Работу выполнил студент 1 курса
магистратуры (группа M4130) ИТМО:
Меньщиков Михаил

Отчёт:

Лабораторная №1

Дисциплина:

Генетические алгоритмы

ИТМО 2024

Содержание:

1 Постановка задания

Цель: Получить навыки вычисления сложности алгоритмов и их оптимизации различными методами.

Задачи:

1. Реализовать на любом ЯП алгоритм, согласно варианту задания.
2. Вычислить сложность алгоритма, привести расчёты, результаты нагрузочных тестов с замером затраченного времени и ресурсов.
3. Выполнить оптимизацию как алгоритмическую если возможно, с выносом инварианта, например, так и программными методами выбранного ЯП.
4. Вычислить сложность оптимизированного алгоритма, привести расчёты, результаты нагрузочных тестов с замером затраченного времени и ресурсов.
5. Описать различие величин сложности, результатов, привести обоснование.
6. Сформулировать выводы.
7. Приложить код в виде ссылки на публичный репозиторий.

Реализуемый вариант задания: 16.

2 Решение

2.1 Наивная реализация алгоритма

Разбор алгоритма Кнута-Мориса-Пратта по поиску подстроки в строке можно найти по следующей ссылке: https://www.youtube.com/watch?v=7g-WEBj3igk. На Рисунке 1 представлена реализация данного алгоритма. Из-за двух последовательных циклов временная сложность алгоритма составляет $O(N + M)$, где N - длина *pattern*-подстроки, которую мы ищем; M - длина *sequence*-строки, в которой мы ищем. Сложность алгоритма по потреблению памяти (*Space Complexity*) составляет $O(N)$.

```
7   def kmp(sequence, pattern):
8       # Алгоритм Кнута-Мориса-Пратта поиска подстроки в строке.
9       # Разбор: https://www.youtube.com/watch?v=7g-WEBj3igk .
10      #
11      # Сложность алгоритма: O(n+m), где
12      # n - длина строки pattern;
13      # m - длина строки sequence.
14
15      # Первый этап: формирование массива pi
16      pi = [0] * len(pattern)
17      i, j = 1, 0
18      while i < len(pattern):
19          if pattern[i] == pattern[j]:
20              pi[i] = j+1
21              i+= 1
22              j+= 1
23          elif j == 0:
24              pi[i] = 0
25              i+= 1
26          else:
27              j = pi[j-1]
28
29      # Второй этап: поиск образа pattern в строке sequence
30      k, l = 0, 0
31      n, m = len(pattern), len(sequence)
32      while k < m:
33          if sequence[k] == pattern[l]:
34              k += 1
35              l += 1
36              if l == n:
37                  return "Образ найден"
38          elif l == 0:
39              k += 1
40              if k == m:
41                  return "Образ отсутствует"
42          else:
43              l = pi[l-1]
```

Рисунок 1. Наивная реализация КМП-алгоритма

Зависимость времени работы КМП-алгоритма от размера sequence-строки представлена на Рисунке 2. Зависимость времени работы КМП-алгоритма от размера *pattern*-подстроки представлена на Рисунке 3.

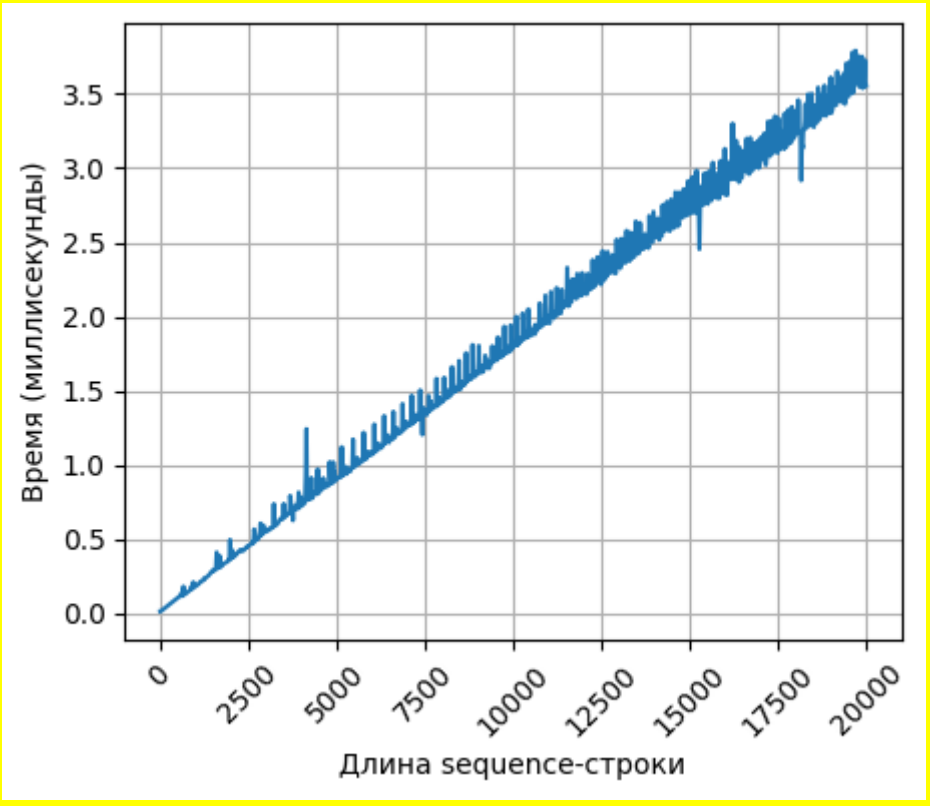

Рисунок 2. Зависимость времени работы от размера sequence-строки наивной реализации КМП-алгоритма. Длина pattern-строки зафиксирована и равна 10

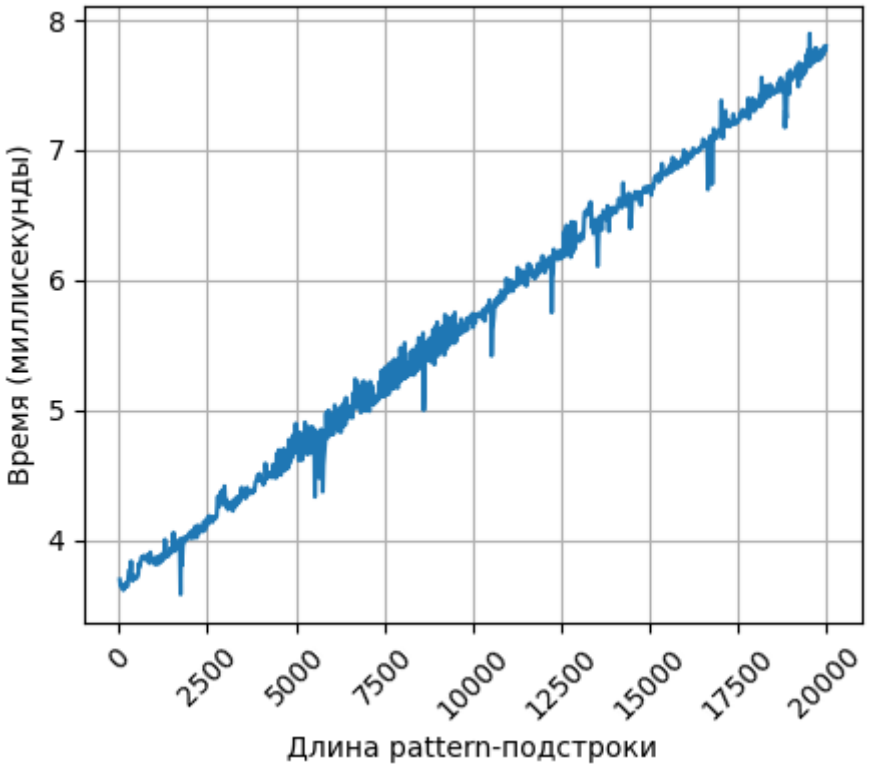

Рисунок 3. Зависимость времени от размера pattern-строки наивной реализации КМП-алгоритма. Длина sequence-строки зафиксирована и равна 20000

2.2 Оптимизация алгоритма

На Рисунке 4 представлен вариант оптимизации КМП-алгоритма с помощью инструментария из программного пакета *numba*.

```python
from numba import njit

@njit
def kmp(sequence, pattern):
    # Алгоритм Кнута-Мориса-Пратта поиска подстроки в строке.
    # Разбор: https://www.youtube.com/watch?v=7g-WEBj3igk .
    #
    # Сложность алгоритма: O(n+m), где
    # n - длина строки pattern;
    # m - длина строки sequence.

    # Первый этап: формирование массива pi
    pi = [0] * len(pattern)
    i, j = 1, 0
    while i < len(pattern):
        if pattern[i] == pattern[j]:
            pi[i] = j+1
            i+= 1
            j+= 1
        elif j == 0:
            pi[i] = 0
            i+= 1
        else:
            j = pi[j-1]

    # Второй этап: поиск образа pattern в строке sequence
    k, l = 0, 0
    n, m = len(pattern), len(sequence)
    while k < m:
        if sequence[k] == pattern[l]:
            k += 1
            l += 1
            if l == n:
                return "Образ найден"
        elif l == 0:
            k += 1
            if k == m:
                return "Образ отсутствует"
        else:
            l = pi[l-1]
```

Рисунок 4. Оптимизированная версия КМП-алгоритма с помощью пакета *numba*

Зависимость времени работы оптимизированного КМП-алгоритма от размера sequence-строки представлена на Рисунке 5. Зависимость времени работы оптимизированного КМП-алгоритма от размера *pattern*-подстроки представлена на Рисунке 6.

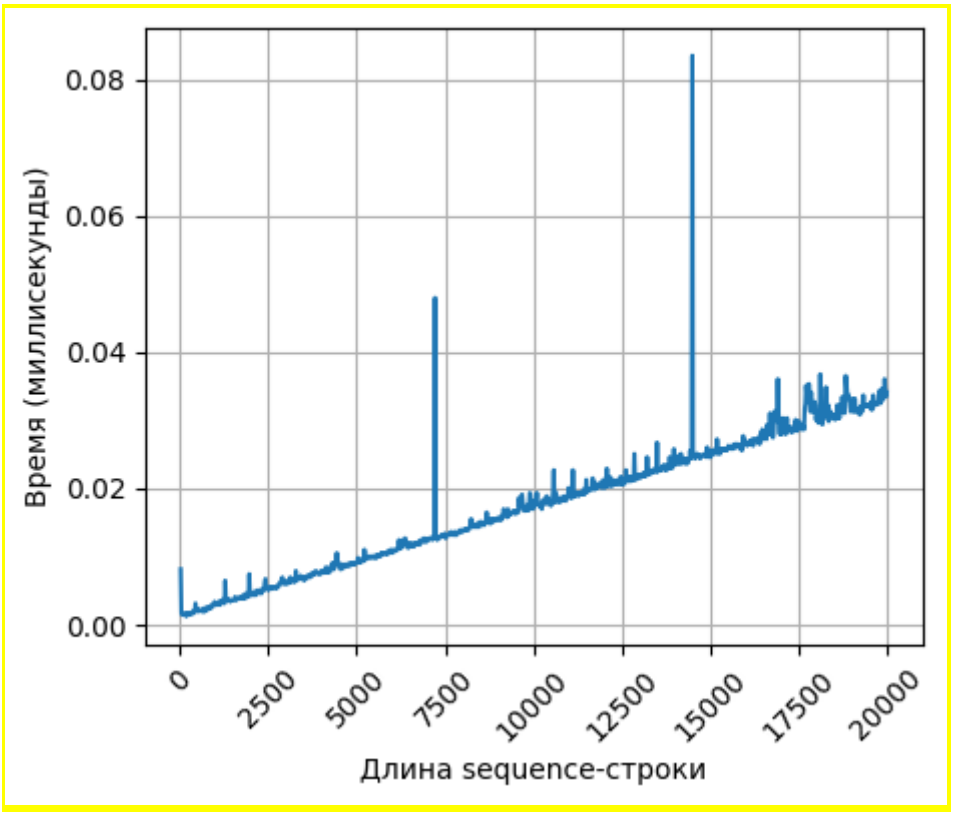

Рисунок 5. Зависимость времени работы от размера sequence-строки оптимизированной реализации КМП-алгоритма. Длина pattern-строки зафиксирована и равна 10

Рисунок 6. Зависимость времени от размера pattern-строки оптимизированной реализации КМП-алгоритма. Длина sequence-строки зафиксирована и равна 20000

2.3 Выводы

В результате нагрузочного тестирования двух вариантов реализации КМП-алгоритма наивное решение удалось ускорить (в среднем), примерно, в 89 раз: сравнительные графики представлены на Рисунке 7 и Рисунке 8.

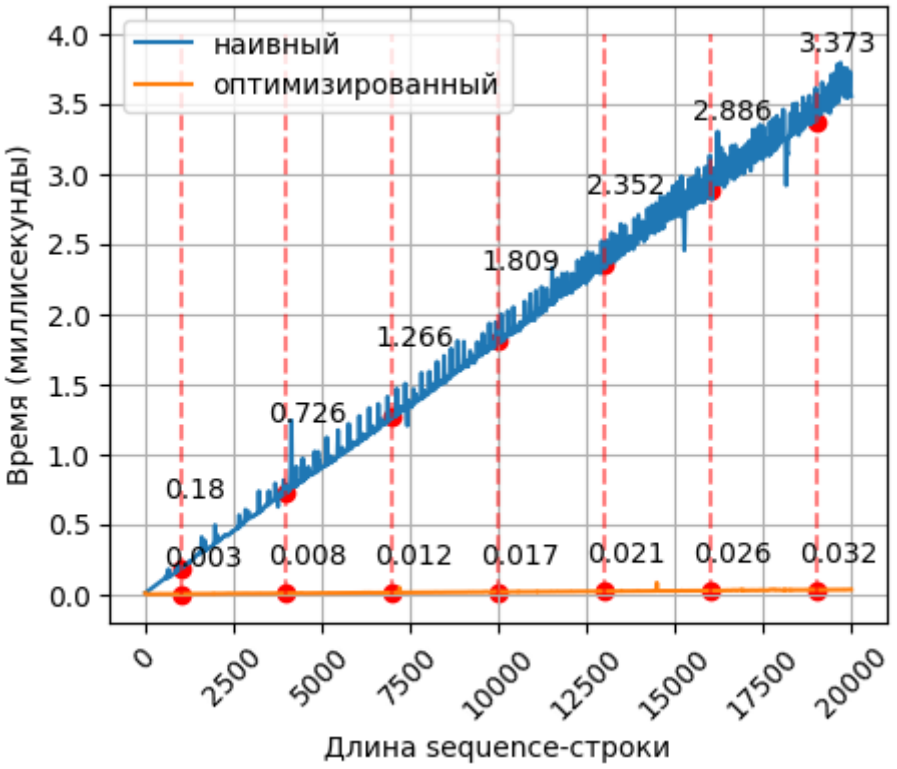

Рисунок 7. Сравнение времени работы наивного и оптимизированного КМП-алгоритма при фиксированной длине pattern-подстроки (10)

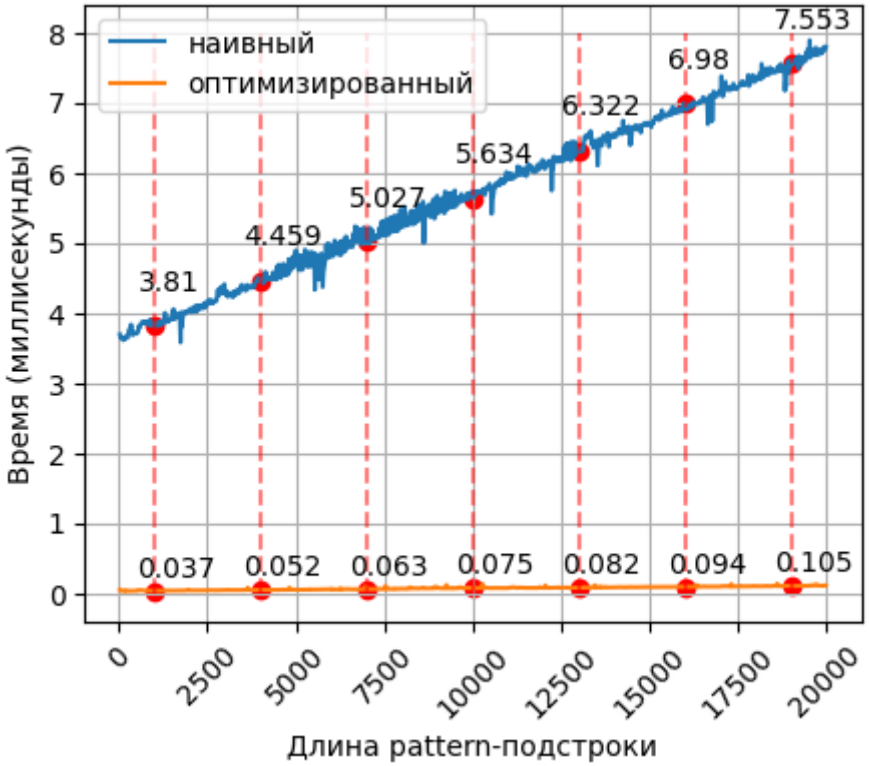

Рисунок 8. Сравнение времени работы наивного и оптимизированного КМП-алгоритма при фиксированной длине sequence-строки (20000)

Numba — это *Just-In-Time* компилятор, который превращает ваш код на питоне в машинный код на лету. Это не просто мелкая оптимизация, а серьёзно ускорение. Если вы знакомы с интерпретируемыми языками, вы знаете, что они обычно медленнее компилируемых из-за необходимости анализировать и исполнять код на лету. Но что, если бы вы могли получить лучшее из обоих миров? JIT-компиляция позволяет интерпретируемому языку, каким является питон, динамически компилировать части кода в машинный код, значительно ускоряя исполнение. Numba использует этот подход, чтобы помочь вашему коду на питоне быть быстрей. Она анализирует вашу функцию, определяет типы данных и затем компилирует её в оптимизированный машинный код. И всё это происходит во время выполнения вашего кода. (Ссылка на источник: https://habr.com/ru/companies/otus/articles/784068/)

Ссылка на репозиторий с кодом данной лабораторной работы: https://github.com/Dzigen/EvalAlg_lab1.

3 Заключение

В результате проделанной работы получены навыки вычисления сложности алгоритмов и их оптимизации различными методами. Для достижения поставленной цели были решены следующие задачи:

- Реализован алгоритм на языке Python, согласно варианту 16.
- Вычислена сложность алгоритма, приведены расчёты, результаты нагрузочных тестов.
- Выполнена оптимизация реализованного алгоритма.
- Вычислена сложность оптимизированного алгоритма, приведены расчёты, результаты нагрузочных тестов с замером затраченного времени и ресурсов.
- Описано различие величин сложности, результатов, приведено обоснование.
- Сформулированы выводы.
- Приложен код в виде ссылки на публичный репозиторий.