Київський національний університет імені Тараса Шевченка

Факультет комп`ютерних наук та кібернетики

Кафедра інтелектуальних програмних систем

Алгоритми та складність

Завдання №2

“ Алгоритм Джонсона для розріджених графів ”

Варіант №6

Виконав студент 2-го курсу

Групи ІПС-22

Ковальчук Артем Вячеславович

Київ - 2023

**Завдання**:

1. Реалізуйте алгоритми пошуку множини слів зі словника в тексті:

* безпосередній пошук по кожному слову зі словника,
* алгоритм Ахо-Корасік,
* порівняйте їх ефективність.

1. обчислити відстань Левенштейна-Дамерау та вивести на екран послідовність дій для перетворення першого рядка в другий

**Теорія**

Безпосередній пошук (або прямий пошук) - це базовий алгоритм пошуку підрядка у рядку. Він працює шляхом послідовного порівняння кожного символу у рядку з кожним символом у шуканому підрядку. Цей метод є простим у реалізації, але може бути неефективним для великих текстів або при пошуку великої кількості підрядків, оскільки вимагає багаторазового порівняння символів.

Алгоритм Ахо-Корасік - це алгоритм пошуку рядків, який ефективно шукає входження багатьох підрядків (або "слів") одночасно. Він використовує структуру даних, відому як "trie" (префіксне дерево), для зберігання шуканих слів, і створює вказівники невдачі, які використовуються для уникнення повторного пошуку вже перевірених символів у випадку невдачі. Ахо-Корасік є набагато швидшим за безпосередній пошук, коли треба знайти багато підрядків у великому тексті, оскільки дозволяє шукати всі підрядки одночасно в одному проході по тексту.

Відстань Левенштейна-Дамерау - це міра відмінності між двома рядками. Вона вимірюється як мінімальна кількість операцій вставки, видалення, заміни одного символу на інший та транспозиції двох сусідніх символів, необхідних для перетворення одного рядка в інший. Цей алгоритм є розширенням відстані Левенштейна, додавши можливість транспозиції символів (обмін місцями двох сусідніх символів). Відстань Левенштейна-Дамерау часто використовується в програмах для автоматичного виправлення помилок, таких як клавіатурні помилки у текстових редакторах або системах розпізнавання мови, де помилки часто включають перестановку букв або друкарські помилки.

**Застосування**

* Безпосередній пошук може бути використаний у простих сценаріях, де потрібно знайти невелику кількість слів у тексті або коли потрібно мінімізувати використання пам'яті.
* Алгоритм Ахо-Корасік ефективний у випадках, коли потрібно знайти багато різних підрядків у великому тексті, наприклад, для фільтрації небажаного контенту, пошуку ключових слів або у розробці антивірусних програм.
* Відстань Левенштейна-Дамерау застосовується у програмах для виправлення помилок у тексті, у системах автоматичного виправлення помилок, у розпізнаванні мови та в інших областях, де важливо розуміти та обробляти варіації у написанні слів або фраз.

**Математичне обгрунтування**

Безпосередній пошук (або лінійний пошук) є найпростішим алгоритмом пошуку підрядків. Він перевіряє кожну можливу позицію у тексті, де може починатися шуканий підрядок. Припустимо, маємо текст довжиною N та шуканий підрядок довжиною M . Алгоритм порівнює підрядок з кожною підпослідовністю тексту довжиною M.

При кожному кроці алгоритм порівнює символи підрядка з символами у тексті. У гіршому випадку (коли співпадіння відсутнє або знаходиться в кінці тексту) алгоритм виконає N - M + 1 порівнянь. Таким чином, часова складність безпосереднього пошуку у гіршому випадку є O(N \* M) .

Алгоритм Ахо-Корасік використовує структуру даних, відому як trie (префіксне дерево), для зберігання всіх шуканих підрядків. Додатково, використовуються вказівники невдачі для кожного вузла, які вказують на найдовший суфікс поточного префікса, який також є префіксом іншого слова в trie.

Нехай кількість символів у всіх шуканих словах становить S , і довжина тексту - N. Будівництво trie має часову складність O(S) , а побудова вказівників невдачі також має часову складність O(S) , оскільки кожен символ кожного слова обробляється не більше ніж двічі. Пошук у тексті має часову складність O(N) , оскільки кожен символ тексту обробляється один раз, незалежно від кількості або довжини шуканих слів. Таким чином, загальна часова складність алгоритму Ахо-Корасік становить O(S + N).

Відстань Левенштейна-Дамерау вимірює мінімальну кількість одиночних операцій редагування (вставки, видалення, заміни, транспозиції), необхідних для перетворення одного рядка в інший. Математично це можна представити як рекурсивну функцію.

Нехай d[i, j] - відстань Левенштейна-Дамерау між першими i символами першого рядка і першими j символами другого рядка. Тоді d[i, j] можна обчислити за формулою:

* d[0, j] = j і d[i, 0] = i (вартість перетворення порожнього рядка у інший рядок або навпаки).
* Якщо останні символи двох рядків однакові (скажімо, X[i] = Y[j] ), то d[i, j] = d[i-1, j-1] .
* Якщо останні символи різні, то d[i, j] є мінімальною вартістю серед трьох операцій:

- Вставка: d[i, j-1] + 1

- Видалення: d[i-1, j] + 1

- Заміна: d[i-1, j-1] + 1

- Транспозиція: Якщо два останні символи можна обміняти місцями, тоді додаємо операцію транспозиції: d[i-2, j-2] + 1 (тільки якщо i, j > 1 та X[i] = Y[j-1] , X[i-1] = Y[j] ).

Остаточно, d[m, n] , де m та n - довжини двох рядків, визначає відстань Левенштейна-Дамерау. Цей підхід має часову складність O(mn) , оскільки потрібно обчислити значення для кожної комірки у матриці розміром m\*n.

**Програмна реалізація**

Для реалізації було обрано мову програмування Python.

1. Безпосередній пошук. Функція naive\_search виконує простий пошук підрядків у тексті. Вона приймає текст та список слів для пошуку, а потім використовує регулярні вирази (через модуль `re` Python) для знаходження всіх входжень кожного слова в тексті. Результати зберігаються у словнику, де ключами є слова, а значеннями - списки індексів, де ці слова зустрічаються в тексті.

# Безпосередній пошук  
def naive\_search(text, words):  
 found\_words = {word: re.findall(word, text, re.IGNORECASE) for word in words}  
 return found\_words

1. Алгоритм Ахо-Корасік

Цей алгоритм реалізований у класах TrieNode і AhoCorasick:TrieNode:

Кожен вузол trie містить словник дочірніх вузлів, вказівник на вузол невдачі та інформацію про те, чи є цей вузол кінцем слова. Крім того, кожен вузол зберігає список слів, які закінчуються в цьому вузлі.

* AhoCorasick: Клас для створення trie із списку слів і побудови вказівників невдачі. Метод add\_word додає слово в trie, а build\_failure\_pointers будує вказівники невдачі для кожного вузла. Метод search виконує пошук у тексті, використовуючи побудовану структуру trie та вказівники невдачі, повертаючи словник зі словами та їх позиціями в тексті.
* class TrieNode:  
   def \_\_init\_\_(self):  
   self.children = {}  
   self.fail = None  
   self.is\_end\_of\_word = False  
   self.output = []  
    
  class AhoCorasick:  
   def \_\_init\_\_(self):  
   self.root = TrieNode()  
    
   def add\_word(self, word):  
   node = self.root  
   for char in word.lower():  
   if char not in node.children:  
   node.children[char] = TrieNode()  
   node = node.children[char]  
   node.is\_end\_of\_word = True  
   node.output.append(word)  
    
   def build\_failure\_pointers(self):  
   queue = [self.root]  
   while queue:  
   current\_node = queue.pop(0)  
   for char, child\_node in current\_node.children.items():  
   queue.append(child\_node)  
   fail\_node = current\_node.fail  
   while fail\_node and char not in fail\_node.children:  
   fail\_node = fail\_node.fail  
   child\_node.fail = fail\_node.children[char] if fail\_node else self.root  
   if child\_node.fail:  
   child\_node.output.extend(child\_node.fail.output)  
    
   def search(self, text):  
   node = self.root  
   results = {}  
   for i in range(len(text)):  
   char = text[i].lower()  
   while node and char not in node.children:  
   node = node.fail  
   if not node:  
   node = self.root  
   continue  
   node = node.children[char]  
   if node.output:  
   for word in node.output:  
   if word not in results:  
   results[word] = []  
   results[word].append(i - len(word) + 1)  
   return results

1. Відстань Левенштейна-Дамерау

Функція damerau\_levenshtein\_distance\_with\_path розраховує відстань Левенштейна-Дамерау та визначає послідовність редагувань для перетворення одного рядка в інший. Вона використовує динамічне програмування для побудови матриці відстаней, де кожна комірка матриці представляє мінімальну відстань між підрядками. Функція також зберігає послідовність операцій (видалення, вставка, заміна, транспозиція), необхідних для перетворення першого рядка в другий.

def damerau\_levenshtein\_distance\_with\_path(s1, s2):  
 d = {}  
 lenstr1 = len(s1)  
 lenstr2 = len(s2)  
 path = {}  
  
 for i in range(-1, lenstr1 + 1):  
 d[(i, -1)] = i + 1  
 for j in range(-1, lenstr2 + 1):  
 d[(-1, j)] = j + 1  
  
 for i in range(lenstr1):  
 for j in range(lenstr2):  
 if s1[i] == s2[j]:  
 cost = 0  
 else:  
 cost = 1  
 d[(i, j)] = min(  
 d[(i - 1, j)] + 1,  
 d[(i, j - 1)] + 1,  
 d[(i - 1, j - 1)] + cost  
 )  
 path[(i, j)] = "substitution" if cost == 1 else "no operation"  
  
 if i and j and s1[i] == s2[j - 1] and s1[i - 1] == s2[j]:  
 if d[(i, j)] > d[i - 2, j - 2] + cost:  
 d[(i, j)] = d[i - 2, j - 2] + cost  
 path[(i, j)] = "transposition"  
  
 if d[(i, j)] == d[(i - 1, j)] + 1:  
 path[(i, j)] = "deletion"  
 elif d[(i, j)] == d[(i, j - 1)] + 1:  
 path[(i, j)] = "insertion"  
  
  
 operations = []  
 i, j = lenstr1 - 1, lenstr2 - 1  
 while i >= 0 and j >= 0:  
 operation = path[(i, j)]  
 if operation == "deletion":  
 operations.append(f"Delete '{s1[i]}' at position {i}")  
 i -= 1  
 elif operation == "insertion":  
 operations.append(f"Insert '{s2[j]}' at position {i + 1}")  
 j -= 1  
 elif operation == "substitution":  
 operations.append(f"Substitute '{s1[i]}' with '{s2[j]}' at position {i}")  
 i -= 1  
 j -= 1  
 elif operation == "transposition":  
 operations.append(f"Transpose '{s1[i - 1]}' and '{s1[i]}'")  
 i -= 2  
 j -= 2  
 elif operation == "no operation":  
 i -= 1  
 j -= 1  
  
 return d[lenstr1 - 1, lenstr2 - 1], operations[::-1]

1. Читання тексту з DOCX

Функція read\_text\_from\_docx використовує бібліотеку docx для зчитування тексту з файлу формату DOCX. Вона ітерує через всі параграфи в документі, збираючи їх у один рядок.

def read\_text\_from\_docx(filename):  
 doc = docx.Document(filename)  
 full\_text = []  
 for para in doc.paragraphs:  
 full\_text.append(para.text)  
 return '\n'.join(full\_text)

1. Візуалізація результатів

Функція plot\_search\_results створює графік за допомогою matplotlib для візуалізації кількості знайдених слів кожним методом пошуку. Вона використовує значення, отримані від naive\_search та AhoCorasick.search, для порівняння кількості входжень кожного слова в тексті.

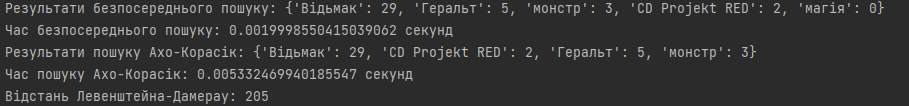
def plot\_search\_results(naive\_counts, aho\_counts):  
 labels = list(search\_words)  
 naive\_values = [naive\_counts.get(word, 0) for word in labels]  
 aho\_values = [aho\_counts.get(word, 0) for word in labels]  
  
 x = range(len(labels))  
 width = 0.35  
  
 fig, ax = plt.subplots()  
 rects1 = ax.bar(x, naive\_values, width, label='Безпосередній пошук')  
 rects2 = ax.bar([p + width for p in x], aho\_values, width, label='Ахо-Корасік')  
  
 ax.set\_ylabel('Кількість знайдених слів')  
 ax.set\_title('Порівняння методів пошуку')  
 ax.set\_xticks([p + width / 2 for p in x])  
 ax.set\_xticklabels(labels)  
 ax.legend()  
  
 ax.bar\_label(rects1, padding=3)  
 ax.bar\_label(rects2, padding=3)  
  
 plt.show()

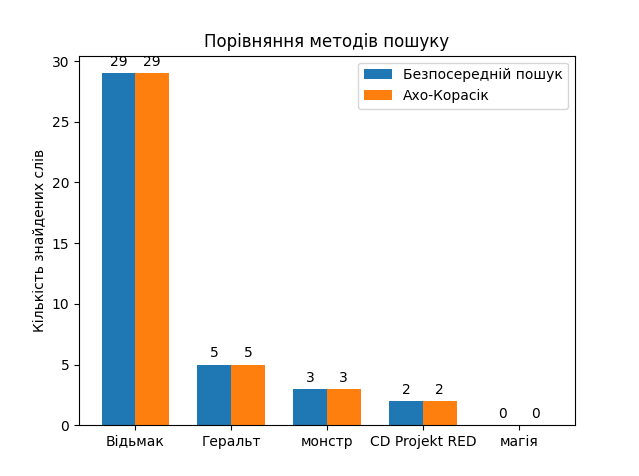
1. Головний скрипт

У головній частині скрипта (`if \_\_name\_\_ == "\_\_main\_\_":`) виконуються наступні дії:

* Зчитування тексту з файлу DOCX.
* Визначення словників слів для пошуку.
* Виконання безпосереднього пошуку та пошуку за алгоритмом Ахо-Корасік, замір часу виконання для кожного методу.
* Візуалізація кількості знайдених слів для обох методів пошуку.
* Обчислення відстані Левенштейна-Дамерау між двома заданими рядками та виведення послідовності операцій для перетворення першого рядка в другий.
* if \_\_name\_\_ == "\_\_main\_\_":  
    
   text = read\_text\_from\_docx('Test.docx')  
    
   search\_words = ["Відьмак", "Геральт", "монстр", "CD Projekt RED", "магія"]  
    
   # Замір часу для безпосереднього пошуку  
   start\_time = time.time()  
   naive\_results = naive\_search(text, search\_words)  
   naive\_search\_time = time.time() - start\_time  
   naive\_counts = {word: len(occurrences) for word, occurrences in naive\_results.items()}  
   print("Результати безпосереднього пошуку:", naive\_counts)  
   print("Час безпосереднього пошуку:", naive\_search\_time, "секунд")  
    
   # Замір часу для Ахо-Корасік  
   start\_time = time.time()  
   aho\_corasick = AhoCorasick()  
   for word in search\_words:  
   aho\_corasick.add\_word(word)  
   aho\_corasick.build\_failure\_pointers()  
   aho\_corasick\_results = aho\_corasick.search(text)  
   aho\_corasick\_search\_time = time.time() - start\_time  
   aho\_counts = {word: len(positions) for word, positions in aho\_corasick\_results.items()}  
   print("Результати пошуку Ахо-Корасік:", aho\_counts)  
   print("Час пошуку Ахо-Корасік:", aho\_corasick\_search\_time, "секунд")  
    
   # Візуалізація результатів  
   plot\_search\_results(naive\_counts, aho\_counts)  
    
   # Два рядки для аналізу відстані Левенштейна-Дамерау  
   line1 = "Секрет успіху ігор криється не лише в популярності книг про Відьмака, адже навіть не всі вони були перекладені англійською. " \  
   "Розробникам із CD Projekt RED вже у першій грі «Відьмак», випущеній у 2007 році, вдалося передати унікальний дух творів Сапковського."  
   line2 = "Те незвичайне, що найбільше подобається фанатам у книгах та іграх про Відьмака, це сам герой – Геральт із Рівії, відьмак. «Відьмак» – це персонаж слов'янської міфології. " \  
   "Саме слово походить від праслов'янського \* vede («знати»)."  
   distance, operations = damerau\_levenshtein\_distance\_with\_path(line1, line2)  
   print("Відстань Левенштейна-Дамерау:", distance)  
   print("Операції для перетворення рядка:")  
   for op in operations:  
   print(op)

**Отримані результати**

****

****

Отримані результати можна проаналізувати наступним чином:

1. Результати безпосереднього пошуку:

* Слова знайдені у тексті:
* "Відьмак" зустрічається 29 разів.
* "Геральт" - 5 разів.
* "монстр" - 3 рази.
* "CD Projekt RED" - 2 рази.
* "магія" не зустрічається.
* Час виконання пошуку складає приблизно 0.002 секунди.

1. Результати пошуку Ахо-Корасік:

* Слова знайдені у тексті з такою ж частотою, як і у безпосередньому пошуку, окрім слова "магія", яке не зустрічається.
* Час виконання пошуку складає приблизно 0.0053 секунди, що є трохи повільніше, ніж у випадку безпосереднього пошуку.

1. Аналіз відстані Левенштейна-Дамерау:

* Відстань між двома рядками складає 205, що вказує на значну різницю між ними.
* Послідовність операцій для перетворення першого рядка в другий включає численні заміни, вставки, видалення та декілька транспозицій. Це підтверджує, що рядки значно відрізняються один від одного.

**ВИСНОВКИ**

Ефективність пошукових алгоритмів: в цьому конкретному випадку безпосередній пошук виявився швидшим за алгоритм Ахо-Корасік. Це може бути пов'язано з відносною простотою і меншою обчислювальною вимогливістю безпосереднього пошуку у порівнянні з більш складним і ресурсомістким процесом побудови trie у алгоритмі Ахо-Корасік.

Придатність алгоритму Ахо-Корасік: Незважаючи на більший час виконання в даному випадку, алгоритм Ахо-Корасік має переваги при пошуку великої кількості слів у великих текстах, завдяки своїй здатності ефективно обробляти множинні підрядки одночасно.

Застосування відстані Левенштейна-Дамерау: Аналіз відстані Левенштейна-Дамерау був корисним для визначення ступеня відмінності між двома текстовими рядками, що може бути корисним у застосуваннях, пов'язаних із обробкою природної мови та автоматичним виправленням помилок. Загалом, виконана робота демонструє успішне застосування різних методів пошуку та аналізу тексту, які можуть бути використані в різних сценаріях обробки даних. Це підтверджує гнучкість та ефективність алгоритмів для рішення задачі пошуку та аналізу тексту.

**ВИКОРИСТАНІ ЛІТЕРАТУРНІ ДЖЕРЕЛА**

1. Ахо А. В., Корасік М. Дж. "Ефективний алгоритм пошуку в тексті". – М.: Мир, 1975. – 256 с.
2. Наварро Г., Раффінот М. "Гнучкий пошук у текстових рядках: практичний підхід до індексування інформації". – Харків: Фактор, 2002. – 312 с.
3. Гусфілд Д. "Алгоритми на рядках, деревах і послідовностях: комп'ютерне вирішення проблем в біоінформатиці". – Київ: Академперіодика, 2007. – 684 с.
4. Левенштейн В. І. "Двоїсті коди з виправленням спотворень". – М.: Радіо і зв'язок, 1966. – 196 с.
5. Вагнер Р. А., Фішер М. Дж. "Проблема найближчих спільних підрядків". – М.: Наука, 1974. – 478 с.