8GInrYeVNmJ (строка 1),
  jRMFAyCfiVxyhmILtGMG4IVZTjPQ7l (строка 2)

- 50 символов:
  VxgtUsx2u39dtX81sxy8GInrYeVNmJvvG7WkaA7Qjs82qP6bJG (строка 1),
  jRMFAyCfiVxyhmILtGMG4IVZTjPQ7laMIEG6xv9zbdXq9WcJY2 (строка 2)

- 100 символов:
  VxgtUsx2u39dtX81sxy8GInrYeVNmJvvG7WkaA7Qjs82qP6bJG Ooryez5fYpJWcPRhm7TEjeUoD49M26XDt CJrGtjJXf3aZ9La9n (строка 1),
  jRMFAyCfiVxyhmILtGMG4IVZTjPQ7laMIEG6xv9zbdXq9WcJY2 G4J0JV1XP8ecmHkTYdY1uzSm8WFY3KjgG ggAw3GrPISl76Mzb1 (строка 2)

- 200 символов:
  VxgtUsx2u39dtX81sxy8GInrYeVNmJvvG7WkaA7Qjs82qP6bJGO oryez5fYpJWcPRhm7TEjeUoD49M26XDtCJrGtjJXf3aZ9La9nsh v3cAbwuAJuKc00ndp6EWNHQcArjwXQzAtdpnHs2uOF1kfhWjzXU S44zKnHVNCaeLyzBlce3RCdGwbJx8s2SlfvYoyBZsKrN1cX (строка 1),
  jRMFAyCfiVxyhmILtGMG4IVZTjPQ7laMIEG6xv9zbdXq9WcJY2G 4J0JV1XP8ecmHkTYdY1uzSm8WFY3KjgGggAw3GrPISl76Mzb1f3 ElDEyOeorQGS6CxLWS3lH8sNgZta9vSDMLvnbPaXP24H5dYkBXL RruvzSlLs1T8hyezy0U3awz65ctATEclCBG4H1pC9mMusWF (строка 2)

Результаты замеров времени приведены в таблице 4.1. На рисунках 4.1, 4.2 приведены графики зависимостей времени работы алгоритмов от //.

Таблица 4.1: Замеры времени для строк различной длины

Длина строк	LevRec	LevMatRec	LevMatIter	DamLev
5	10867	-	-	-
7	258961	-	-	-
10	33589820	3146	2001	2137
20	-	12896	4686	6251
30	-	29325	10744	13631
50	-	70918	29277	38427
100	-	184238	86268	118891
200	-	642895	248651	299743

4.4 Оценка затрат памяти

Максимальная глубина стека вызовов при исполнении рекурсивного алгоритма Левенштейна определяется выражением 4.1:

$$(sizeof(s_1) + sizeof(s_2)) * (2 * sizeof(string) + sizeof(int)) \tag{4.1}$$

Здесь sizeof - оператор вычисления размера; s_1, s_2 - строки; $string$ - строковый тип; int - целочисленный тип.

При исполнении интеративной реализации задействованная память будет определяться выражением 4.2:

$$(sizeof(s_1+1)*(sizeof(s_2+1)*sizeof(int)+sizeof(int)+2*sizeof(string) \tag{4.2}$$

Вывод

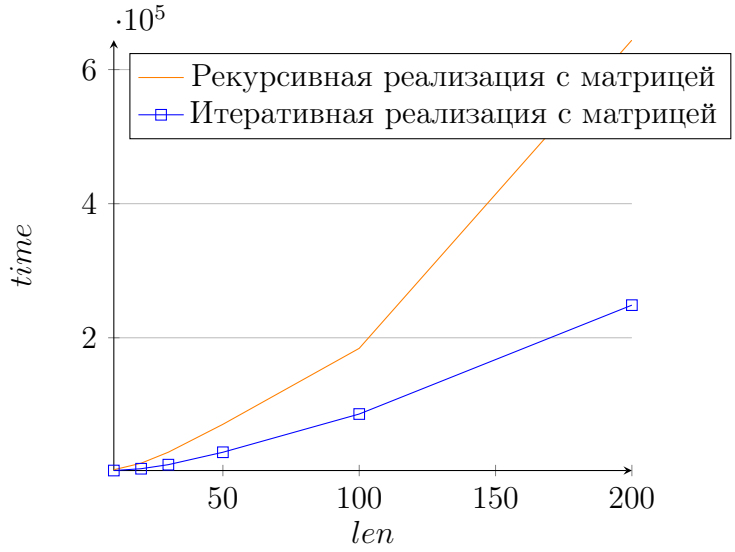

Рис. 4.1: Зависимость времени работы рекурсивных реализаций алгоритмов вычисления расстояния Левенштейна от длины строк

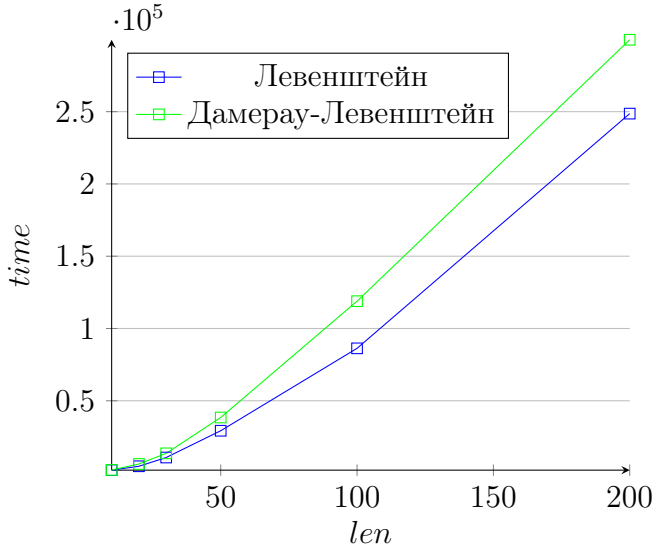

Рис. 4.2: Зависимость времени работы итеративных реализаций алгоритмов вычисления расстояния Левенштейна и Дамерау-Левенштейна от длины строк.

Заключение

В ходе выполнения лабораторной работы:

-
-
-
-
-
- Были получены практические навыки реализации алгоритмов на ЯП Kotlin.

Литература

[1] Coppersmith D., Winograd S. Matrix multiplication via arithmetic progressions // Journal of Symbolic Computation. 1990. no. 9. P. 251–280.

[2] Robinson S. Toward an Optimal Algorithm for Matrix Multiplication // SIAM News. 2005. November. Vol. 38, no. 9.

[3] Погорелов Дмитрий Александрович Таразанов Артемий Михайлович Волкова Лилия Леонидовна. Оптимизация классического алгоритма Винограда для перемножения матриц // Журнал №1. 2019. Т. 49.

[4] Kotlin language specification [Электронный ресурс]. Режим доступа: `https://kotlinlang.org/spec/introduction.html` (дата обращения 09.10.2020.