Lab01: Сложность алгоритмов и их оптимизация

Вариант 24: Алгоритм Бойера-Мура

Алгоритм

Алгоритм Бойера-Мура используется для поиска всех вхождений строки в тексте. Паттерн выравнивается с текстом в некоторой позиции, и происходит его сравнение в каждом символе с права налево. При совпадении подстрок в каждом символе - объявляем о найденном вхождении. При обнаружении несовпадения сдвигаем паттерн на ≥ 1 позиции к концу строки. Величина сдвига определяется несколькими эвристиками.

Эвристика плохого символа

При обнаружении несовпадения в текущем выравнивании $s[i + j] \neq t[j]$ мы понимаем, что вхождения подстроки в данном индексе нет. Однако вместо того, чтобы сдвинуть паттерн на 1 позицию вправо, мы можем определить самое правое вхождение символа $s[i + j]$ в паттерн, а затем сдвинуть паттерн так, чтобы нужный символ оказался напротив $s[i + j]$.

Предпосчет за $O(m)$

Эвристика совпавшего суффикса

Если паттерн представляет собой строку вида "$AaSCbS$", а строка при текущем выравнивании выглядит как "$* \neg b S$", то мы можем сдвинуть паттерн на $len(CbS)$. При этом мы будем знать, что подстрока паттерна совпадает с подстрокой текста. Для этого храним массив $$suffshift[n] = \min {i: t[-n:] = t[-i: -(i - n)] & t[-(i+1)] \neq t[-(n+1)]}$$

Такой массив можно предпосчитать за $O(m)$ с помощью Z-функции.

Правило Галиля

Правило Галиля позволяет не проверять префикс паттерна при сдвиге после нахождения очередного вхождения. Такой сдвиг происходит на величину $suffshift[0]$, а значит мы знаем, что префикс паттерна $\ t[:-(suffshift[0] + 1)]$ совпадает со строкой и можно не проводить сравнения.

Анализ времени работы

В базовом варианте время работы алгоритма в лучшем случае составляет $T_{best}(n, m) = O(n + m)$ - нет вхождений подстроки.

В худшем $T_{worst}(n, m) = O(n \cdot m)$, $t = "a"*m, s = "a"* n$

Однако можно показать, что при добавлении правила Галиля время работы всегда будет составлять $T(n, m) = O(n + m)$.

Это делается за счет оценки количества повторных сравнений символов и дает оценку $4n$ (если просто, доказывается и $3n$)на количество сравнений при поиске.

Реализованные алгоритмы

- Классический алгоритм: реализованы все эвристики, в том числе правило Галиля на чистом python.
- Параллельная реализация:
    - Разбиваем входную строку на $k = \lfloor n / m \rfloor$ строк длины $2m$ с перекрытием длины m.
    - Получаем k задач с размерами $(2m, m) \Rightarrow T_i = T(2m, m) = O(m)$
    - Задачи между собой не пересекаются и все ответы можно слить в один для исходной строки за линию.
    - Если каждую задачу запустить на своем процессоре, то можно найти все вхождения за $O(m)$.
- Cython реализация: все вычисления, описанные на python транслировали в C и скомпилировали.

Результаты тестов

Тестовые данные генерировались случайно в количестве 100000. Длина строки-образца выбиралась из $Uni[1, 100]$, а строки-текста из $Binom(1000, 0.5)$.

Algorithm	**Time**	**Memory**
base	4e-5	78820
parallel	111e-5	79188
cython	2e-5	79204

Как видно из приведенной таблицы, несмотря на теоретическое улучшение производительности в параллельном варианте алгоритма, отсутствие достаточного количества процессоров, а также большая константа, возникающая из создания пула потоков для каждой пары строки и образца, ухудшают время работы.

При этом чисто программная оптимизация, заключающаяся в трансляции питоновского кода в C, с дальнейшей компиляцией позволяют ускорить алгоритм вдвое относительно базовой версии.

В ходе тестов обе оптимизации показали ухудшение в количестве используемой памяти, однако оно не является настолько явным как изменение времени работы.

Выводы

- Алгоритм является эффективным с точки зрения теоретической сложности.
- Параллелизация алгоритма не принесла никакого результата из-за технических ограничений, порождающих большую константу, не позволившую обогнать базовую реализацию.
- Чисто программная оптимизация, позволившая транслировать алгоритм в C, вдвое ускорила время выполнения.