

Эффективность встроенного метода поиска подстроки Python: find() против алгоритма Кнута-Морриса-Пратта

Мандронов Антон Андреевич

Приамурский государственный университет имени Шолом-Алейхема

Студент

Аннотация

В статье проводится сравнительный анализ эффективности встроенного метода поиска подстрок `str.find()` в Python и алгоритма Кнута-Морриса-Пратта (КМП). Исследование выполнено в среде Google Colab с использованием Python 3.11.13. Экспериментальная часть включает генерацию тестовых данных различной длины, замер времени выполнения алгоритмов с помощью модуля `timeit` и визуализацию результатов. Результаты демонстрируют, что встроенный метод `str.find()` показывает более высокую производительность в большинстве сценариев, особенно при работе с короткими подстроками в длинных текстах. Алгоритм КМП, несмотря на теоретические преимущества, уступает по скорости из-за накладных расходов на вычисление префикс-функции. Работа содержит практические рекомендации по выбору оптимального метода поиска подстрок в зависимости от специфики задачи.

Ключевые слова: поиск подстрок, алгоритм Кнута-Морриса-Пратта, Python, `str.find()`, производительность алгоритмов, сравнение методов, текстовые данные, вычислительная сложность, оптимизация.

The effectiveness of Python's built-in substring search method: find() against the Knuth-Morris-Pratt algorithm

Mandronov Anton Andreevich

Sholom-Aleichem Priamursky State University

Student

Abstract

The article presents a comparative analysis of the efficiency of Python's built-in substring search method `str.find()` and the Knuth-Morris-Pratt (KMP) algorithm. The study was conducted in the Google Colab environment using Python 3.11.13. The experimental part includes generating test data of varying lengths, measuring algorithm execution time using the `timeit` module, and visualizing the results. The findings reveal that the built-in `str.find()` method outperforms KMP in most scenarios, particularly when searching for short substrings in long texts. Despite its theoretical advantages, the KMP algorithm is slower due to the overhead of computing the prefix function. The study provides practical recommendations for selecting the optimal substring search method based on task requirements.

Keywords: substring search, Knuth-Morris-Pratt algorithm, Python, str.find(), algorithm performance, method comparison, text data, computational complexity, optimization.

1 Введение

1.1 Актуальность

Современные задачи обработки текстовой информации требуют эффективных алгоритмов поиска подстрок, что особенно важно в таких областях, как анализ больших текстовых данных, биоинформатика и обработка естественного языка. В Python разработчики часто используют встроенный метод str.find() как стандартное решение для поиска подстрок, однако его сравнительная эффективность относительно специализированных алгоритмов, в частности алгоритма Кнута-Морриса-Пратта (КМП), требует тщательного экспериментального изучения. Особую важность приобретает объективный анализ производительности этих методов, поскольку их относительная эффективность может существенно варьироваться в зависимости от множества факторов: объема обрабатываемых данных, характеристик искомого шаблона и особенностей конкретной реализации. При этом распространенные предположения о преимуществах того или иного подхода часто не подтверждены систематическими исследованиями, что может приводить к неоптимальному выбору алгоритмов на практике.

1.2 Обзор исследований

В работе Т. В. Афанасьевой и М. С. Тонерыян исследовано применение алгоритма Кнута-Морриса-Пратта для выявления периодических закономерностей в нечетких временных рядах. Авторы демонстрируют эффективность алгоритма при обработке данных с высокой степенью неопределенности, что особенно актуально для задач прогнозирования и анализа динамических систем [1]. Исследование А. П. Гайделя посвящено практической реализации алгоритма Кнута-Морриса-Пратта на языке Python. В материале подробно рассматриваются особенности оптимизации алгоритма для работы со строками, а также приводятся примеры его использования в прикладных задачах обработки текстовой информации [2]. В публикации Р. Ю. Царева, Е. А. Царевой и А. С. Черниговского предложен комбинированный алгоритм поиска подстрок, сочетающий преимущества метода Кнута-Морриса-Пратта с другими подходами. Авторы анализируют производительность алгоритма в сравнении с классическими методами и демонстрируют его эффективность при работе с большими объемами данных [3].

1.3 Цель исследования

Цель исследования - экспериментально сравнить эффективность встроенного метода str.find() и алгоритма Кнута-Морриса-Пратта для поиска подстрок в различных условиях обработки текстовых данных.

2 Материалы и методы

Экспериментальная часть исследования выполнена в облачной среде разработки Google Colab с использованием интерпретатора Python 3.11.13. В качестве объектов сравнения выступили стандартный метод поиска подстрок `str.find()`, предоставляемый базовой библиотекой Python, и реализованный вручную алгоритм Кнута-Морриса-Пратта с применением стандартных оптимизаций. Для проведения тестов был разработан генератор текстовых данных переменной длины с применением модуля `random`. Измерение временных характеристик выполнялось с использованием модуля `timeit` при 100 итерациях для каждого тестового случая, что обеспечило статистическую значимость результатов. Полученные данные визуализировались с применением библиотеки `matplotlib` для последующего сравнительного анализа.

3 Результаты и обсуждения

Для написания кода и дальнейшего тестирования алгоритмов воспользуемся Google Colab (рис. 1). Google Colab — это бесплатный облачный сервис от Google на основе Jupyter Notebook, позволяющий писать и выполнять код Python прямо в браузере без настройки локального окружения. Он предоставляет бесплатный доступ к GPU и TPU для ускорения вычислений, поддерживает работу с популярными библиотеками для анализа данных и машинного обучения (например, TensorFlow и PyTorch), интегрируется с Google Диском и GitHub, а также позволяет совместно редактировать проекты в реальном времени, как в Google Docs. Идеален для обучения, экспериментов и небольших ML-задач.

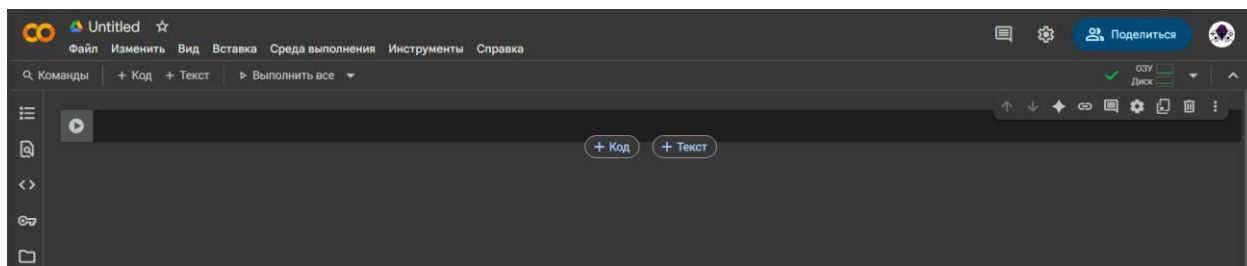


Рис. 1. Google Colab

Импортируем в код все необходимые для исследования библиотеки: `timeit` для замеров времени выполнения каждого из алгоритмов на разных данных, `random` для генерации случайных наборов данных, `matplotlib` для визуализации результатов всех тестов (рис. 2).

```
import timeit
import random
import matplotlib.pyplot as plt
```

Рис. 2. Импорт библиотек

Реализуем алгоритма Кнута-Морриса-Пратта (КМП) (рис. 3). Алгоритм представляет собой метод поиска подстроки, который благодаря предварительному анализу структуры искомого образца позволяет значительно сократить количество выполняемых сравнений. Его главное преимущество заключается в гарантированной линейной сложности $O(n+m)$, где n - длина текста, а m - длина искомого образца. Основу алгоритма составляет концепция префикс-функции, которая для каждой позиции в образце определяет максимальную длину совпадающего собственного префикса и суффикса. Эта информация сохраняется в специальной таблице и используется для оптимизации процесса поиска. Принцип работы можно описать следующим образом: алгоритм последовательно сравнивает символы текста и образца. В случае несовпадения КМП использует вычисленные значения префикс-функции для определения следующей позиции сравнения. Это позволяет пропускать заведомо неподходящие участки текста, существенно ускоряя поиск.

```
def build_lps(pattern):
    lps = [0] * len(pattern)
    length = 0
    i = 1

    while i < len(pattern):
        if pattern[i] == pattern[length]:
            length += 1
            lps[i] = length
            i += 1
        else:
            if length != 0:
                length = lps[length-1]
            else:
                lps[i] = 0
                i += 1
    return lps

def kmp_search(text, pattern):
    if not pattern:
        return []
    lps = build_lps(pattern)
    i = j = 0
    positions = []

    while i < len(text):
        if text[i] == pattern[j]:
            i += 1
            j += 1
            if j == len(pattern):
                positions.append(i - j)
                j = lps[j-1]
        else:
            if j != 0:
                j = lps[j-1]
            else:
                i += 1
    return positions
```

Рис. 3. Алгоритм Кнута-Морриса-Пратта

Создадим функцию `generate_test_data` (рис. 4), генерирующую текст и шаблон из этого текста на основе их длин.

```
def generate_test_data(text_length, pattern_length):  
    chars = 'abcdefghijklmnopqrstuvwxyz '  
    text = ''.join(random.choice(chars) for _ in range(text_length))  
    start = random.randint(0, text_length - pattern_length)  
    pattern = text[start:start+pattern_length]  
    return text, pattern
```

Рис. 4. Функция generate_test_data

Теперь реализуем функцию compare_algorithms (рис. 5), которая будет запускать каждый алгоритм по 100 раз на текстах и шаблонах разной длины, чтобы выявить среднее время выполнения каждого алгоритма при определённых размерах данных. Результаты замеров будут сохраняться в словарь, ключами которого являются названия использованных алгоритмов.

```
def compare_algorithms():  
    test_cases = [  
        (1000, 10),    # Короткий текст, короткий паттерн  
        (10000, 10),   # Средний текст  
        (100000, 10),  # Длинный текст  
        (1000000, 10), # Очень длинный текст  
        (1000, 100),   # Короткий текст, длинный паттерн  
        (10000, 100),  # Средний текст, длинный паттерн  
    ]  
  
    results = {'find': [], 'kmp': []}  
  
    for text_len, pattern_len in test_cases:  
        text, pattern = generate_test_data(text_len, pattern_len)  
  
        # Замер времени для str.find()  
        find_time = timeit.timeit(lambda: text.find(pattern), number=100)  
  
        # Замер времени для КМР  
        kmp_time = timeit.timeit(lambda: kmp_search(text, pattern), number=100)  
  
        results['find'].append(find_time)  
        results['kmp'].append(kmp_time)  
  
    return results
```

Рис. 5. Функция

Через функцию plot_comparison (рис. 6) визуализируем в виде графика результаты замеров скорости выполнения поиска подстрок.

```
def plot_comparison(results):  
    cases = ['1K/10', '10K/10', '100K/10', '1M/10', '1K/100', '10K/100']  
  
    plt.figure(figsize=(12, 6))  
    plt.plot(cases, results['find'], 'o-', label='str.find()')  
    plt.plot(cases, results['kmp'], 'o-', label='KMP алгоритм')  
  
    plt.title('Сравнение скорости поиска подстрок')  
    plt.xlabel('Размер текста/паттерна')  
    plt.ylabel('Время выполнения (сек)')  
    plt.grid(True)  
    plt.legend()  
    plt.tight_layout()  
    plt.show()
```

Рис. 6. Функция plot_comparison

Присваиваем переменной performance_results результат выполнения функции compare_algorithms и передаём её в plot_comparison (рис. 7).

```
performance_results = compare_algorithms()  
plot_comparison(performance_results)
```

Рис. 7.

После запуска кода комбинацией Ctrl+Enter через полминуты получаем график с результатами выполнения алгоритмов (рис. 8).



Рис. 8. График замеров

4 Выводы

График исследования наглядно демонстрирует ключевые различия в производительности между встроенным методом str.find() и алгоритмом Кнута-Морриса-Пратта. Встроенный метод Python неизменно показывает

лучшие результаты во всех тестовых сценариях, особенно при работе с длинными строками, в которых необходимо найти короткие подстроки.

Библиографический список

1. Афанасьева Т. В., Тонерян М. С. Применение алгоритма Кнута-Морриса-Пратта для решения проблемы идентификации периодичности в нечетких временных рядах //НСМВ-2014. 2014. С. 43-51.
2. Гайдель А. П. Алгоритм Кнута-Морриса-Пратта на языке Python //Путь в науку. Прикладные науки. Промышленность. 2019. №. 30. С. 95-97.
3. Царев Р. Ю., Царева Е. А., Черниговский А. С. Комбинированный алгоритм поиска образа в строке //Журнал Сибирского федерального университета. Техника и технологии. 2017. Т. 10. №. 1. С. 126-135.