Київський національний університет імені Тараса Шевченка
Факультет комп’ютерних наук та кібернетики
Кафедра математичної інформатики

Курсова робота на тему:

Розробка методики порівняння ефективності алгоритмів патерн-метчингу

Виконав студент 3 курсу
Гришечкін Тихон Сергійович

Науковий керівник:
доцент, доктор фізико-математичних наук Завадський І.О

Київ, 2024

Зміст

1 Вступ

Задача пошуку рядку є важливою задачею в інформатиці. Вона має широке застосування в різних областях, зокрема:

- Інформаційний пошук: Алгоритми пошуку рядків використовуються в пошукових системах для знаходження релевантних документів на основі запитів користувачів.
- Біоінформатика: Пошук підрядків є ключовим у знаходженні послідовностей нуклеотидів або амінокислот у геномах та протеомах. [1]
- Обробка текстів: Алгоритми пошуку використовуються для знаходження та заміни текстових фрагментів у текстових редакторах.
- Бази даних: Пошук підрядків використовується в базах даних для виконання запитів зі згадками текстових фрагментів або шаблонів.
- Компіляція: Компілятори використовують алгоритми пошуку підрядків для аналізу та оптимізації програмного коду.

Тому важливою задачею є оцінка ефективності цих алгоритмів. Оцінка ефективності дозволяє визначити, який з алгоритмів найкраще підходить для конкретних умов та обмежень. У даній курсовій роботі ми запропонуємо методику порівняння ефективності різних алгоритмів пошуку рядку. Ми проведемо експериментальне дослідження на основі ряду критеріїв, таких як швидкість виконання, та стійкість до різних типів вхідних даних.

Результати даного дослідження можуть бути корисними для вибору оптимального алгоритму в залежності від конкретної задачі та умов її виконання.

2 Алгоритми пошуку рядка

2.1 Постановка задачі патерн-метчингу

Нехай дані дві послідовності: текст T та патерн P (далі також інколи позначається, як «шаблон»), де $|T| = n$ та $|P| = m$. Задача полягає у знаходженні всіх входжень патерна P у текст T.

При цьому передбачається, що:

- T та P складаються з символів алфавіту Σ.
- Алфавіт Σ може бути скінченним або нескінченним, але у нашій постановці розглядається скінченний алфавіт.

Розглядається задача пошуку підстроки у формулюванні **single-pattern matching**, що передбачає пошук одного заданого патерна в тексті. Не розглядаються задачі **approximate pattern searching**, де потрібно знаходити патерни з певною степенню схожості, або **regular pattern searching**, де патерн може бути виражений у вигляді регулярного виразу.

2.2 Загальна характеристика алгоритмів пошуку рядка

Найпростішим методом для пошуку підрядка в рядку є алгоритм повного перебору (brute force). Його суть полягає у перевірці кожної позиції в тексті T як можливої початкової позиції для патерна P. Для кожної такої позиції перевіряється, чи співпадає патерн P з відповідною підстрокою в тексті T. Цей метод має часову складність $O(n \cdot m)$, де n – довжина тексту, а m – довжина патерна. Через високу обчислювальну складність цей алгоритм є дуже неоптимальним для великих текстів і патернів.

У 1970 році Д. Морріс та В. Пратт [2] розробили перший оптимальний алгоритм пошуку підстроки – алгоритм Кнута-Морріса-Пратта (КМР). Цей алгоритм значно покращив ефективність пошуку, зменшивши часову складність до $O(n + m)$.

З часом з'явилося багато інших алгоритмів (більше 124 див. [3]). Всі ці алгоритми можна умовно поділити на чотири основні категорії:

- алгоритми на основі порівняння символів

- алгоритми на основі суфіксних автоматів
- алгоритми на основі побітового паралелізму
- гібридні алгоритми.

Алгоритми на основі порівняння символів

Алгоритми на основі порівняння символів (character-based approaches або comparison based approaches) є класичним підходом, який просто порівнює символи для вирішення задач пошуку рядка.

Алгоритми на основі порівняння символів мають дві основні стадії: пошук і зсув. Алгоритм Бойера-Мура [5], є стандартним та еталонним методом цього типу. Він використовує таблицю зсувів для пропуску непотрібних порівнянь, що значно підвищує швидкість пошуку.

Алгоритми на основі суфіксних автоматів

Суфіксний автомат (Suffix automaton) складається з детермінованого ациклічного кінцевого автомата та суфіксного автомата, які разом дозволяють ефективно здійснювати пошук рядка.

Цей метод використовує спрямований ациклічний граф, де вершини є станами, а ребра – переходами між станами. Суфіксний автомат розпізнає всі суфікси патерна і дозволяє швидко знаходити підрядки. Прикладом алгоритму даного класу є є Backward-DAWG-Matching алгоритм (BDM).

Алгоритми на основі побітового паралелізму

Алгоритми на основі паралелізму бітового рівня були запропоновані для прискорення процесу пошуку шляхом використання внутрішнього паралелізму бітових операцій, що зменшує кількість операцій у рази до ω, де ω – кількость бітів у машинному слові.

Ці алгоритми є швидкими та ефективними, особливо коли довжина патерна менша за довжину машинного слова. До таких алгоритмів належать наприклад Shift-OR (SO), Shift-AND (SA) алгоритми.

Гібридні алгоритми

Гібридні алгоритми комбінують переваги різних методів для вирішення складних задач. Вони поєднують методи з різних категорій для досягнення кращої ефективності.

Таким чином, за роки розвитку цієї області задач з’явилося багато алгоритмів для пошуку підрядка, які можна класифікувати за чотирма основ-

ними категоріями. Кожен клас алгоритмів має свої недоліки та переваги, які важливо розуміти при виборі оптимального алгоритму для своєї задачі.

2.3 Алгоритм Бойера-Мура

Розглянемо детальніше деякі алгоритми пошуку.

Першим розглянемо алгоритм Бойера-Мура (BM), який є класичним представником класу «character-based» алгоритмів.

Алгоритм шукає входження патерну **p**, прикладаючи його до тексту **t** зліва направо, але при цьому порівняння, виконуються справа наліво, починаючи з останнього символу патерну.

Далі якщо при порівнянні знайдено перший неспівпадаючий символ патерну, то виконується зсув патерну на деяку кількість позицій, за допомогою наступних евристик:

- *Евристика стоп символів*:

  Для цього для кожного символу алфавіту Σ запам'ятовуємо індекс останнього його входження в патерн, якщо він є, або запам'ятовуємо -1, якщо він не входить в патерн (вважаємо, що індексація символів починається з нуля). Назвемо цей масив індексів $StopTable$.

  Тепер розглянемо символ тексту c, що не співпав на поточній ітерації порівняння. Позначимо поточний індекс патерну на якому відбулось неспівпадіння як i. Тоді якщо виконується

  $$i - StopTable[c] > 0$$

  то відбувається зсув патерну на $i - StopTable[c]$ кроків, інакше кажучи на найменшу кількість кроків щоб відбулось співпадіння з символом c. (рис. 1)

- *Евристика співпадаючого суфікса*:

  Якщо при порівнянні шаблону справа наліво збігся суфікс S, а символ c, що стоїть перед S в шаблоні не збігся (тобто шаблон має вигляд PcS), то евристика співпадаючого суфікса зсуває шаблон на найменше число позицій вправо так, щоб рядок S збігся з шаблоном, а символ, що передує даному збігу S в шаблоні, відрізнявся б від c.

Заради цього для патерну p перед початком пошуку, обраховуються відповідні величини зсуву, за допомогою алгоритму, схожого за ідеєю на алгоритми обчислення «префікс-функції» та «Z-функції».

Як і випадку цих алгоритмів ці зсуви обчислюються за $O(m)$ операцій.

Далі в алгоритмі просто використовуються ці заздалегідь обчислені зсуви при неспівпадінні символів.

Зазначимо, що якщо на кожній ітерації просто знову порівнювати символи, навіть з використанням зсувів асимптотика алгоритму складає $O(n*m)$ операцій. Для досягнення складності $O(n+m)$, треба не порівнювати з шаблоном вже співпадаючий суфікс, знайдений на попередній ітерації.

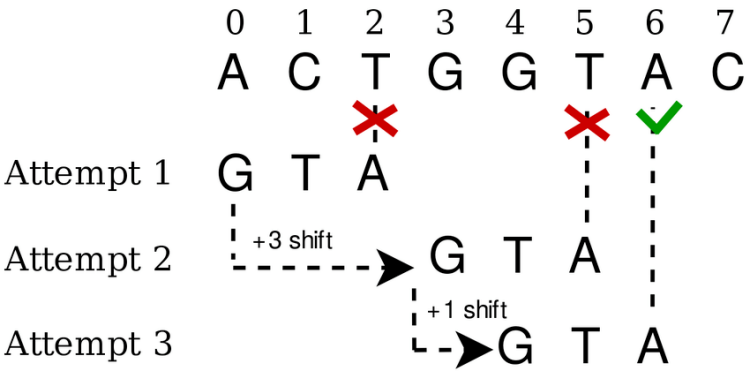

Рис. 1: Приклад використання евристики стоп символів

Рис. 2: Приклад використання евристики співпадаючого суфікса

Додамо також, що існує багато модифікацій алгоритму Бойера-Мура, які використовують інші евристики або реалізують вже існуючі інакше.

Прикладами таких алгоритмів є алгоритми Бойєра - Мура - Хорспула (BMH), Чжу - Такаокі (ZT), турбо - Бойера - Мура (TBM).

Наприклад BMH використвує тільки «евристику стоп символів», але тільки за останнім символом, тому є спрощени варіантом Бойера-Мура.

ZT використовує також «евристику стоп символів», але для пар символів, заради оптимізації при маленьких алфавітах.

Зазначимо також, що довгі патерни зазвичай дають довші зсуви, тому алгоритми класу «порівняння символів» добре працюють на таких патернах.

2.4 Алгоритми Shift-OR (SO) та Shift-AND (SA)

Розглянемо алгоритми Shift-OR (SO) та Shift-AND (SA), які є представниками класу алгоритмів основаних на паралелізмі бітових операцій.

Опишемо SA алгоритм.

Нехай маємо i-й символ тексту t.

Введемо матрицю $Mask$ (рис 3), де

$$Mask[i][j] = 1, \text{якщо } t[i-j..i] = p[0..j]$$

$$Mask[i][j] = 0, \text{якщо } t[i-j..i] \neq p[0..j]$$

Тоді маємо входження шаблону на i-му кроці, коли $Mask[i][m-1] = 1$.

			X	A	B	X	A	B	A	A	X	A
		0	1	2	3	4	5	6	7	8	9	10
A	1	0	0	1	0	0	1	0	1	1	0	1
B	2	0	0	0	1	0	0	1	0	0	0	0
A	3	0	0	0	0	0	0	0	1	0	0	0
A	4	0	0	0	0	0	0	0	0	1	0	0
C	5	0	0	0	0	0	0	0	0	0	0	0

Рис. 3: Приклад матриці $Mask$

Але вичислення матриці $Mask$, на кожному кроці займає $O(m)$ операцій тому такий алгоритм нічим не краще «Brute-Force» алгоритму, і фактично ним і є.

Тоді нехай маємо матрицю pos, де для символу c маємо

$$pos[c][j] = 1, \text{якщо } p[j] = c$$

$$pos[c][j] = 0, \text{якщо } p[j] \neq c$$

Можемо помітити наступну властивість.

$$Mask[i] = (Mask[i-1] << 1) \ \& \ pos[t[i]]$$

Де $<<$ - операція побітового зсуву вліво.

Тепер можемо в рамках реалізації замінити вектори $Mask[i]$ та $pos[c]$, на звичайні змінні, для яких всі операції будуть виконані $O(1)$, але отримаємо очевидне обмеження $m < \omega$, де ω - довжина машинного слова.

Shift-or алгоритм аналогічний алгоритму Shift-and, де просто інвертовано вектори $Mask[i]$ та $pos[c]$, а операція and замінюється на or. А критерієм входження стає $Mask[i][m-1] = 0$.

Алгоритми, що основані на паралелізмі бітових операцій називаються так бо на кожному кроці виконується ω паралельних порівнянь, хоча фактично виконується лише декілька бітових операцій.

Такі алгоритми дають значне прискорення на патернах невеликої довжини, для яких вони і були створені. Для $m > \omega$, зазвичай ці алгоритми оптимізують або об'єднують з іншими.

Наприклад у SA, для патернів більшої довжини, при знаходженні співпадіння довжини ω решту символів просто перевіряють до першого неспівпадіння.

Також перевагою цих алгоритмів є те, що вони легко модифікуються до «наближеного пошуку входження рядка» (approximate pattern searching). Ці алгоритми вбудовані у реалізацію Unix утиліти *agrep*, що і виконує цей наближений пошук.

2.5 Алгоритм Forward Dawg Matching

Розглянемо також алгоритм Forward Dawg Matching (FDM), що є представником класу «automata-based» алгоритмів.

Алгоритм Forward Dawg Matching обчислює найдовший підрядок ша-

блону, що закінчується на кожній позиції в тексті.

Це можливо завдяки використанню найменшого суфіксного автомата (також називається DAWG, Directed Acyclic Word Graph) для шаблону. Найменший суфіксний автомат слова w - це детермінований кінцевий автомат $S(w) = (Q, q_0, T, E)$. Мова, прийнята $S(w)$, це $L(S(w)) = \{u \in \Sigma^* : \exists v \in \Sigma^*$ така, що $w = vu\}$.

Підготовча фаза алгоритму Forward Dawg Matching полягає в обчисленні найменшого суфіксного автомата для шаблону x. Це займає лінійний час та простір додаткової пам'яті відносно довжини шаблону.

Під час фази пошуку алгоритм Forward Dawg Matching аналізує символи тексту зліва направо за допомогою автомата $S(x)$, починаючи з початкового стану q_0. Для кожного стану q у $S(x)$ найдовший шлях від q_0 до q позначається як length(q).

Структура автомата використовує поняття суфіксних посилань. Для кожного стану q суфіксне посилання q позначається як $S[q]$.

S[q] - посилання на стан автомата, що є найбільшим суфіксом стану q.

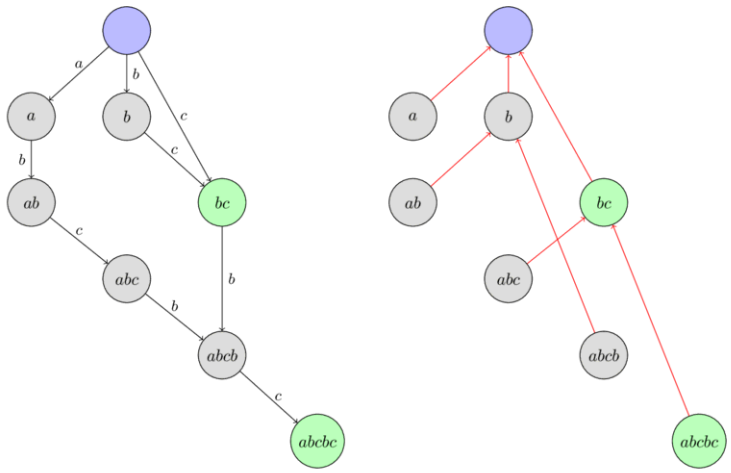

Рис. 4: Приклад суфіксного автомата рядка "abcbc", з його суфіксними посиланнями

Для стану p нехай Path$(p) = (p_0, p_1, \ldots, p_\ell)$ буде суфіксним шляхом p, таким чином що $p_0 = p$, для $1 \leq i \leq \ell$, $p_i = S[p_{i-1}]$ і $p_\ell = q_0$.

Для кожного символу тексту $t[j]$ послідовно, нехай p буде поточним станом, тоді алгоритм Forward Dawg Matching здійснює перехід, по символу $t[j]$ для першого стану Path(p), для якого такий перехід визначений. Поточний стан p оновлюється за допомогою цільового стану цього переходу або

початкового стану q_0, якщо не існує переходу по символу $t[j]$ з будь-якого стану Path(p).

Збіг шаблону x знайдено, коли length$(p) = m$, де $m = |x|$.

Недоліками алгоритмів, що основані на суфіксних автоматах є додаткові витрати на побудову автомата.

Перевагами таких алгоритмів є те що, їх можна реалізувати так, щоб співпадіння на кожному новому індексі тексту була знайдена простими переходами по автомату, за $O(1)$ операцій, що підходить для так званих «real-time-systems».

Також перевагою є те, що на кожному кроці алгоритм знаходить найбільший підрядок шаблону, що закінчується на поточній позиції тексту, а отже вирішує більш загальну задачу ніж пошук входжень патерна в текст.

3 Порівняння ефективності алгоритмів

3.1 Фактори впливу на ефективність алгоритмів

Одразу зазначимо, що не будемо порівнювати алгоритми за об'ємом зайнятої додаткової пам'яті, адже більшість алгоритмів займають $O(m)$, або $O(m*|\Sigma|)$ додаткової пам'яті, що є незначним показником при невеликих патернах та алфавітах на яких в більшості випадків виконують пошук.

Також зазначимо, що в цій роботі не будуть порівнюватись алгоритми, що виконують препроцесинг над текстом (так звані «offline exact string matching algorithms»), адже їх потрібно порівнювати окремо від «онлайн» алгоритмів.

Наведемо основні фактори впливу на швидкодію алгоритмів патерн метчингу:

- $|T|$ - довжина тексту.
- $|P|$ - довжина шаблону.
- $|\Sigma|$ - розмір алфавіту.
- Кількість різних підрядків P та T.
- Кількість входжень P в T.
- Граматика мови, до якої належить текст і шаблон (якщо існує).
- "Ворожість" користувача. Іншими словами: чи буде користувач навмисно задавати дані, на яких алгоритм повільно працюватиме?
- Архітектура процесора. Деякі процесори мають автоінкрементні або SIMD-операції, які дозволяють швидко порівняти дві ділянки ОЗП (наприклад, rep cmpsd x86). Деякі алгоритми використовують такі операції (SSE Filter Algorithm).
- Характер даних шаблона P у порівнянні з T. Наприклад якщо P - деяке число, що шукається в англійському тексті, то більшість $character - based$ алгоритмів оптимізує пошук використовуючи свої евристики зсуву.

Додамо що цей перелік не є кінцевим і можна навести ще багато інших факторів впливу.

3.2 Методика порівняння алгоритмів

У праці Ф. Саймона та Л. Тьєррі [6] та інших роботах з порівнянням алгоритмів патерн-метчингу, при порівнянні алгоритмів пошуку, увагу було зосереджено на великій кількості алгоритмів та різних можливих вхідних текстах. При тестуванні патерни завжди обирались випадковим чином із тексту.

Тому у власній роботі, я вирішив розширити множину можливих патернів і порівняння будуть виконуватись по наступним типам патернів:

- Випадкові патерни, заданого алфавіту.
- Патерни випадковим чином, обрані із заданого тексту.
- Патерни, що часто зустрічаються в тексті. Наприклад для тексту англійської літератури було обрано патерни найбільш повторюваних слів англійської мови. Аналогічно для української - найбільш повторюваних слів української мови. Для json бази даних - деякі ключові слова та значення полів, що часто зустрічаються.

Також зазначу, що не було спроби визначити ефективність алгоритмів, за рядом експериментів, окрім як просумувати абсолютні значення часу виконання алгоритмів.

Тому у власній роботі я вводжу поняття відносного результату роботи алгоритму $\sigma_{\text{алг}}$.

Нехай ми порівнюємо деяку множину алгоритмів. Нехай деякий алгоритм відпрацював найшивдше з усіх за час $t_{\min}$.

Тоді відносний результат алгоритму, що відпрацював за час $t_{\text{алг}}$, дорівнює:

$$\sigma_{\text{алг}} = \frac{t_{\text{алг}} - t_{\min}}{t_{\min}}$$

Введення відносних результатів дозволяє зробити загальний аналіз результатів роботи алгоритмів за декількома експериментами.

Наприклад маємо такі результати роботи алгоритмів (табл 1):

	Алгоритм 1	**Алгоритм 2**
Тест 1	40	20
Тест 2	500	700

Табл. 1: Абсолютні результати алгоритмів у мс

Тоді маємо наступні відносні результати роботи алгоритмів (табл 2):

	Алгоритм 1	**Алгоритм 2**
Тест 1	1	0
Тест 2	0	0.4

Табл. 2: Відносні результати алгоритмів σ

Сумарний час виконання 1 алгоритму - 540 мс. Сумарний середній відносний результат 0.5.

Сумарний час виконання 2 алгоритму - 720 мс. Сумарний середній відносний результат 0.2.

Маємо що 1-й алгоритм сумарно працював менше часу, а 2-й в середньому менше відхилявся від оптимального за тестом.

Середній відносний результат може бути непоганою оцінкою роботи алгоритма за декількома тестами, хоча можливо можна обрати більш доцільну формулу для обчислення $\sigma_{\text{алг}}$.

Далі я буду користуватись формулою наданою в цьому підрозділі.

Додамо що в практичній частині для порівняння тести будуть проводитися на наступних текстах:

- Англійська література - «The Lord of the Rings: The Fellowship of the Ring»
- Українська література - «Лісова пісня»
- Випадкова послідовність геному.
- Json база даних.
- Велика веб сторінка (html).
- rand2 - випадковий рядок на алфавіті довжини 2

- rand32 - випадковий рядок на алфавіті довжини 32
- rand64 - випадковий рядок на алфавіті довжини 64
- Великий текст на мові програмування *c*.

3.3 Набір алгоритмів для порівняння

Для порівняння ефективності на практиці мною було обрано наступні алгоритми (табл 3):

	Повна назва	**Рік розробки**	**Клас алгоритму**
KR	Karp-Rabin	1987	comparison
KMP	Knuth-Morris-Pratt	1977	comparison
BNDMQ2	BNDM with q-grams	2009	bit-parallelism
BM	Boyer-Moore	1977	comparison
BXS	BNDM with Extended Shifts	2010	bit-parallelism
BOM	Backward-Oracle-Matching	1999	automata
Skip	Skip-Search	1998	comparison
Hash3	Wu-Manber algorithm	2007	comparison
FS	Fast-Search	2003	comparison
SSM	Simple String Matching	2015	hybrid
SBNDM	Simplified BNDM	2003	bit-parallelism
BSDM	Backward SNR DAWGM	2012	automata

Табл. 3: Інформація про обрані алгоритми пошуку рядка

При виборі набору алгоритмів орієнтувався на набір, що містиме деякі класичні алгоритми (**KR**,**KMP**,**BM**). Інші обирались серед більш сучасних, так щоб алгоритмів різних класів було представлено приблизно порівну.

4 Практична частина

4.1 Порівняння алгоритмів на випадкових патернах

В цьому підрозділі будемо порівнювати алгоритми на повністю випадкових патернах, заданого алфавіту для кожного тесту.

Середовище тестування:

Всі тестування проводились на персональному комп'ютері з наступними характеристиками:

- **Процесор:** Intel(R) Core(TM) i7-9750H CPU 2.60GHz
- **Оперативна пам'ять:** 16.0 GB
- **Жорсткий диск:** 512 GB SSD
- **Операційна система:** Windows 10 Home 64-bit
- **Графічний процесор:** NVIDIA GeForce GTX 1650 8GB

Проведемо тестування на англійському тексті («The Lord of the Rings: The Fellowship of the Ring», табл. 4), та випадкових патернах довжини m.

$$m = 2^k, (1 <= k <= 12)$$

Додамо що, результат вказаний у табл. 4 обраховувався не одним тестуванням, а для кожного m проводилась константна кількість (в моєму випадку 30) раундів тестування, і сумарний час їх виконання для кожного алгоритму було записано у таблицю.

m	KR	KMP	BNDMQ2	BM	BXS	BOM	Skip	Hash3	FS	SSM	SBNDM	BSDM
2	148.09	232.28	157.14	187.87	60.26	225.18	57.41	232.22	154.28	138.43	198.38	**49.66**
4	139.92	230.29	51.46	95.36	31.52	167.41	31.58	265.74	77.60	67.08	71.31	**29.69**
8	129.49	226.57	22.05	46.34	14.96	136.70	15.57	90.37	38.90	33.42	30.36	**13.64**
16	132.52	228.02	10.62	23.38	9.41	131.90	10.14	39.02	20.21	17.19	14.39	**9.03**
32	135.19	226.09	**5.97**	12.66	6.36	125.83	7.05	18.99	11.07	9.46	7.47	7.23
64	130.19	225.36	5.58	7.06	**4.72**	119.16	5.43	9.88	6.27	5.20	5.58	5.96
128	134.70	225.72	5.46	4.60	**3.64**	93.22	4.97	5.97	4.37	**3.64**	5.63	6.00
256	138.60	238.41	6.07	3.96	236.48	71.59	4.93	4.58	4.03	**2.95**	6.08	5.66
512	130.37	223.59	5.47	3.63	225.89	46.16	4.74	4.26	3.71	**2.73**	6.29	5.64
1024	134.62	227.50	5.93	3.69	226.83	41.15	5.86	4.01	4.09	**3.03**	6.33	6.09
2048	129.63	228.63	5.53	3.89	228.88	54.92	7.95	4.23	4.43	**3.61**	5.83	6.18
4096	135.46	229.16	5.76	5.36	229.06	104.23	14.62	**5.27**	5.99	5.51	6.95	6.77

Табл. 4: Результати роботи алгоритмів на англійському тексті (час у мілісекундах)

Кожен найкращий результат за заданим m, виділено у таблиці жирним шрифтом.

Проаналізуємо отримані результати. Бачимо, що SSM алгоритм отримує оптимальні результати на великих m

5 Висновки

Література

[1] Alsmadi, I., & Nuser, M. (2012). String matching evaluation methods for DNA comparison. International Journal of Advanced Science and Technology, 47(1), 13-32.

[2] J. H. Morris, Jr and V. R. Pratt. A linear pattern-matching algorithm. Report 40, University of California, Berkeley, 1970.

[3] Faro, S., & Lecroq, T., Borzi, S., Di Mauro, S.,& Maggio, A. (2016, August). The String Matching Algorithms Research Tool. In Stringology (pp. 99-111).

[4] Faro, S., & Lecroq, T. (2013). The exact online string matching problem: A review of the most recent results. ACM Computing Surveys (CSUR), 45(2), 1-42.

[5] R. S. Boyer and J. S. Moore. A fast string searching algorithm. Commun. ACM, 20(10):762–772, 1977.

[6] Faro, S., & Lecroq, T. (2010). The exact string matching problem:

a comprehensive experimental evaluation. arXiv preprint arXiv:1012.2547.