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 №1 по дисциплине "Анализ алгоритмов"

Тема Расстояние Левенштейна и Дамерау-Левенштейна

Студент Байрамгалин Я.Р.

Группа ИУ7-53Б

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

Москва — 2022 г.

Оглавление

Введение

Расстояния Левенштейна и Дамерау-Левенштейна применяются в таких сферах, как:

- компьютерная лингвистика (автозамена в посиковых запросах, текстовая редактура);
- биоинформатика (последовательности белков);
- нечеткий поиск записей в базах (борьба с мошенниками и опечатками);
- сравнения текстовых файлов утилитой `diff` и ей подобными (здесь роль «символов» играют строки, а роль «строк» — файлы);
- сравнения генов, хромосом и белков в биоинформатике.

Задачами данной лабораторной являются:

1. построение алгоритмов Левенштейна и Дамерау-Левенштейна;
2. сравнение времени выполнения различный версий алгоритма Дамерау-Левенштейна, а именно нерекурсивного, рекурсивного, рекурсивного с кешированием;
3. составление отчета о проделанной работе.

Целью настойщей лабораторной работы является исследование быстродействия алгоритмов Левенштейна и Дамерау-Левенштейна.

1 Аналитическая часть

Расстоянием Левенштейна для двух строк a, b называют число d, такое что

$$d(|a|, |b|) = \begin{cases} 0, & \text{если i = 0, j = 0;} \\ i, & \text{если j = 0, i > 0;} \\ j, & \text{если i = 0, j > 0;} \\ \min(d(i-1, j) + 1, & \\ \qquad d(i, j-1) + 1, & \text{иначе.} \\ \qquad d(i-1, j-1) + X(a[i], b[j]) & \\ ) & \end{cases} \tag{1.1}$$

где $|m|$ — длина строки m; X(a[i], b[j]) — равно 0, если $a[i] = b[j]$, 0 иначе.

В формуле 1.1 выражение

$$d(i-1, j) + 1 \tag{1.2}$$

соответствует операции удаления j-го символа из строки a; выражение

$$d(i, j-1) + 1 \tag{1.3}$$

соответствует операции добавления i-го символа к строке a; выражение

$$d(i-1, j-1) + X(a[i], b[j]) \tag{1.4}$$

соответствует операцие замены i-го символа строки a на j-ый символ строки b, если такая замена требуется (символы различны).

Расстоянием Дамерау-Левенштейна для двух строк a, b называют число d, такое что

$$d_{a,b}(i,j) = \begin{cases} \max(i,j), & \text{если } \min(i,j) = 0; \\ \min(& \\ \qquad d_{a,b}(i,j-1)+1, & \text{если i, j} > 0 \\ \qquad d_{a,b}(i-1,j)+1, & \text{и } a[i] = b[j-1] \\ \qquad d_{a,b}(i-1,j-1)+X(a[i],b[j]), & \text{и a}[i-1] = b[j]; \\ \qquad d_{a,b}(i-2,j-2)+1 & \\ ) & \\ \min(& \\ \qquad d_{a,b}(i,j-1)+1, & \\ \qquad d_{a,b}(i-1,j)+1, & \text{иначе.} \\ \qquad d_{a,b}(i-1,j-1)+X(a[i],b[j]) & \\ ) & \end{cases}, \tag{1.5}$$

1.1 Рекурсивный алгоритм Дамерау — Левенштейна

Рекурсивный алгоритм по своей сути повторяет определение из формулы 1.5. Он является крайне неэффективным, и работает неприемлимо долго даже на относительно небольших строках (12-15) символов.

1.2 Рекурсивный алгоритм Дамерау — Левенштейна с кешированием

Идея использования кеширования в рекурсивном влгоритме Дамерау-Левенштейна состоит в том, чтобы запоминать уже подсчитанные значения $d(i, j)$. Для этого необходимо создать матрицу M размера i, j, заполнить ее, например, значение -1. После вычисления очередного $d(i, j)$ стоит поместить его значение в $M_i j$.

1.3 Матричный алгоритм нахождения расстояния Дамерау — Левенштейна

При больших i, j прямая реализация формулы 1.1 может быть малоэффективна по времени исполнения, так как множество промежуточных значения $D(i, j)$ вычисляются не по одному разу. Для оптимизации нахождения можно использовать матрицу для хранения соответствующих промежуточных значений.

Матрица размером $(length(S1) + 1)$x$((length(S2) + 1)$, где $length(S)$ — длина строки S. Значение в ячейке $[i, j]$ равно значению $D(S1[1...i], S2[1...j])$. Первая строка и первый столбец тривиальны.

Всю таблицу (за исключением первого столбца и первой строки) заполняем в соответствии с формулой 1.6.

$$A[i][j] = min \begin{cases} A[i-1][j] + 1 \\ A[i][j-1] + 1 \\ A[i-1][j-1] + X(S1[i], S2[j]) \end{cases} . \quad (1.6)$$

В результате расстоянием Левенштейна будет ячейка матрицы с индексами $i = length(S1)$ и $j = length(S2)$.

Вывод

Рассмотрены теоретические аспекты алгоритмов Левенштейна и Дамерау — Левештейна. Получены достаточные знания для программной реализации.

2 Конструкторская часть

Схемы алгоритмов

На рисунках 2.1 - 2.3 изображены схема алгоритма матричного алгоритма Левенштейна, схема рекурсивного алгоритма Дамерау-Левенштейна, схема рекурсивного алгоритма Дамерау-Левенштейна с кешированием соответственно.

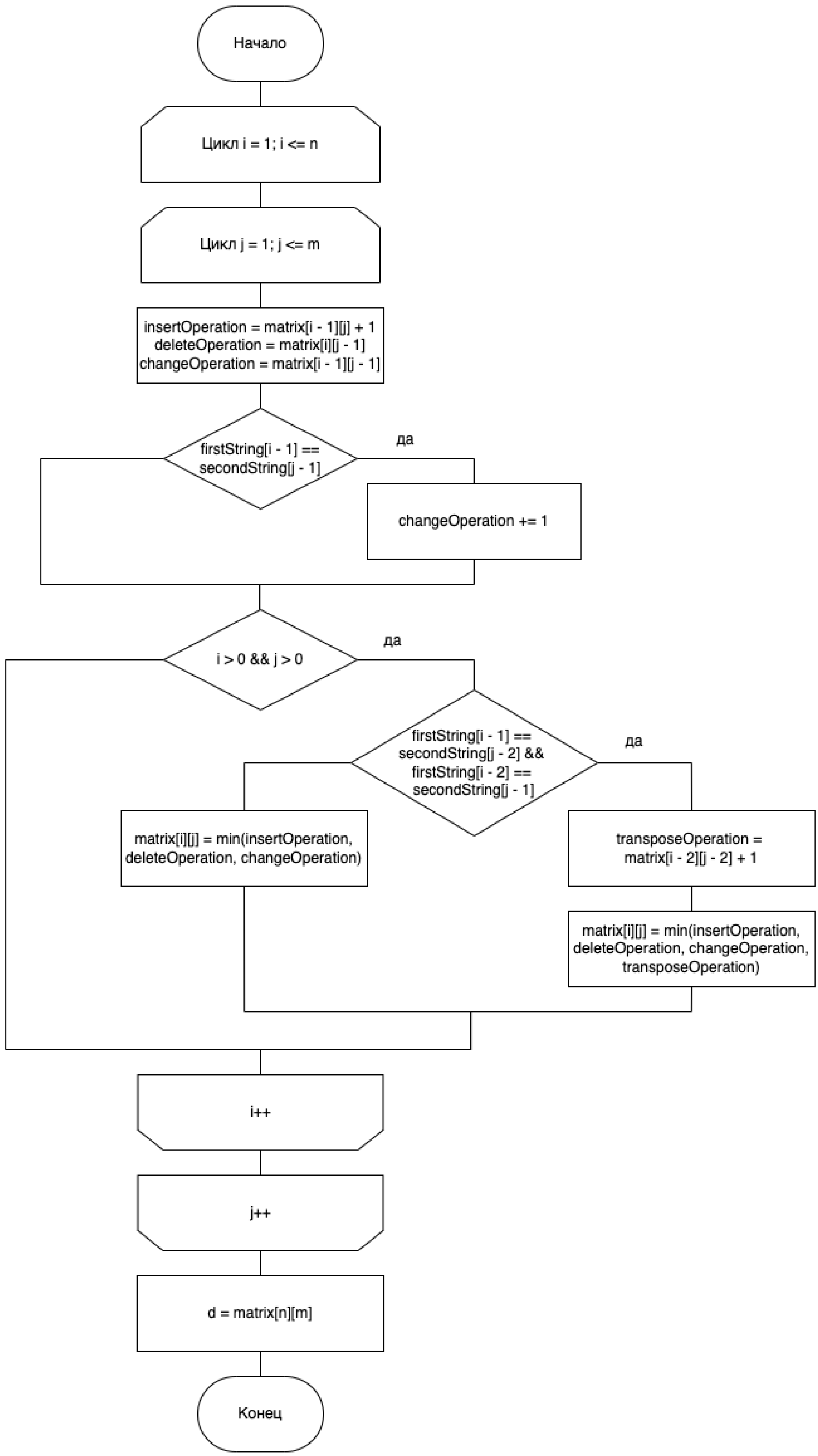

Рис. 2.1: Схема стандартного алгоритма умножения матриц

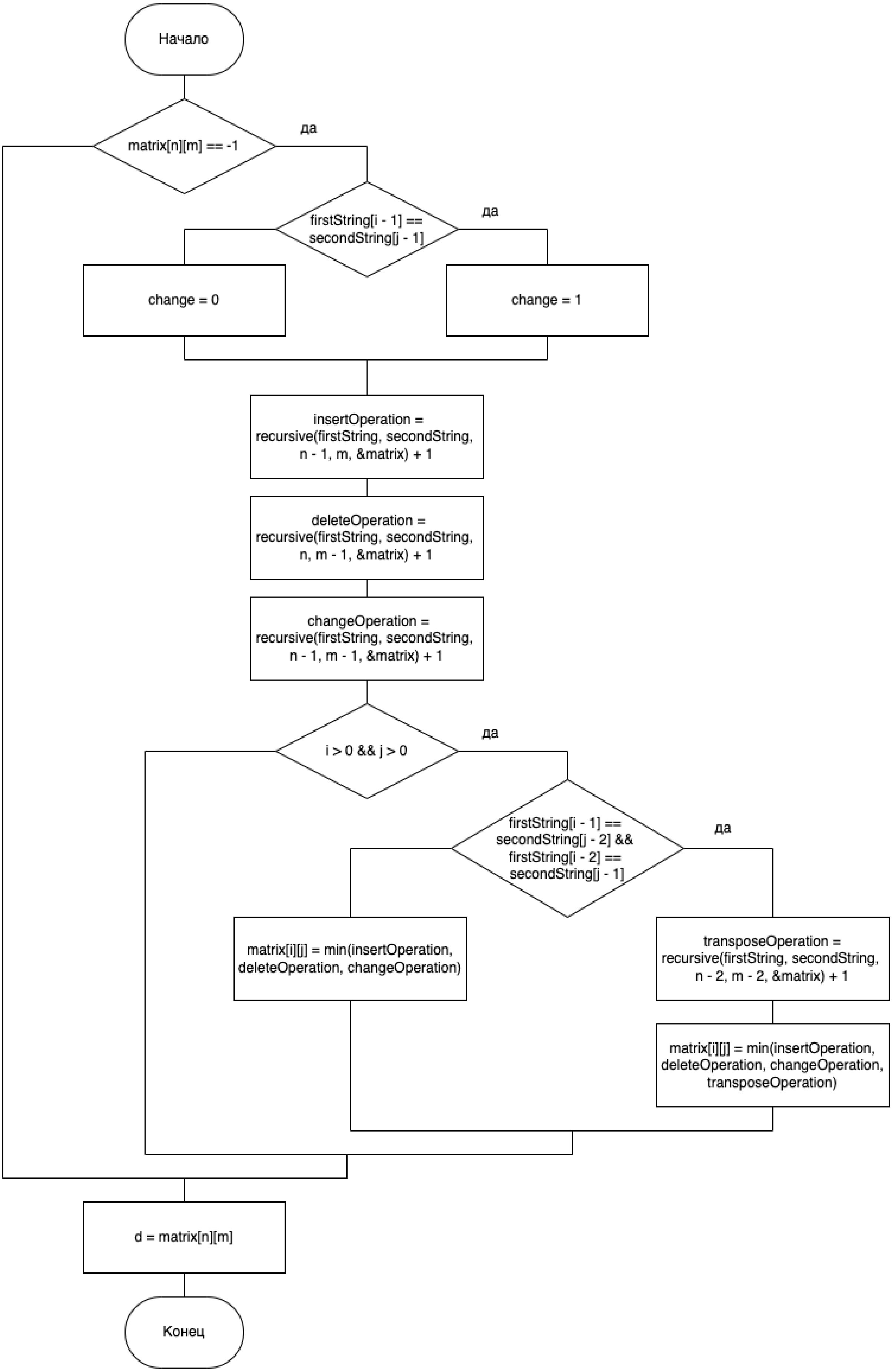

Рис. 2.2: Схема алгоритма Винограда

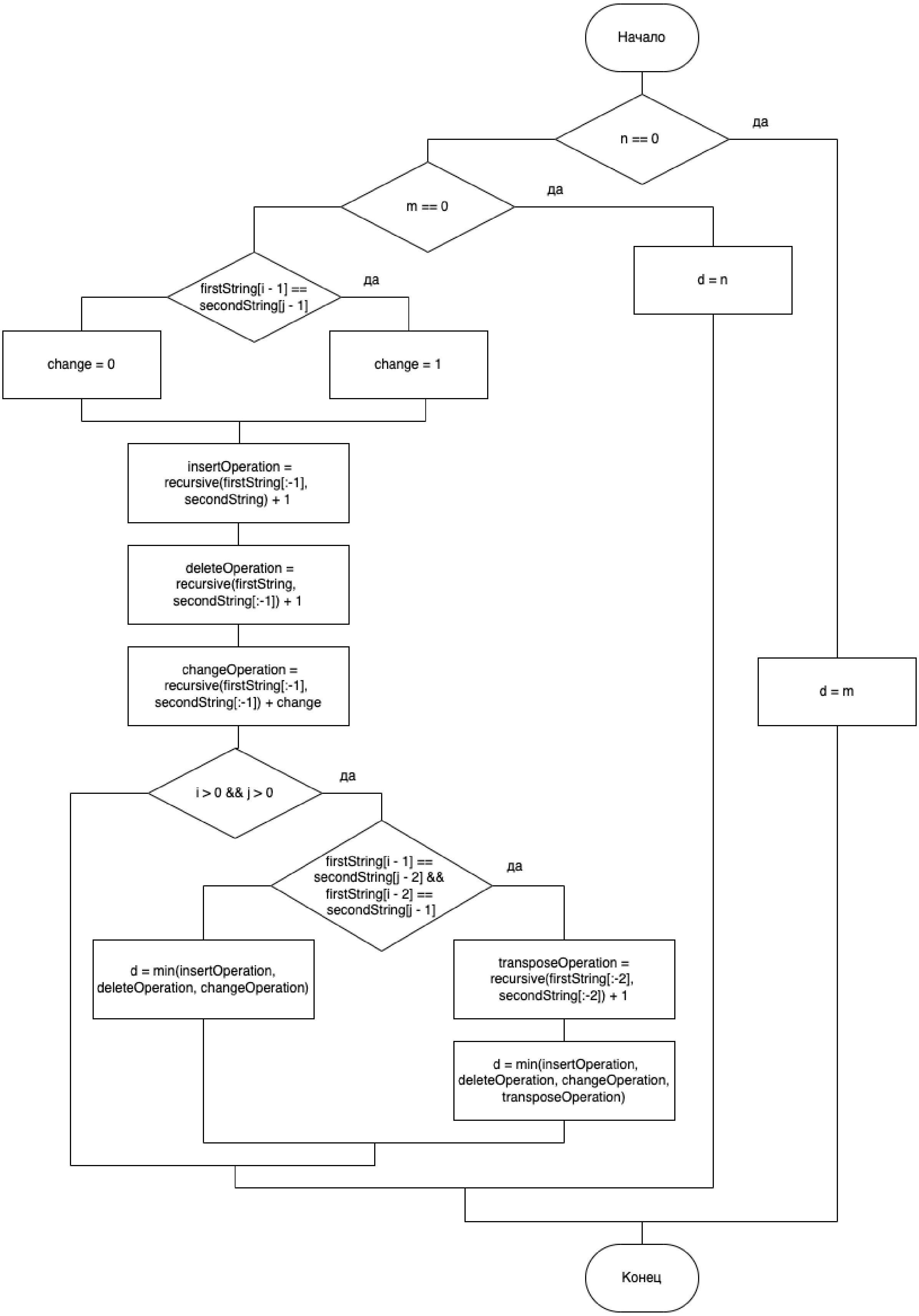

Рис. 2.3: Схема функций оптимизированного алгоритма Винограда

3 Технологическая часть

В данном разделе будут приведены требования к программному обеспечению, используемые технологии и реализации алгоритмов.

3.1 Требования к программному обеспечению

К программе предъявляется ряд требований:

- на вход подаются 2 произвольные строки ASCII символов;
- в случае успешного выполнения возвращается одно целое число — расстояние Дамерау-Левенштейна;
- в случае если по какой-либо причине не удалось найти расстояние Дамерау-Левенштейна — вывести ошибку.

3.2 Выбор средств реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран язые программирования C++[1].

Язык позволяет управлять всеми ресурсами компьютера и, тем самым позволяет писать эффективные алгоритмы.

Время работы алгоритмов было замерено с помощью функции из листинга 3.1, разработанной самостоятельно.

```cpp
template <typename F, typename... Args>
auto GetExecutionTime(F function, Args&&... args) {
  const auto start_time = std::chrono::high_resolution_clock::now();
  const auto return_value = function(std::forward<Args>(args)...);
  const auto end_time = std::chrono::high_resolution_clock::now();
  return std::pair(
      std::chrono::duration_cast<TimeUnit>(end_time - start_time),
      return_value);
}
```

Листинг 3.1: Функция для замера времени исполнения функции

3.3 Реализация алгоритмов

В листингах 3.2, 3.3, 3.4 представлены реализации матричного алгоритма Дамерау-Левенштейна, рекурсивного алгоритма Дамерау-Левенштейна, рекурсивного алгоритма Дамерау-Левенштейна с кешированием.

```cpp
int DamerauLevenshteinDistance(
    const std::string& first, const std::string& second) {
  using Row = std::vector<int>;
  using Matrix = std::vector<Row>;

  Matrix distances(first.size() + 1, Row(second.size() + 1));

  for (size_t i = 0; i < distances.size(); ++i) {
    distances[i][0] = int(i);
  }
  for (size_t j = 0; j < distances[0].size(); ++j) {
    distances[0][j] = int(j);
  }
  for (size_t i = 1; i < distances.size(); ++i) {
    for (size_t j = 1; j < distances[0].size(); ++j) {
      int cost = 1;
      if (first[i - 1] == second[i - 1]) {
        cost = 0;
      }
      distances[i][j] = std::min({
        distances[i - 1][j] + 1,
        distances[i][j - 1] + 1,
        distances[i - 1][j - 1] + cost});

      if (i > 1 && j > 1 &&
      first[i - 1] == second[j - 2] && first[i - 2] == second[j - 1]) {
```

```
        distances[i][j] = std::min({
          distances[i][j], distances[i - 2][j - 2] + cost});
      }
    }
  }

  return distances[first.size()][second.size()];
}
```

Листинг 3.2: Матричный алгоритм Дамерау-Левенштейна

```
int DamerauLevenshteinDistanceRecursive(
const std::string& first, const std::string& second) {
  if (first.empty()) {
    return int(second.size());
  }
  if (second.empty()) {
    return int(first.size());
  }

  std::function<int(int, int)> calculate_distance_recursively;
  calculate_distance_recursively = [&](int i, int j) -> int {
    if (i == 0 && j == 0) {
      return (first[i] != second[j]) ? 1 : 0;
    }
    if (i == 0) {
      return int(j);
    }
    if (j == 0) {
      return int(i);
    }

    int cost = (first[i] != second[j]) ? 1 : 0;

    int delete_operation_cost = calculate_distance_recursively(i - 1, j) +
    1;
    int insert_operation_cost = calculate_distance_recursively(i, j - 1) +
    1;
    int correspondence_operation_cost =
    calculate_distance_recursively(i - 1, j - 1) + cost;
    if (i > 1 && j > 1 &&
    first[i] == second[j - 1] && first[i - 1] == second[j]) {
      int swap_operation_cost =
      calculate_distance_recursively(i - 2, j - 2) + 1;
      return std::min({
        delete_operation_cost, insert_operation_cost,
        correspondence_operation_cost, swap_operation_cost});
    }

```

```
    return std::min({
      delete_operation_cost, insert_operation_cost,
      correspondence_operation_cost});
  };

  return calculate_distance_recursively(
  int(first.size() - 1), int(second.size() - 1));
}
```

Листинг 3.3: Рекурсивный алгоритм Дамерау-Левенштейна

```
int DamerauLevenshteinDistanceRecursiveCached(
const std::string& first, const std::string& second) {
  using Row = std::vector<std::optional<int>>;
  using Matrix = std::vector<Row>;

  if (first.empty()) {
    return int(second.size());
  }
  if (second.empty()) {
    return int(first.size());
  }

  Matrix calculated_distances(first.size() + 1, Row(second.size() + 1));
  std::function<int(int, int)> calculate_distance_recursively;
  calculate_distance_recursively = [&](int i, int j) -> int {
    if (i == 0 && j == 0) {
      return (first[i] != second[j]) ? 1 : 0;
    }
    if (i == 0) {
      return int(j);
    }
    if (j == 0) {
      return int(i);
    }
    if (calculated_distances[i][j].has_value()) {
      return calculated_distances[i][j].value();
    }

    int cost = (first[i] != second[j]) ? 1 : 0;

    int delete_operation_cost = calculate_distance_recursively(i - 1, j) +
    1;
    int insert_operation_cost = calculate_distance_recursively(i, j - 1) +
    1;
    int correspondence_operation_cost =
    calculate_distance_recursively(i - 1, j - 1) + cost;
    if (i > 1 && j > 1 &&
    first[i] == second[j - 1] && first[i - 1] == second[j]) {
```

```
      int swap_operation_cost =
      calculate_distance_recursively(i - 2, j - 2) + 1;
      int result = std::min({
        delete_operation_cost, insert_operation_cost,
        correspondence_operation_cost, swap_operation_cost});
      calculated_distances[i][j] = result;
      return result;
    }

    int result = std::min({
      delete_operation_cost, insert_operation_cost,
      correspondence_operation_cost});
    calculated_distances[i][j] = result;

    return result;
  };

  return calculate_distance_recursively(
  int(first.size() - 1), int(second.size() - 1));
}
```

Листинг 3.4: Рекурсивный алгоритм Дамерау-Левенштейна с кешированием

3.4 Тестрование

В таблице 3.1 рассмотрены случаи ожидаемого поведения программы. Все тесты были успешно пройдены.

Таблица 3.1: Функциональные тесты

Строка 1	Строка 2	Ожидаемый результат
«пустая строка»	«пустая строка»	0
cat	hat	1
apple	aplpe	1
qwerty	queue	4
wolf	wolf	0
word	«пустая строка»	4
mitsubishi	mercedes-benz	11

Вывод

Разработаны и тестированы 3 алгоритма. Все алгоритмы соответствуют заявленным требованиям.

4 Исследовательская часть

В данном разделе будет приведен равнительный анализ алгоритмов на основе полученных данных.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование, следующие.

- Операционная система: macOs Monterey 12.4[2].
- Память: 16 Гбайт.
- Процессор: 2,6 ГГц 6-ядерный процессор Intel Core i7[3].

Тестирование проводилось на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, а также непосредственно системой тестирования.

4.2 Время выполнения алгоритмов

В таблице 4.1 представлены замеры времени работы алгоритмов. Здесь и далее: М — матричный алгоритм, Р — рекурсивный алгоритм, РК – рекурсивный алгоритм с кешем. Время в микросекундах. Прочерк «—» означает, что время выполнения алгоритма слишком велико.

Таблица 4.1: Результаты замеров времени алгоритмов (миллисекунды)

Строка 1	Строка 2	М	Р	РК
Similar	Similar	19	1075	23
kjFFkjRhtZ	ZFdbYi4nQd	21	17042	59
words diff	wrods dfif	20	139389	52
Two similar sentences	Two similar sentences	57	—	113
NVyhNfPnYykGwDZETiI5	GD34eG0rIZ73qGorddY5	50	—	94
some real words with errors	smoe real wrods wiht erors	84	—	66

Вывод

При увеличении размера матриц, увеличивается и эффективность работы алгоритма Винограда в сравнении со стандартным алгоритмом. В среднем реализация по Винограду работает быстрее в 1.7 раз. Оптимизированная реализация алгоритма Винограда также позволяет уменьшить время работы при больших размерностях: например, для размерности матрицы, равной 200, эксперимент показал, что алгоритм Винограда быстрее стандартного алгоритма чуть менее, чем в 2 раза, когда оптимизированная версия выигрывает у обычного умножения более, чем в 2 раза.

Заключение

При увеличении размера матриц, увеличивается и эффективность работы алгоритма Винограда в сравнении со стандартным алгоритмом. В среднем реализация по Винограду работает быстрее в 1.7 раз. Оптимизированная реализация алгоритма Винограда также позволяет уменьшить время работы при больших размерностях: например, для размерности матрицы, равной 200, эксперимент показал, что алгоритм Винограда быстрее стандартного алгоритма чуть менее, чем в 2 раза, когда оптимизированная версия выигрывает у обычного умножения более, чем в 2 раза.

В ходе выполнения лабораторной работы были решены следующие задачи:

- были реализованы 3 алгоритма Дамерау-Левенштейна: матричный, рекурсивный, рекурсивный с кешем;
- был произведен анализ времемени выполнения разработанных алгоритмов;
- подготовлен отчет о лабораторной работе.

Список использованных источников

[1] C++ language [Электронный ресурс]. Режим доступа: https://en.cppreference.com/w/cpp/language (дата обращения: 04.10.2022).

[2] macos Monterey 12.03 [Электронный ресурс]. Режим доступа: https://www.apple.com/ru/macos/monterey/ (дата обращения: 04.10.2022).

[3] Процессор Intel® Core™ i5-1135G7 [Электронный ресурс]. Режим доступа: https://www.intel.ru/content/www/ru/ru/products/sku/208658/intel-core-i51135g7-processor-8m-cache-up-to-4-20-ghz/specifications.html (дата обращения: 04.10.2022).