



МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА

Институт искусственного интеллекта

Кафедра высшей математики

КУРСОВАЯ РАБОТА
по дисциплине
«Объектно-ориентированное программирование»

Тема курсовой работы
«Использование хеширования в задачах поиска в строке»

Студент группы КМБО-07-22

Невский В.Е.

Руководитель курсовой работы
доцент кафедры Высшей математики
к.т.н

Парфёнов Д.В.


Работа представлена к
защите

«19» *декабря* 20 *23* г.


(подпись студента)

«Допущен к защите»

«19» *декабря* 20 *23* г.


(подпись руководителя)

Курсовая работа по дисциплине ООП на тему
«Исследование квантования в задачах поиска в
строении» выполнена студ. Невежухи В.Е. с старшим
наставником. Оценка «отлично».

Проф. Парфенов Л.В.

МОСКВА — 2023



МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА

Институт искусственного интеллекта

Кафедра высшей математики

Утверждаю

Исполняющий обязанности заведующего
кафедрой А.В.Шатина А.В.Шатина

«22» сентября 2023 г.

ЗАДАНИЕ
на выполнение курсовой работы
по дисциплине «Объектно-ориентированное программирование»

Студент *Невский В.Е.*

Группа *КМБО-07-22*

1. Тема: «Использование хеширования в задачах поиска в строке»

2. Исходные данные:

Для поиска в тексте необходимо применить реализации алгоритма на основе хеширования.
Сравнить скорость работы алгоритма с:

- Алгоритмом Бойера – Мура и его модификациями;
- Алгоритмом Кнута-Морриса-Пратта;

Реализовать процесс нечёткого поиска

Ввести поиск по проценту совпадения

Ввести функцию сортировки результатов по проценту совпадения

3. Перечень вопросов, подлежащих разработке, и обязательного графического материала:

Проиллюстрировать работу реализаций алгоритма привести оценки их производительности.
При выводе информации о результатах поиска либо подсветить фрагмент текста, внутри которого найдено слово (несколько слов перед и после найденного); либо вывести этот фрагмент

При выводе информации о результатах нечёткого поиска, выводить процент совпадения

4. Срок представления к защите курсовой работы: до «22» декабря 2023 г.

Задание на курсовую
работу выдал

«22» сентября 2023 г.



(Парфёнов Д.В.)

Задание на курсовую
работу получил

«22» сентября 2023 г.



(Невский В.Е.)

Оглавление

Глава 1. Теория.....	3
Постановка задачи.	3
Хеши. Основные понятия. Коллизии и способы их решения.	3
Выбор хеш-функции для поиска.	5
Описание алгоритма поиска на основе хеширования.	6
Алгоритмы-конкуренты.	7
Неточный поиск. Расстояние Дameraу-Левенштейна.	10
Алгоритм неточного поиска.	13
Глава 2. Реализация.....	14
Поиск с хешированием.	14
Алгоритм Бойера-Мура.	17
Алгоритм Кнута-Морриса-Пратта.	18
Алгоритм неточного поиска.	19
Сравнение эффективности алгоритмов.	22
Заключение.....	28
Приложения.....	29
Список литературы	41

Постановка задачи.

Основными задачами данной работы являются:

- 1) Реализация точного поиска в тексте на основе хеширования.
- 2) Сравнение работы алгоритма с конкурентами.
- 3) Реализация неточного поиска в тексте с указанными процентом совпадения.

Хеши. Основные понятия. Коллизии и способы их решения.

Для начала введём основные понятия о том, что такое хеши.

Хеш-функция — это некая функция, которая сопоставляет объектам какого-то множества числовые значения из промежутка.

Преобразование, производимое хеш-функцией, называется хэшированием. Результат хеширования и есть хеш.

Рассмотрим на небольшом примере:

Предположим, у нас есть список преподавателей потока КМБ-22. Нашей задачей является присвоение уникального номера каждому преподавателю. Придумаем простую хеш-функцию, которая берёт первую букву имени и первую букву фамилии, складывает их числовые значения в таблице ASCII (для удобства будем записывать имена и фамилии преподавателей на латинице) и выдаёт результат. Принцип и результат применения нашей хеш-функции можем увидеть на рисунке 1.

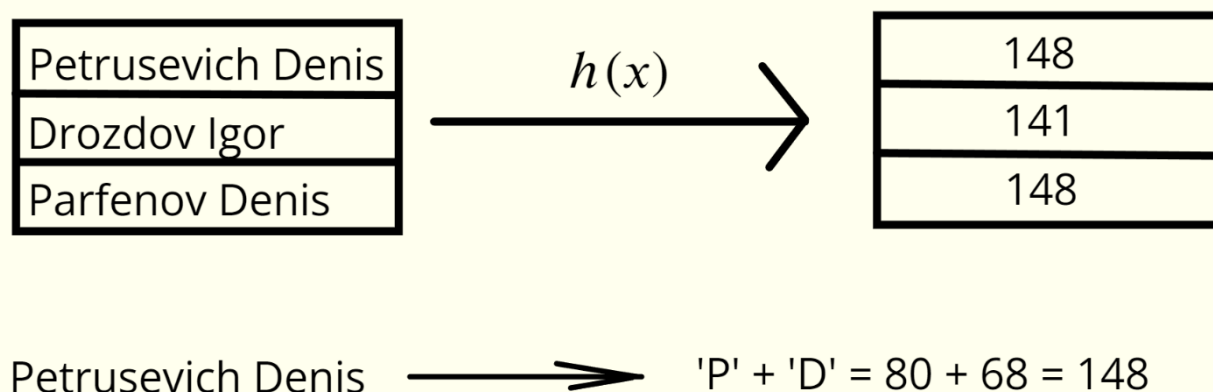


Рисунок 1. Принцип работы хеш-функции на примере списка преподавателей.

Диапазон значений, в которые наша хеш-функция обращает имена преподавателей нетрудно вычислить: самое маленькое значение будет $'A' + 'A' = 130$, а самое большое $'Z' + 'Z' = 180$.

Плюсы такой хеш-функции:

- 1) Несложная формула \Rightarrow не требует больших вычислений.
- 2) Легкая в придумывании и запоминании.

Минусы:

- 1) Маленький диапазон значений.
- 2) Высокая вероятность возникновения коллизий.

Коллизия — сопоставление двум объектам из входного множества одного значения из числового диапазона.

В нашем примере у нас возникла коллизия: преподаватели Petrusevich Denis и Parfenov Denis имеют одинаковое значение.

Как решить данную проблему?

- 1) Изменить хеш-функцию на “хорошую”.
- 2) Применить метод цепочек.
- 3) Использовать метод открытой адресации.

Рассмотрим решения проблемы поочерёдно.

1.

Что мы понимаем под “хорошей” хеш-функцией?

“Хорошая” хеш-функция — хеш-функция, которая является инъективным отображением набора данных на числовое множество, т.е. в которой отсутствуют коллизии.

Подобрать такую функцию можно только исходя из набора данных: на каких-то наборах данных эта хеш-функция себя ведёт хорошо, на каких-то плохо.

К сожалению, не существует идеальных хеш-функций. Если мы добавим большое количество данных, то коллизии обязательно возникнут. Это объясняется парадоксом дней рождений: в мультимножество нужно добавить $\Theta(\sqrt{n})$ случайных чисел от 1 до n , чтобы какие-то два совпали. Таким образом, получаем, что любую хеш-функцию можно сделать “плохой” просто добавив большое количество данных.

2.

Метод цепочек представляет из себя хранение списка ключей для каждого значения. То есть при возникновении коллизий, мы просто будем добавлять

ключи в один список, а затем, при поиске, будем вычислять хеш и проходить по списку в поисках нужного нам значения.

Этот метод чаще всего применяется для разрешения коллизий.

3.

Метод открытой адресации предполагает, что при возникновении коллизий, мы будем искать следующее значение, которое ещё не занято и записывать туда наш ключ. В нашем случае значение для Parfenov Denis было бы присвоено следующему свободному, т.е. 149. Последовательность значений, которая будет просматриваться называется последовательностью проб. Очевидно, что если мы добавим $n + 1$ объект, то наша хеш-функция сломается.

Выбор хеш-функции для поиска.

Теперь, познакомившись с основными понятиями, можно составить алгоритм, который будет делать поиск в тексте с помощью наших хешей.

В качестве основы будем использовать полиномиальное хеширование, т.е. наша хеш-функция будет представлять из себя многочлен.

Пусть у нас есть текст, состоящий из n символов. Тогда хеш-функция будет иметь вид:

$$h(text) = (a_1p^{n-1} + a_2p^{n-2} + \dots + a_{n-1}p + a_n) \% q \quad (1)$$

Здесь q — некоторый большой модуль, который задаёт диапазон наших значений (поскольку взятие по модулю q даёт нам q различных элементов от 0 до $q-1$),

p — простой модуль, $|a_i| < p < q$,

a_i — i -й символ исходного текста, $i = 1, 2, \dots, n$.

Но вычислять хеш каждый раз по такой формуле было бы очень затратным и бессмысленным. Поэтому мы заранее предпосчитаем хеши для префиксов длины 1, 2, ..., n и сможем вычислять хеш любой подстроки нашего текста.

Разберём на примере:

Пусть дана строка $s = \text{“abcde”}$, мы хотим посчитать хеш для подстроки $s_1 = \text{“cde”}$. Как нам это сделать с помощью предпосчитанных префиксов?

По формуле (1) посчитаем хеш для всей строки:

$$h(s) = ap^4 + bp^3 + cp^2 + dp + e,$$

Теперь посчитаем искомый хеш:

$$h(s1) = cp^2 + dp + e,$$

Отнимем $h(s) - h(s1)$:

$$h(s) - h(s1) = ap^4 + bp^3 = p^3(ap + b) = p^3 * h(s - s1),$$

$$h(s1) = h(s) - p^3 * h(s - s1).$$

Мы смогли выразить подстроку через подсчитанный суффикс, теперь подставим в общем виде:

$$h(j - i) = h(j) - p^{j-i+1} * h(i).$$

Теперь мы умеем считать хеш от подстроки любой длины за $\Theta(1)$.

Главное для правильности вычислений подобрать хорошие значения для p , q . Мы будем использовать значение $q = 2^{31} - 1$, поскольку взятие модуля по данному числу эквивалентно количеству единиц в битовой записи целых чисел. Значение p будет выбираться случайно из диапазона $[\frac{n}{2}; n - 2]$. При таких подобранных p и q вероятность коллизий практически нулевая, а также будет сложно подобрать входные данные, которые бы заставили нашу хеш-функцию плохо работать.

Описание алгоритма поиска на основе хеширования.

Пусть нам дан текст $text$ длины n и подстрока для поиска $pattern$ длины m . Для того, чтобы найти все вхождения $pattern$ в $text$, нам необходимо:

- 1) Предпосчитать все префиксные хеши для $text$.
- 2) Предпосчитать хеш для $pattern$.
- 3) Начать поиск.

Поиск включает в себя:

Проход по всем индексам $i = 1, 2, \dots, n - m + 1$. Для каждого индекса берём подстроку $current$ длины m и считаем для него хеш с помощью префиксов, а затем сравниваем его с хешом для $pattern$. Если они не равны, то продолжаем поиск. Если хеши оказались равными, то здесь есть 2 подхода:

- 1) Сразу добавить i в ответ.
- 2) Сравнить строки $pattern$, $current$.

В первом случае, мы пренебрегаем проверкой равенства строки $pattern$ и $current$. Такой вариант более быстрый, но с небольшим шансом мы можем попасть на коллизию и получить неверный ответ.

Временная сложность для 1 случая $\Theta(n + m)$, поскольку мы делаем предпосчёт префиксов за $\Theta(n)$, подсчёт хеша для pattern за $\Theta(m)$, все остальные вычисления мы делаем за $\Theta(1)$.

Для 2 случая худший случай будет иметь оценку $\Theta(nm)$, поскольку для каждого вхождения подстроки pattern, мы будем делать полную проверку на совпадение найденной строки с искомой.

В конечном итоге для алгоритма потребуется $\Theta(n)$ памяти для хранения префиксов хешей.

Алгоритмы-конкуренты.

Для сравнения работы нашего алгоритма с другими, возьмём следующих кандидатов:

- Алгоритм Бойера-Мура.
- Алгоритм Кнута-Морриса-Прата

Алгоритм Бойера-Мура.

Данный алгоритм считается наиболее быстрым среди алгоритмов общего назначения, предназначенных для поиска в тексте.

Суть: алгоритм сравнивает символы шаблона pattern справа налево, с символами text. Если символы совпали, то производится сравнение предпоследнего символа шаблона и так до конца. Если все символы совпали, то подстрока найдена, и поиск окончен. Если произошло несовпадение, то алгоритм использует одну из двух эвристических функций, чтобы сдвинуть позицию для сравнения вправо. Данные функции называются “эвристика хорошего суффикса” и “эвристика плохого символа”.

Правило сдвига хорошего суффикса.

Если при сравнении совпало один или больше символов, то шаблон сдвигается в зависимости от того, какой суффикс совпал. Если существуют такие подстроки равные u , что они полностью входят в x и идут справа от символов, отличных от $x[i]$, то сдвиг происходит к самой правой из них, отличной от u (См. Рисунок 2).

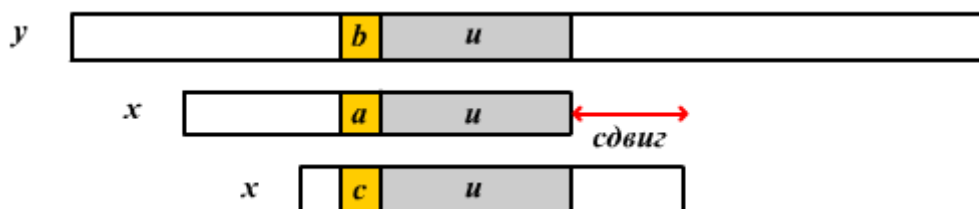


Рисунок 2. Демонстрация работы сдвига для правила хорошего суффикса^[4].

Если не существует таких подстрок, то смещение состоит в выравнивании самого длинного суффикса v подстроки $y[i + j + 1, \dots, j + m - 1]$ с соответствующим префиксом x (См. Рисунок 3).

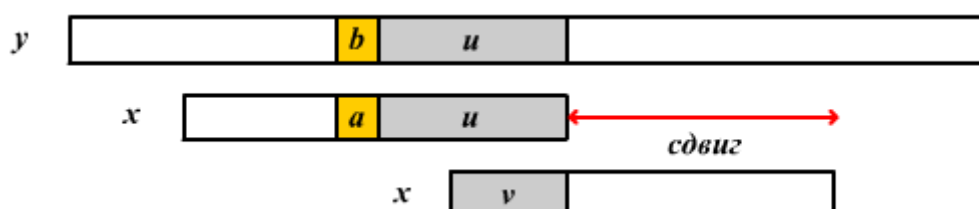


Рисунок 3. Сдвиг при отсутствии подстрок u в x ^[4].

Правило сдвига плохого символа.

В таблице символов указывается последняя позиция в шаблоне (исключая последнюю букву) каждого из символов алфавита. Все символы, которые не вошли в шаблон получают значение m .

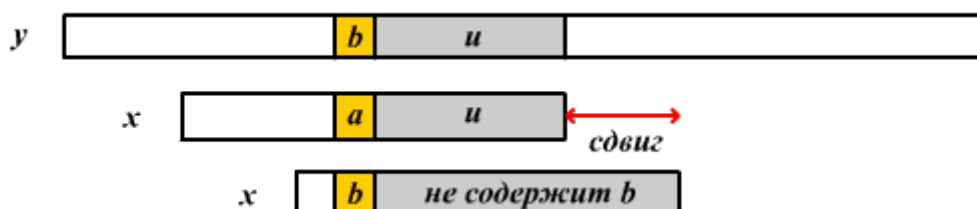


Рисунок 4. Демонстрация сдвига для правила плохого символа^[4].

Выше на рисунке 4 представлена базовая ситуация, когда символ b содержится в x . Если же символ не встречается в x , шаблон двигается полностью (См. Рисунок 5).

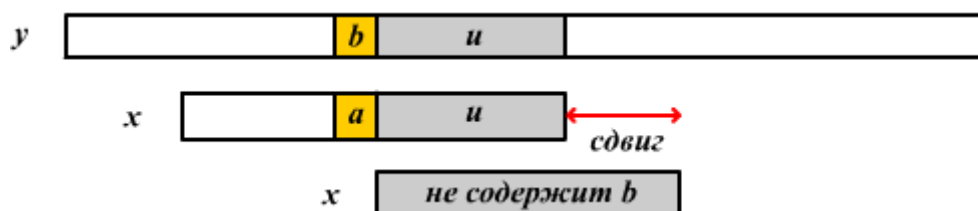


Рисунок 5. Сдвиг для правила плохого символа при отсутствии вхождения b в x ^[4].

В этом и заключается суть всего алгоритма. Но на практике воспользуемся упрощённой версией — алгоритм Бойера-Мура-Хорспула, который использует только эвристику плохого символа и на случайных данных ведёт себя лучше, чем обычная версия алгоритма.

Временная сложность данного алгоритма составляет $\Theta(n + t)$ в лучшем и $\Theta(nt)$ в худшем случае (подстрока встречается на каждой позиции). Затраты по памяти $\Theta(t)$ для хранения таблицы плохих символов.

Алгоритм Кнута-Морриса-Пратта.

Ещё один алгоритм с хорошей скоростью работы и затратами на память. Данный алгоритм построен на следующих убеждениях:

Поэлементно сравниваем образец и текст. Если произошло несовпадение на индексе i , то мы можем предположить, что какой-то префикс для подстроки $[0, i - 1]$ совпадает с его суффиксом, тогда мы можем без потерь передвинуть указатель на длину этого префикса. Т.е. нам необходимо предпосчитать префикс-функцию для образца, а затем двигать указатель на значение префикс-функции для элемента, идущего перед текущим несовпадающим.

Рассмотрим подробнее на примере:

Есть искомая подстрока “abcabcad”. Вычислим для неё таблицу префикс-значений (См. Рисунок 6).

Для подстроки “a” ответ 0.

Для подстроки “ab” тоже 0.

“abc” = 0.

“abc**a**” = 1, т.к. префикс длины 1 совпадает с суффиксом длины 1. “a” = “**a**”.

“abc**ab**” = 2, здесь совпали префикс “ab” и суффикс “ab”.

“abc**abc**” = 3.

“abc**abca**” = 4, стоит заметить, что здесь префикс и суффикс пересекаются и значение 4 является максимальным для данной строки.

“abcab**ca**d” = 0, нет совпадающих суффиксов и префиксов.

Теперь начнём наш поиск. В качестве текста возьмём “abcdaabgfaabca**bcad**”. Изначально указатель стоит на 1 элемент. Проверяем, совпадают ли элементы? Да, идём дальше. На 4 элементе у нас несовпадение, смотрим на значение префикс функции в предыдущем 3 элементе — оно равно 0, значит сдвигаем указатель на 0 элемент и подвигаем 0 элемент к текущему. Далее аналогично ищем совпадения и в самом конце видим, что все элементы совпали и мы нашли полное вхождение. Опять смотрим на значение, лежащее в предыдущем элементе, это 4 => подвигаем всю строку до 4 элемента и так далее.

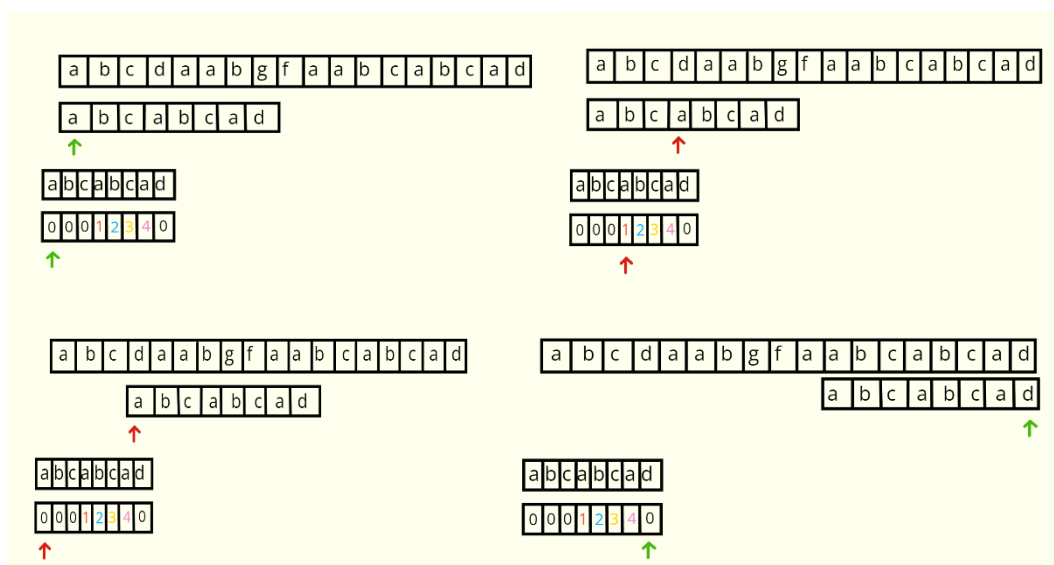


Рисунок 6. Пример работы сдвигов для алгоритма Кнута-Морриса-Пратта.

Скорость работы данного алгоритма $\Theta(n+m)$ для поиска по тексту и вычислений префикс-функции. Потребуется $\Theta(m)$ памяти для хранения вычисленной префикс-функции.

Неточный поиск. Расстояние Дameraу-Левенштейна.

Последняя поставленная задача — реализация алгоритма неточного поиска.

Для неточного поиска нам важно знать процент схожести двух строк. Но что такое схожесть?

Рассмотрим такую метрику, как расстояние Левенштейна. Она определяет минимальное количество односимвольных операций (вставка, удаление, замена), необходимых для превращения одной строки в другую. В общем случае операциям можно назначить стоимость, но у нас по умолчанию стоимость каждой операции равна 1.

Введём некоторые обозначения:

- $w(a, b)$ — стоимость операции замены символа a на символ b .
- $w(\epsilon, b)$ — стоимость операции вставки символа b .
- $w(b, \epsilon)$ — стоимость удаления символа b .

Тогда заметим, что:

- $w(a, a) = 0$,
- $w(a, b) + w(b, c) = w(a, c)$,
- $w(a, b) + w(c, d) = w(c, d) + w(a, b)$,
- $w(\epsilon, a) + w(\epsilon, b) = w(\epsilon, b) + w(\epsilon, a)$,
- $w(a, \epsilon) + w(b, \epsilon) = w(b, \epsilon) + w(a, \epsilon)$,
- $w(a, \epsilon) + w(\epsilon, a)$ — неоптимально (можно обе отменить),
- $w(\epsilon, a) + w(a, b)$ — неоптимально (лишняя замена),
- $w(a, b) + w(b, \epsilon)$ — неоптимально (лишняя замена).

Теперь перейдём непосредственно к формуле вычисления расстояния.

Пусть у нас есть строки $S1$ длины N , $S2$ длины M . Тогда расстояние Левенштейна между этими строками равно $d(S1, S2) = D(M, N)$, где

$$D(i, j) = \begin{cases} 0, i = 0, j = 0 \\ i, j = 0, i > 0 \\ j, i = 0, j > 0 \\ \min \begin{cases} D(i, j - 1) + 1, \\ D(i - 1, j) + 1, \\ D(i - 1, j - 1) + m(S1[i], S2[j]) \end{cases} & j > 0, i > 0 \end{cases},$$

где $m(a, b) = \begin{cases} 0, a = b \\ 1 \end{cases}$.

Здесь шаг по i символизирует удаление из первой строки, по j — вставку в первую строку, а шаг по обоим индексам символизирует замену символа или отсутствие изменений.

Очевидно, что верны следующие утверждения:

- $d(S1, S2) \geq ||S1| - |S2||$,
- $d(S1, S2) \leq \max(|S1|, |S2|)$,
- $d(S1, S2) = 0 \Leftrightarrow S1 = S2$.

(2)

Ниже, в таблице 1 можем увидеть то, как работает наша формула.

Таблица 1. Пример работы алгоритма для вычисления расстояния Левенштейна между строками “Polynomial” и “Exponential”.

		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6

Алгоритм вычисления расстояния Левенштейна требует $\Theta(nm)$ памяти для хранения матрицы, и такую же временную сложность.

Оптимизация: поскольку вычисления производятся динамически, нам достаточно хранить только две строки. Длину строки можно брать как наименьшую из длин строк. Тогда количество занимаемой памяти снижается до $\Theta(\min(n, m))$.

Так как мы решаем задачу поиска в тексте, базовая версия расстояния Левенштейна может оказаться неэффективной. Например, пользователь печатал очень быстро и какие-то буквы поменялись местами. Так из слова “Привет”, пользователь вывел “Пирвет”. Расстояние между такими строками равно 2 (например, удалить символ ‘и’ и вставить его в нужную позицию), что равняется 1/3 от длины строк. То есть если бы мы осуществляли поиск с точностью совпадения выше 66%, то мы бы не сочли данное слово, как похожее, хотя они отличаются только позицией двух соседних символов.

Данную проблему мы можем решить, используя модификацию — расстояние Дамерау-Левенштейна, которое помимо уже имеющихся операций, также рассматривает транспозиции (перестановка двух соседних символов в строке).

Формула для вычисления данного расстояния практически не отличается от формулы для расстояния Левенштейна, мы просто рассматриваем ещё один случай транспозиции.

$$DL(i, j) = \begin{cases} D(i, j), & \min(i, j) = 0 \\ \min \left\{ \begin{array}{l} D(i, j) \\ D(i-2, j-2) + 1 \end{array} \right. & i, j > 1 \text{ и } s1_i = s2_{j-1} \text{ и } s1_{i-1} = s2_j \end{cases}$$

Данный алгоритм имеет ту же асимптотическую сложность, что и алгоритм Левенштейна.

Алгоритм неточного поиска.

Теперь у нас есть всё, чтобы реализовать неточный поиск.

Алгоритм:

- 1) Перебираем индексы i , $i = 1, 2, \dots, n$,
- 2) Для каждого i перебираем длины подстроки $length$,
- 3) Вычисляем расстояние Дамерау-Левенштейна между искомым $pattern$ и $text[i, \dots, i + length - 1]$,
- 4) Если расстояние удовлетворяет точности нашего поиска, смотрим, есть ли пересечение отрезков текста с уже существующим ответом. Если есть, то выбираем лучший, иначе просто добавляем к ответу.

Осталось определиться с тем, какими будут длины наших рассматриваемых подстрок.

Пусть точность поиска равна k , $0 < k < 1$, т.е. если $k = 0.75$, то нас интересуют строки, расстояние между которыми не превышает $m * (1 - k)$.

Отсюда можем сделать вывод, что

$$length \in [m - m * (1 - k), m + m * (1 - k)].$$

Это следует из утверждений (2).

Глава 2. Реализация

Для реализации данных алгоритмов будем использовать C++.

Демонстрация работы кода будет производиться на основе оконного приложения Windows Forms, поэтому в исходном коде каждого алгоритма имеются небольшие изменения для правильности работы с средой C++/CLR.

Каждый из алгоритмов при поиске возвращает вектор из позиций найденных вхождений. Неточный поиск возвращает пару, вторым элементом которого является длина найденного совпадения.

Для правильной работы нижеприведённых алгоритмов, стоит подключить все необходимые для этого библиотеки (см. *Приложение 1*).

Поиск с хешированием.

Создадим класс Hash, который будет производить поиск с помощью хеширования. В качестве полей добавим модуль $q = 2^{31} - 1$, простой модуль p , вектор для префикс-значений и вектор для подсчёта степеней нашего простого модуля p . Также добавим состояние для поиска — переменную `type`. Если `type = 1`, то при совпадении хешей мы не делаем проверок, для `type = 2` мы делаем полную проверку.

```
// Hash searching
class Hash
{
private:
    // Hash prefix-values
    std::vector<int> hashes;

    // p^i, for i = 0,1,...,n-1,n
    std::vector<int> powers;

    // Module q
    static constexpr uint32_t q = static_cast<uint64_t>((1LL << 31) -
1LL);

    // Prime p
    int p;

    // Type of searching
    int type;
...
};
```

Листинг 1. Поля исходного класса Hash.

Добавим функции `build()` для подсчёта префиксов и степеней числа p , а также генератор для выбора p .

```

// Build
void build()
{
    p = gen(n / 2, n - 2);
    build_powers();
    build_hashes();
}

// Build powers
void build_powers()
{
    powers.assign(n, 1LL);

    for (int i = 1; i < n; ++i)
        powers[i] = (powers[i - 1] * 1LL * p) % q;
}

// Build hashes
void build_hashes()
{
    hashes.assign(n + 1, 0);

    for (int i = 1; i <= n; ++i)
        hashes[i] = (hashes[i - 1] * 1LL * p + text[i - 1]) % q;
}

// Generate
static int gen(int min = 0LL, int max = INT_MAX)
{
    std::mt19937_64 mt{ std::random_device{}() };
    std::uniform_int_distribution<> die{ min, max };
    return die(mt);
}

```

Листинг 2. Функции для предпосчёта в классе Hash.

Заметим, что здесь используется 1LL, чтобы не переполнить тип int (поскольку модуль q очень большой).

Теперь напомним простые функции для вычисления хешей для строк и подстрок, а также сравнение для строк с совпадающими хешами.

```

// Calculate hash
int hash(const std::wstring& str) const
{
    int result{};
    const int sz = static_cast<int>(str.size());
    for (int i = 0; i < sz; ++i)
        result = (result + str[i] * 1LL * powers[sz - i - 1]) % q;
    return result;
}

// Hash from i to j
int hash(int i, int j) const
{

```

```

        return (hashes[j] - (hashes[i - 1] * 1LL * powers[j - i + 1]) % q + q)
% q;
}

// Compare hashes
void cmp_hashes(int curr, int index, int patt, const std::wstring& currs,
const std::wstring& pattern, std::vector<int>& ans) const
{
    if (curr == patt)
    {
        const int sz = static_cast<int>(pattern.size());
        if (type == 2)
        {
            int cnt = sz / 2;
            int curr_index;
            while (cnt-->0)
            {
                curr_index = gen(0, sz);
                if (currs[curr_index] != pattern[curr_index])
                    return;
            }
        }
        else if (type == 3)
        {
            for (int i = 0; i < sz; ++i)
                if (currs[i] != pattern[i])
                    return;
        }

        ans.push_back(index);
    }
}

```

Листинг 3. Функции для подсчёта хешей.

Теперь осталось реализовать функцию поиска pattern.

```

// Find pattern positions in text
std::vector<int> find(const std::wstring& pattern) const
{
    const int patt_size = static_cast<int>(pattern.size());
    if (patt_size > n) return {};

    const int end = n - static_cast<int>(pattern.size()) + 1;
    const int patt_hash = hash(pattern);

    std::wstring currs{ text.begin(), text.begin() + patt_size };

    int curr_hash = hash(1, patt_size);

    std::vector<int> ans;
    cmp_hashes(curr_hash, 0, patt_hash, currs, pattern, ans);

    for (int i = 1; i < end; ++i)
    {
        currs.erase(0, 1);
        currs += text[i + patt_size - 1];
    }
}

```

```

        curr_hash = hash(i + 1, i + 1 + patt_size - 1);

        cmp_hashes(curr_hash, i, patt_hash, currs, pattern, ans);
    }

    return ans;
}

```

Листинг 4. Итоговая функция для поиска с помощью хеширования.

Так же в класс были добавлены некоторые мелкие вспомогательные функции. С полным кодом можно ознакомиться в *Приложении 2*.

Алгоритм Бойера-Мура.

Для реализации алгоритма необходимо просто построить таблицу плохих символов и осуществить поиск. Таблицу будем хранить в векторе целых чисел.

```

// Boyer-Moore algorithm
class BoyerMoore
{
private:
    // Table
    std::vector<int> table;

    // Build table
    void build()
    {
        table.resize(1 << (8 * sizeof(wchar_t)), -1);
        const int len = (int)pattern.length();
        for (int i = 0; i < len; ++i)
            table[static_cast<int>(pattern[i])] = i;
    }

public:
    // Find
    std::vector<int> find(const std::wstring& text)
    {
        std::vector<int> ans;
        int text_len = (int)text.length();
        int patt_len = (int)pattern.length();
        int shift{};

        while (shift <= text_len - patt_len)
        {
            int j = patt_len - 1;

            while (j >= 0 && pattern[j] == text[shift + j])
                --j;

            if (j < 0)

```

```

        {
            ans.push_back(shift);
            shift += (shift + patt_len < text_len) ? patt_len -
table[text[shift + patt_len]] : 1;
        }
        else shift += std::max(1, j - table[text[shift + j]]);
    }

    return ans;
}
...
};

```

Листинг 5. Основные функции для работы алгоритма Бойера-Мура

Полный код представлен в *Приложении 3*.

Алгоритм Кнута-Морриса-Пратта.

Данный алгоритм тоже очень прост в реализации: просто начать поиск и посчитать префиксы.

```

// Knuth-Morris-Pratt
class KMP
{
private:
    // Build prefix_function for pattern
    std::vector<int> build_prefix(const std::wstring& pattern)
    {
        const int n = (int)pattern.size();
        std::vector<int> prefix(n + 1, -1);

        int k;

        for (int i = 1; i <= n; ++i)
        {
            k = prefix[i - 1];
            while (k >= 0 && pattern[k] != pattern[i - 1])
            {
                k = prefix[k];
            }
            prefix[i] = k + 1;
        }

        return prefix;
    }

public:
    // Find std::wstring from begin index
    std::vector<int> find(const std::wstring& pattern, int begin = 0)
    {
        std::vector<int> prefix = build_prefix(pattern);
        std::vector<int> ans;
    }

```

```

const int n = (int)text.size();
const int m = (int)pattern.size();

for (int i = begin, k = 0; i < n; ++i)
{
    while (k >= 0 && pattern[k] != text[i])
        k = prefix[k];

    ++k;

    if (k == m)
    {
        ans.push_back(i - m + 1);
        k = prefix[k];
    }

    return ans;
}

...
};

```

Листинг 6. Основные функции для алгоритма Кнута-Морриса-Пратта.

С полным кодом для данного алгоритма можно ознакомиться в *Приложении 4*.

Алгоритм неточного поиска.

Начнём пошаговую реализацию:

Напишем функцию для подсчёта расстояния Дameraу-Левенштейна между двумя строками.

```

// Innacurate searching with Damerau-Levenshtein distance
class Innacurate_search
{
private:

    // Text
    std::wstring text;

    // Pattern
    std::wstring pattern;

    // Text size
    int n;

    // Pattern size
    int m;

    // Cost of inserting a symbol

```

```

int insertCost;

// Cost of the deletion a symbol
int deleteCost;

// Cost of replace the symbol
int replaceCost;

// Cost of transposition
int transposeCost;

// Percent of similarity
double similar;

// Calculating the Damerau-Levenshtein distance between strings
int damerau_Levenshtein_Distance(const std::wstring& source, const
std::wstring& target, int k) const
{
    const int len1 = (int)source.size();
    const int len2 = (int)target.size();

    if (len1 > len2)
        return damerau_Levenshtein_Distance(target, source, k);

    std::vector<int> prev(len1 + 1), curr(len1 + 1);

    std::iota(prev.begin(), prev.end(), 0);

    for (int j = 1; j <= len2; j++)
    {
        curr[0] = j;

        for (int i = 1; i <= len1; i++)
        {
            const int cost = (source[i - 1] == target[j - 1]) ? 0 :
replaceCost;

            curr[i] = std::min({ curr[i - 1] + insertCost, prev[i] +
deleteCost, prev[i - 1] + cost });

            if (i > 1 && j > 1 && source[i - 1] == target[j - 2] &&
source[i - 2] == target[j - 1])
                curr[i] = std::min(curr[i], prev[i - 2] +
transposeCost);
        }

        prev.swap(curr);
    }

    return (prev[len1] > k ? k + 1 : prev[len1]);
}
...
};

```

Листинг 7. Код для вычисления расстояния Дамерау-Левенштейна.

По умолчанию веса для вставки, удаления, замены и транспозиции у нас будут 1, а процент совпадения 0.75.

Теперь осталось написать сам алгоритм поиска.

```
// Find inclusions of pattern in text (filtered)
std::vector<std::pair<int, int>> find()      const
{
    // Return pair<position, length>
    std::vector<std::pair<int, int>> ans;

    // We are interested in strings, which are similar more than 75%
    const int k = m * (1.0 - similar);

    const int minSubstrLength = std::max(1, m - k);
    const int maxSubstrLength = m + k;
    int lastLevenshtein{};

    for (int i = 0; i <= n - minSubstrLength; ++i)
    {
        // Pair of <best distance, length of the best distance string>
        std::pair<int, int> best{ k + 1, 0 };

        // Iterate allowable length of substrings(m +-k)
        for (int currLength = minSubstrLength; currLength <= std::min(n -
i, maxSubstrLength); ++currLength)
        {
            std::wstring substring = text.substr(i, currLength);
            const int currDist = damerau_Levenshtein_Distance(substring,
pattern, k);

            if (currDist <= k && currDist < best.first)
            {
                best.first = currDist;
                best.second = currLength;
            }

            if (best.second)
            {
                if (ans.size())
                    if (ans.back().first + ans.back().second > i)
                        if (lastLevenshtein > best.first)
                            ans.back() = { i, best.second };
                        else ;
                    else
                        ans.push_back({ i, best.second });
                else
                    ans.push_back({ i, best.second });

                lastLevenshtein = best.first;
            }
        }

        return ans;
    }
}
```

Листинг 8. Код для поиска всех вхождений неточных совпадений.

С исходным кодом можно ознакомиться в *Приложении 5*.

Сравнение эффективности алгоритмов.

Теперь, когда у нас есть реализация для всех алгоритмов, необходимо проверить их работоспособность и сделать сравнение.

Для измерения скорости работы воспользуемся вспомогательным классом `Timer`.

```
// Timer
class Timer
{
private:
    using Clock = std::chrono::steady_clock;
    using Second = std::chrono::duration<double, std::ratio<1> >;
    std::chrono::time_point<Clock> m_beg{ Clock::now() };
public:

    void reset()
    {
        m_beg = Clock::now();
    }

    double elapsed() const
    {
        return std::chrono::duration_cast<Second>(Clock::now() -
m_beg).count();
    }
};
```

Листинг 9. Вспомогательный класс для измерения времени поиска.

Ещё изменим способ вычислений в нашем классе `Hash` (поскольку тесты проводятся на случайном тексте и одноразово, в случае многократного использования поиска на одном и том же тексте, стоит использовать префикс-функцию), будем считать хеш сразу. Значение нового хеша это:

$$h(s_i) = (h(s_{i-1}) - a[i-1] * p^{m-1}) * p + a[i].$$

Приступим к тестированию. Все тесты проводились на машине со следующими характеристиками: Windows 10 x64, Ryzen 7 3750H.

Для первого теста возьмём текст произведения “Война и Мир” (можно найти по данной ссылке:

<https://gist.github.com/Semionn/bdcb66640cc070450817686f6c818897>).

Данный текст содержит около полутора миллиона символов, чего нам, конечно, будет мало, поэтому продублируем данный текст ещё 99 раз. В качестве искомой подстроки будем брать:

- 1) ‘и’ — самый встречаемый символ в тексте.
- 2) ‘Ъ’ — символ, который ни разу не встречается в тексте.
- 3) Вырезка из текста длины 80, т.е. подстрока длины 80, которая содержится в тексте.
- 4) 80 случайных символов.
- 5) Вырезка длины 3303.
- 6) 3303 случайных символов.

Таблица 2. Среднее время работы алгоритмов поиска на тексте “Война и Мир”

	Hash		BM	KMP	Naive	Naive Opt.
	Type1	Type2				
‘и’	4.11	4.07	2.48	1.32	6.85	0.72
‘Ъ’	3.92	3.85	2.23	0.91	5.86	0.50
80	5.33	5.22	0.11	0.86	32.3	0.51
80 random	5.26	5.26	0.14	1.02	32.6	0.49
3303	5.54	5.47	0.09	0.98	95.1	0.58
3303 random	5.77	5.73	0.10	0.90	103.6	0.55

Если посмотреть на результаты тестов в таблице 2, можем заметить, что на случайном тексте, где не много вхождений pattern, разница в скоростях между случайной подстрокой и подстрокой, которая есть в тексте практически нет. Ещё стоит заметить, что наш хеш-алгоритм работает в среднем за одно и то же время, независимо от размера искомой подстроки. Алгоритм Бойера-Мура оказался лидирующим на больших данных, но очень плохо себя показал при поиске коротких подстрок.

Тест 2.

Теперь возьмём и составим текст из одной буквы ‘а’ длины 100 миллионов. Это будет являться худшим случаем для поиска.

Ниже в таблице 3 представлены результаты тестирования на текущем файле. Видим, что разница между типами алгоритма хеширования появляется в случае многократного вхождения. Лучшим среди алгоритмов в случае, если строка многократно входит в текст является алгоритм Кнута-Морриса-Пратта.

Бойер Мур является самым быстрым для больших подстрок, которые не содержатся в тексте.

Таблица 3. Среднее время работы алгоритмов поиска на тексте из одинакового символа.

	Hash Type1/Type2		BM	KMP	Naive	Naive Opt.
‘a’	1.51	1.55	1.02	0.83	2.37	0.95
‘б’	0.98	1.05	0.59	0.19	1.69	0.15
80	4.16	1.61	3.27	0.52	12.50	4.9
80 random	1.51	1.50	0.06	0.56	9.37	0.23
3303	6.12	1.47	1.52	0.34	31.3	10.9
3303 random	1.53	1.52	0.007	0.33	27.8	0.8

Осталось провести тестирование для последнего алгоритма — алгоритма неточного поиска. Для начала проведём тесты на то, насколько хорошо алгоритм умеет находить подстроки с разной точностью.

Возьмём текст:

“Здравстуйте здравстйуте здравствуйте здравствуйте здравие вдразуйсте привет это пример текста

Привет пиртве пирвет ветрип привет привет привет привет приветп ривет

Алгоритм неточного поиска также может искать точные вхождения подстрок.”

Возьмём подстроку “Здравствуйте” и сделаем поиск по нескольким точностям: 75, 50, 30.

В первом случае (Рисунок 7) видим, что наш алгоритм нашёл первые 4 слова. Действительно, они достаточно похожи на искомую подстроку.

Возьмём второй случай (Рисунок 8). Алгоритм нашёл слова, где были переставлены хаотичным образом все буквы в исходной подстроке.

Для третьего случая (Рисунок 9) наш алгоритм стал находить какие-то очень странные подстроки, вовсе не похожие на искомую.

На самом деле эта проблема связана с размером искомой подстроки: для очень коротких подстрок для схожести на 50% может быть различие в паре символов, а для больших подстрок даже схожесть 85% не гарантирует, что подстроки имеют что-то общее.

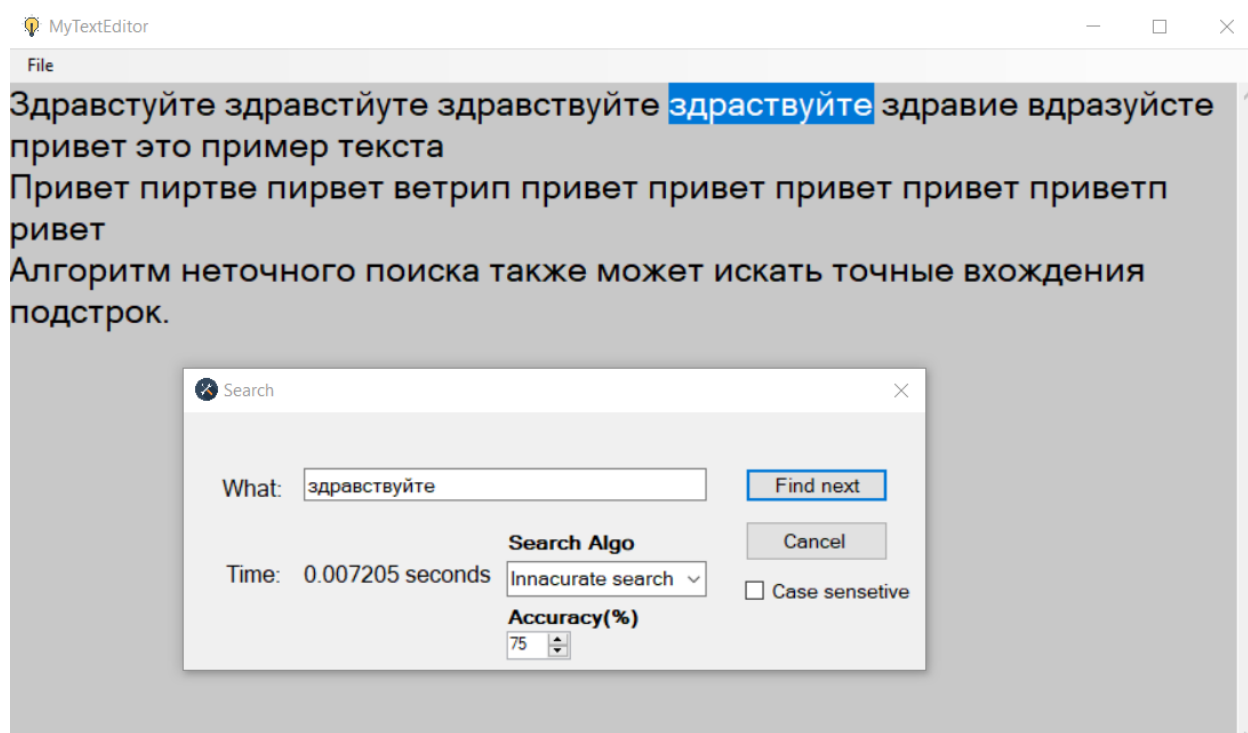


Рисунок 7. Пример неточного поиска для точности 75%.

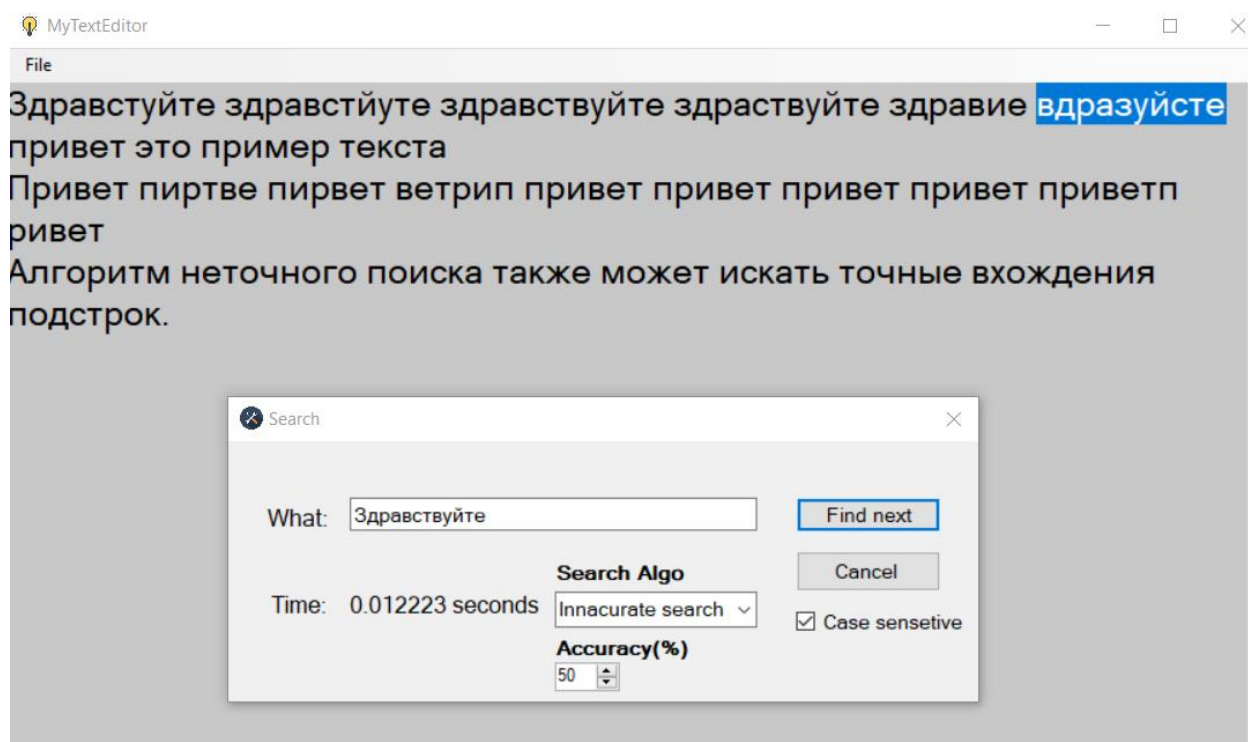


Рисунок 8. Пример неточного поиска для точности 50%.

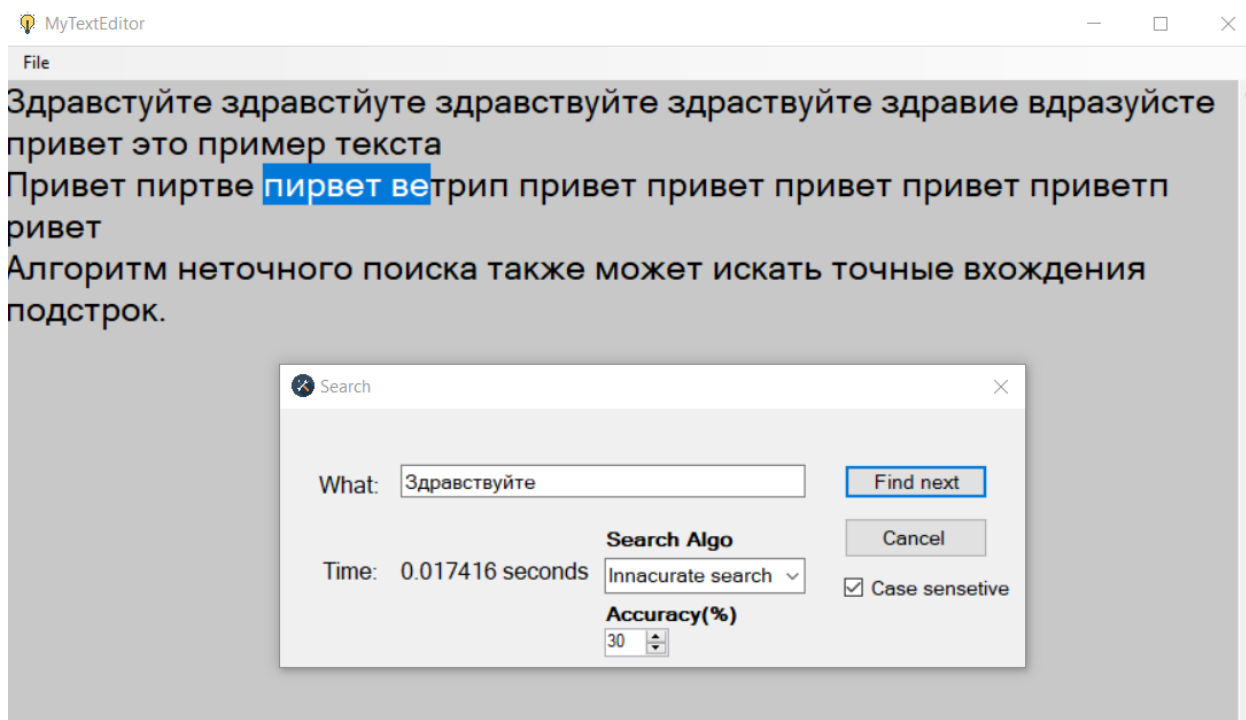


Рисунок 9. Пример неточного поиска для точности 30%.

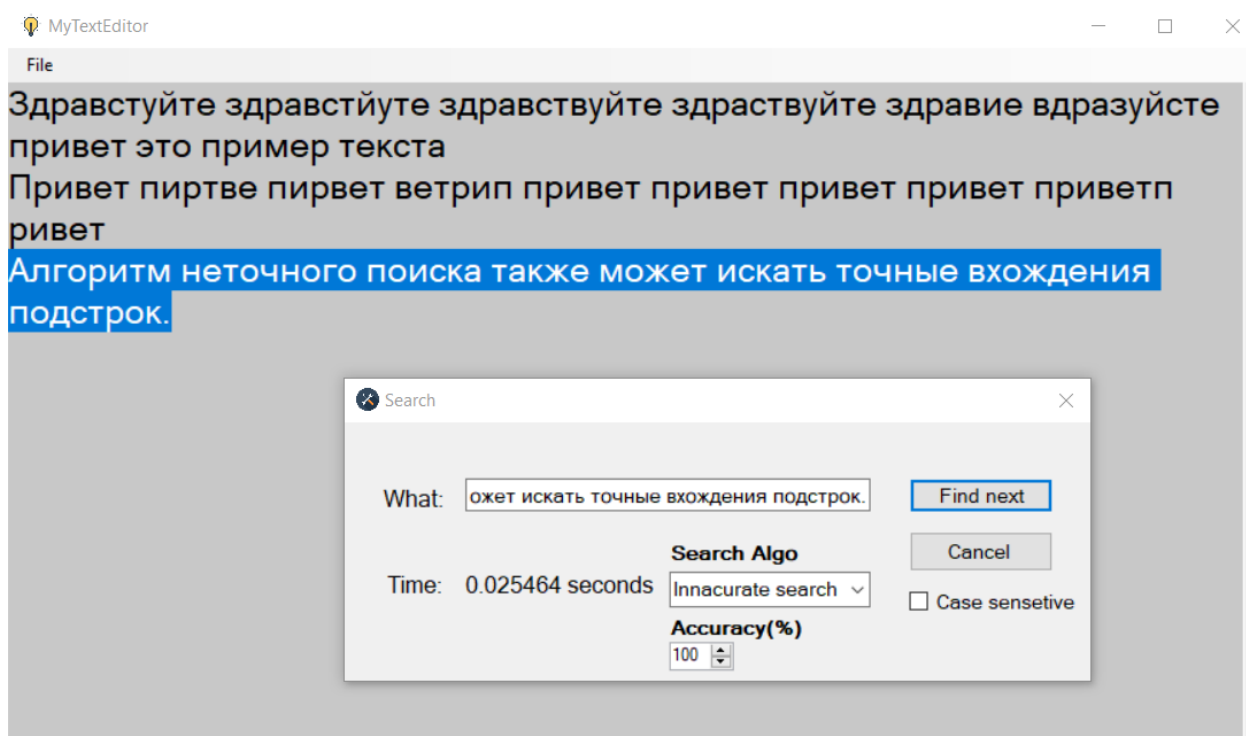


Рисунок 10. Пример неточного поиска для точности 100%.

Стоит заметить, что мы можем выставить точность 100% (Рисунок 10) и наш алгоритм будет делать точный поиск подстроки в тексте. Сложность такого алгоритма будет примерно равна наивному.

Теперь возьмём слово “Здарстуйте” и продублируем его на 200 тысяч символов и сделаем поиск с разной точностью. Результаты есть в таблице 4.

Таблица 4. Время работы неточного поиска на тексте длины 200.000 и подстроке длины 10.

Точность (%)	100	75	50	30
Время поиска (сек)	0.78	6.47	10.62	12.21

Видно, что даже на небольших текстах и подстроках время поиска существенно увеличилось, однако мы решаем более сложную задачу.

Заключение

Задача о поиске подстроки в тексте является достаточно востребованной на сегодняшний день, поскольку количество данных растёт с каждым днём и очень важно находить необходимую информацию в короткие сроки. Для поиска в подстроке мы рассмотрели алгоритмы Бойера-Мура, алгоритм на хешировании, Кнута-Морриса-Пратта и сравнили скорость их работы с наивным алгоритмом. Алгоритм на основе хеширования следует использовать в случае, если необходимо многократно делать запросы к исходному тексту, поскольку алгоритм строится на основе исходного текста. На случайных же данных, лучше использовать алгоритмы Кнута-Морриса-Пратта и Бойера-Мура, поскольку они лучше себя ведут на случайных текстах, что не может показать алгоритм на хешах. Также был предложен алгоритм для неточного поиска в тексте, что тоже очень необходимо, когда неизвестно точное вхождение подстроки. Его скорость работы достаточно медленная, однако он более универсальный по сравнению с предложенными ранее алгоритмами.

Приложения

Приложение 1. Необходимые библиотеки для работы кода.

```
#include <iostream>
#include <string>
#include <algorithm>
#include <vector>
#include <random>
#include <locale.h>
#include <locale>
#include <fstream>
#include <sstream>
#include <chrono>
#include <numeric>
```

Приложение 2. Исходный код на C++ для класса Hash.

```
// Hash searching
class Hash
{
private:

    // Our text
    std::wstring text;

    // Size of the text
    int n;

    // Hash prefix-values
    std::vector<int> hashes;

    //  $p^i$ , for  $i = 0, 1, \dots, n-1, n$ 
    std::vector<int> powers;

    // Module q
    static constexpr uint32_t q = static_cast<uint64_t>((1LL << 31) -
1LL);

    // Prime p
    int p;

    // Type of searching
    int type;

    // Build
    void build()
    {
        p = gen(n / 2, n - 2);
        build_powers();
        build_hashes();
    }

    // Build powers
    void build_powers()
    {
        powers.assign(n, 1LL);
```



```

        for (int i = 1; i < n; ++i)
            powers[i] = (powers[i - 1] * 1LL * p) % q;
    }

    // Build hashes
    void build_hashes()
    {
        hashes.assign(n + 1, 0);

        for (int i = 1; i <= n; ++i)
            hashes[i] = (hashes[i - 1] * 1LL * p + text[i - 1]) % q;
    }

    // Generate
    static int gen(int min = 0LL, int max = INT_MAX)
    {
        std::mt19937_64 mt{ std::random_device{}() };
        std::uniform_int_distribution<> die{ min, max };
        return die(mt);
    }

    // Calculate hash
    int hash(const std::wstring& str) const
    {
        int result{};
        const int sz = static_cast<int>(str.size());
        for (int i = 0; i < sz; ++i)
            result = (result + str[i] * 1LL * powers[sz - i - 1]) % q;
        return result;
    }

    // Hash from i to j
    int hash(int i, int j) const
    {
        return (hashes[j] - (hashes[i - 1] * 1LL * powers[j - i + 1]) % q
+ q) % q;
    }

    // Compare hashes
    void cmp_hashes(int curr, int index, int patt, const std::wstring&
currs, const std::wstring& pattern, std::vector<int>& ans) const
    {
        if (curr == patt)
        {
            const int sz = static_cast<int>(pattern.size());
            if (type == 2)
            {
                int cnt = sz / 2;
                int curr_index;
                while (cnt--)
                {
                    curr_index = gen(0, sz);
                    if (currs[curr_index] != pattern[curr_index])
                        return;
                }
            }
            else if (type == 3)
            {
                for (int i = 0; i < sz; ++i)

```

```

        if (currs[i] != pattern[i])
            return;
    }

    ans.push_back(index);
}

public:

    // Basic constructor
    Hash()
        : type(2)
        , text(L"")
        , n(0)
    { }

    // Construcotr with std::wstring
    Hash(const std::wstring& text) : Hash()
    {
        rebuild(text);
    }

    // Constructor with std::string
    Hash(const std::string& text) : Hash(std::wstring{
text.begin(),text.end() })
    { }

    // Rebuild for new text
    void rebuild(const std::wstring& text)
    {
        if (Hash::text == text) return;

        Hash::text = text;
        n = static_cast<int>(text.size());
        build();
    }

    // Rebuild for std::string
    void rebuild(const std::string& text)
    {
        rebuild(std::wstring{ text.begin(),text.end() });
    }

    // Find pattern positions in text
    std::vector<int> find(const std::wstring& pattern) const
    {
        const int patt_size = static_cast<int>(pattern.size());
        if (patt_size > n) return {};

        const int end = n - static_cast<int>(pattern.size()) + 1;
        const int patt_hash = hash(pattern);

        std::wstring currs{ text.begin(), text.begin() + patt_size };

        int curr_hash = hash(1, patt_size);

        std::vector<int> ans;
        cmp_hashes(curr_hash, 0, patt_hash, currs, pattern, ans);
    }

```

```

        for (int i = 1; i < end; ++i)
        {
            currs.erase(0, 1);
            currs += text[i + patt_size - 1];

            curr_hash = hash(i + 1, i + 1 + patt_size - 1);

            cmp_hashes(curr_hash, i, patt_hash, currs, pattern, ans);
        }

        return ans;
    }

    // Find with std::string
    std::vector<int> find(const std::string& pattern)    const
    {
        return find(std::wstring{ pattern.begin(), pattern.end() });
    }

    // Checking build it or not
    bool is_build() const
    {
        return n;
    }

    // Turn on quickSearch type
    void quick()
    {
        type = 1;
    }

    // Turn on optimal search
    void optimal()
    {
        type = 2;
    }

    // Turn on accurate slow mode
    void accurate()
    {
        type = 3;
    }

    // Basic destructor
    ~Hash() = default;

};

```

Приложение 3. Исходный код на C++ для алгоритма Бойера-Мура.

```

// Boyer-Moore algorithm
class BoyerMoore
{
private:

```

```

// Pattern for search
std::wstring pattern;

// Table
std::vector<int> table;

// Build table
void build()
{
    table.resize(1 << (8 * sizeof(wchar_t)), -1);
    const int len = (int)pattern.length();
    for (int i = 0; i < len; ++i)
        table[static_cast<int>(pattern[i])] = i;
}

public:

// Default
BoyerMoore() = default;

// Basic constructor
BoyerMoore(const std::wstring& pattern)
    : pattern(pattern)
{
    build();
}

// std::string constructor
BoyerMoore(const std::string& _pattern)
{
    std::wstring curr(_pattern.begin(), _pattern.end());
    BoyerMoore::pattern = curr;
    build();
}

// Rebuild(std::string)
void rebuild(const std::string& _pattern)
{
    std::wstring curr(_pattern.begin(), _pattern.end());
    BoyerMoore::pattern = curr;
    rebuild(curr);
}

// Rebuild
void rebuild(const std::wstring& pattern)
{
    if (BoyerMoore::pattern == pattern) return;

    BoyerMoore::pattern = pattern;
    build();
}

// Check build it or not
bool is_build() const
{
    return pattern != L"";
}

// Find (std::string)
std::vector<int> find(const std::string& text)
{
    std::wstring curr(text.begin(), text.end());
    return find(curr);
}

```

```

// Find
std::vector<int> find(const std::wstring& text)
{
    std::vector<int> ans;
    int text_len = (int)text.length();
    int patt_len = (int)pattern.length();
    int shift{0};

    while (shift <= text_len - patt_len)
    {
        int j = patt_len - 1;

        while (j >= 0 && pattern[j] == text[shift + j])
            --j;

        if (j < 0)
        {
            ans.push_back(shift);
            shift += (shift + patt_len < text_len) ? patt_len - table[text[shift + patt_len]] : 1;
        }
        else shift += std::max(1, j - table[text[shift + j]]);
    }

    return ans;
}

// Basic destructor
~BoyerMoore() = default;

};

```

Приложение 4. Исходный код на C++ для алгоритма Кнута-Морриса-Пратта.

```

// Knuth-Morris-Pratt
class KMP
{
private:

    // Text for searching
    std::wstring text;

    // Build prefix_function for pattern
    std::vector<int> build_prefix(const std::wstring& pattern)
    {
        const int n = (int)pattern.size();
        std::vector<int> prefix(n + 1, -1);

        int k;

        for (int i = 1; i <= n; ++i)
        {
            k = prefix[i - 1];
            while (k >= 0 && pattern[k] != pattern[i - 1])
            {
                k = prefix[k];
            }
            prefix[i] = k + 1;
        }
    }
};

```

```

    }

    return prefix;
}

public:

    // Basic constructor
    KMP() = default;

    // Constructor with std::wstring
    KMP(const std::wstring& text)
        : text(text)
    { }

    // Consturtor with std::string
    KMP(const std::string& text) : KMP(std::wstring{
text.begin(),text.end() })
    { }

    // Rebuild with std::wstring
    void rebuild(const std::wstring& text)
    {
        KMP::text = text;
    }

    // Rebuild with std::string
    void rebuild(const std::string& text)
    {
        rebuild(std::wstring{ text.begin(),text.end() });
    }

    // Check build it or not
    bool is_build() const
    {
        return text != L"";
    }

    // Find std::wstring from begin index
    std::vector<int> find(const std::wstring& pattern, int begin = 0)
    {
        std::vector<int> prefix = build_prefix(pattern);
        std::vector<int> ans;

        const int n = (int)text.size();
        const int m = (int)pattern.size();

        for (int i = begin, k = 0; i < n; ++i)
        {
            while (k >= 0 && pattern[k] != text[i])
                k = prefix[k];

            ++k;

            if (k == m)
            {
                ans.push_back(i - m + 1);
                k = prefix[k];
            }
        }
    }

```

```

    }

    return ans;
}

// Find std::string from begin index
std::vector<int> find(const std::string& pattern, int begin = 0)
{
    return find(std::wstring{ pattern.begin(), pattern.end() }, begin);
}

// Basic destructor
~KMP() = default;
};

```

Приложение 5. Исходный код на C++ для неточного поиска.

```

// Innacurate searching with Damerau-Levenshtein distance
class Innacurate_search
{
private:

    // Text
    std::wstring text;

    // Pattern
    std::wstring pattern;

    // Text size
    int n;

    // Pattern size
    int m;

    // Cost of inserting a symbol
    int insertCost;

    // Cost of the deletion a symbol
    int deleteCost;

    // Cost of replace the symbol
    int replaceCost;

    // Cost of transposition
    int transposeCost;

    // Percent of similarity
    double similar;

    // Calculating the Damerau-Levenshtein distance between strings
    int damerau_Levenshtein_Distance(const std::wstring& source, const
std::wstring& target, int k) const
    {
        const int len1 = (int)source.size();
        const int len2 = (int)target.size();

        if (len1 > len2)
            return damerau_Levenshtein_Distance(target, source, k);
    }
};

```

```

        std::vector<int> prev(len1 + 1), curr(len1 + 1);

        std::iota(prev.begin(), prev.end(), 0);

        for (int j = 1; j <= len2; j++)
        {
            curr[0] = j;

            for (int i = 1; i <= len1; i++)
            {
                const int cost = (source[i - 1] == target[j - 1]) ? 0 :
replaceCost;

                curr[i] = std::min({ curr[i - 1] + insertCost, prev[i] +
deleteCost, prev[i - 1] + cost });

                if (i > 1 && j > 1 && source[i - 1] == target[j - 2] &&
source[i - 2] == target[j - 1])
                    curr[i] = std::min(curr[i], prev[i - 2] +
transposeCost);
            }

            prev.swap(curr);
        }

        return (prev[len1] > k ? k + 1 : prev[len1]);
    }

public:
    // Default
    Innacurate_search()
        : n(0)
        , m(0)
        , text{ L"" }
        , pattern{ L"" }
        , insertCost(1)
        , deleteCost(1)
        , replaceCost(1)
        , transposeCost(1)
        , similar(0.75)
    { }

    // Basic constructor with std::wstring
    Innacurate_search(const std::wstring& text, const std::wstring&
pattern)
        : text(text)
        , pattern(pattern)
    {
        n = (int)text.length();
        m = (int)pattern.length();
    }

    // Basic constructor with std::string
    Innacurate_search(const std::string& text, const std::string& pattern)
    : Innacurate_search(std::wstring{ text.begin(), text.end() }, std::wstring{
pattern.begin(), pattern.end() })
    { }

```



```

// Full constructor with std::wstring
Innacurate_search(const std::wstring& text, const std::wstring&
pattern, int insertCost, int deleteCost, int replaceCost, int
transposeCost) : Innacurate_search(text, pattern)
{
    Innacurate_search::insertCost = insertCost;
    Innacurate_search::deleteCost = deleteCost;
    Innacurate_search::replaceCost = replaceCost;
    Innacurate_search::transposeCost = transposeCost;
}

// Full constructor with std::string
Innacurate_search(const std::string& text, const std::string& pattern,
int insertCost, int deleteCost, int replaceCost, int transposeCost)
    : Innacurate_search(std::wstring{ text.begin(),text.end() },
std::wstring{ pattern.begin(),pattern.end() }, insertCost, deleteCost,
replaceCost, transposeCost)
{ }

// Rebuild text std::wstring
void text_rebuild(const std::wstring& text)
{
    Innacurate_search::text = text;
    n = (int)text.length();
}

// Rebuild text std::string
void text_rebuild(const std::string& text)
{
    text_rebuild(std::wstring{ text.begin(),text.end() });
}

// Rebuild pattern std::wstring
void pattern_rebuild(const std::wstring& pattern)
{
    Innacurate_search::pattern = pattern;
    m = (int)pattern.length();
}

// Rebuild pattern std::string
void pattern_rebuild(const std::string& pattern)
{
    pattern_rebuild(std::wstring{ pattern.begin(),pattern.end() });
}

// Find inclusions of pattern in text (filtered)
std::vector<std::pair<int, int>> find() const
{
    // Return pair<position, length>
    std::vector<std::pair<int, int>> ans;

    // We are interested in strings, which are similar more than 75%
    const int k = m * (1.0 - similar);

    const int minSubstrLength = std::max(1, m - k);
    const int maxSubstrLength = m + k;
    int lastLevenshtein{};

    for (int i = 0; i <= n - minSubstrLength; ++i)
    {

```

```

        // Pair of <best distance, length of the best distance string>
        std::pair<int, int> best{ k + 1, 0 };

        // Iterate allowable length of substrings (m +-k)
        for (int currLength = minSubstrLength; currLength <=
std::min(n - i, maxSubstrLength); ++currLength)
        {
            std::wstring substring = text.substr(i, currLength);
            const int currDist =
damerau_Levenshtein_Distance(substring, pattern, k);

            if (currDist <= k && currDist < best.first)
            {
                best.first = currDist;
                best.second = currLength;
            }
        }

        if (best.second)
        {
            if (ans.size())
                if (ans.back().first + ans.back().second > i)
                    if (lastLevenshtein > best.first)
                        ans.back() = { i, best.second };
                    else ;
                else
                    ans.push_back({ i, best.second });
            else
                ans.push_back({ i, best.second });

            lastLevenshtein = best.first;
        }
    }

    return ans;
}

// Set accuracy
void accuracy(double similar)
{
    Innacurate_search::similar = similar / 100.0;
}

// Setters
void setInsertCost(int insertCost)
{
    Innacurate_search::insertCost = insertCost;
}

void setDeleteCost(int deleteCost)
{
    Innacurate_search::deleteCost = deleteCost;
}

void setReplaceCost(int replaceCost)
{
    Innacurate_search::replaceCost = replaceCost;
}

```

```
void setTransposeCost(int transposeCost)
{
    Innacurate_search::transposeCost = transposeCost;
}

// Get the current accuracy
const double get_accuracy() const
{
    return similar * 100.0;
}

// Destructor
~Innacurate_search() = default;

};
```

Список литературы

1. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ: [часть III глава 11]. – Москва: Издательство «Вильямс», 2005 (перевод второго издания 2001 года). – ISBN 978-5-907114-11-1.
2. Wikipedia. Levenshtein distance [Электронный ресурс]. – Режим доступа: https://en.wikipedia.org/wiki/Levenshtein_distance. Дата обращения: 05.12.2023.
3. Wikipedia. Boyer–Moore string-search algorithm [Электронный ресурс]. – Режим доступа: https://en.wikipedia.org/wiki/Boyer%E2%80%93Moore_string-search_algorithm. Дата обращения: 05.12.2023.
4. GeeksforGeeks. KMP Algorithm for Pattern Searching [Электронный ресурс]. – Режим доступа: <https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/>. Дата обращения: 05.12.2023.
5. Gusfield, Dan. Algorithms on Strings, Trees, and Sequences. – Cambridge: Cambridge University Press, 1997. – ISBN 978-0521585194.