The Journal of Supercomputing (2024) 80:17741–17759
https://doi.org/10.1007/s11227-024-06115-8

Pattern matching algorithms in blockchain for network fees reduction

Robert Susik[1] **· Robert Nowotniak**[2]

Accepted: 28 March 2024 / Published online: 2 May 2024

Abstract
Blockchain received a vast amount of attention in recent years and is still growing. The second generation of blockchain, such as Ethereum, allows execution of almost any program in EVM, making it a global protocol for distributed applications. The code deployment and each operation performed in EVM cost the network fee called gas, whose price varies and can be significant. That is why code optimization and well-chosen algorithms are crucial in programming on the blockchain. This paper evaluates the gas usage of several exact pattern matching algorithms on the EVM. We also propose an efficient implementation of the algorithms in the Solidity/YUL language. We evaluate the gas fees of all the algorithms for different parameters (such as pattern length, alphabet size, and text size). We show a significant gas fee and execution time reduction with up to 22-fold lower gas usage and 55-fold speed-up compared to StringUtils (a popular Solidity string library).

Keywords Blockchain · Ethereum · Fee · Matching · Pattern · String

1 Introduction

Text pattern matching in blockchain is not a very popular topic. The reason can be found in the cost of running the blockchain code. Each operation generates a cost, so developers limit the functionality of smart contracts to a minimum. In this way, they move much of the business logic outside the blockchain. The costs, of course, justify this approach, but it also brings with it some problems and difficulties. This

✉ Robert Susik
rsusik@kis.p.lodz.pl

Robert Nowotniak
rnowotniak@metasolid.tech

[1] Institute of Applied Computer Science, Lodz University of Technology, Al. Politechniki 11, 90–924 Lodz, Poland

[2] MetaSolid.tech, Gdańsk, Poland

means that applications are divided into two parts, even though the application logic does not. This involves the obligation to maintain the application in two places, one of which (the one on the blockchain) is publicly available and versioned without the possibility of obliterating traces of code modification because nothing "gets lost" in the blockchain. However, the second part may be closed code. Such a division may be desirable, but in many cases, the only justification is the cost of maintaining the application.

We believe that reducing the costs of maintaining applications on the blockchain could eliminate these limitations and, at the same time, contribute to the development of applications operating on the blockchain and their business logic. The developing web3 trend may bring a gas cost reduction that would enable the creation of applications that run only on the blockchain without maintaining an off-chain server. In such a case, text algorithms (including text search) and other algorithms (e.g., artificial intelligence) could be used. In this article, we discuss the first topic, which, in our opinion, is interesting, but its application is currently quite limited. Nevertheless, we want to show how underdeveloped the libraries and algorithms in the blockchain environment are at the current stage and how much the costs of running applications can be reduced by implementing more efficient algorithms.

1.1 Background

Blockchain emerged as a peer-to-peer network with immutable transaction records on a shared public ledger designed for implementing transactions of electronic cash (cryptocurrency). Nakamoto introduced the first successful implementation in 2008 called Bitcoin [1]. Over the years, blockchain gained popularity [2] and became a promising technology that found application in many computer science fields. Several alternative blockchains were introduced (Namecoin,[1] Litecoin,[2] Peercoin,[3] etc.) before the second generation of blockchain was developed.

Ethereum [3], the first Blockchain 2.0, was introduced as a protocol for building decentralized applications running in the blockchain. In short, it is a distributed data storage plus smart contracts platform [4] that introduces an Ethereum Virtual Machine (EVM). One of the main advantages of EVM is the support of Turing-complete [5] programming language, which allows for writing decentralized applications based on smart contracts [6]. Ethereum has its own cryptocurrency called Ether, which is also used as a computational crypto-fuel to execute a code in EVM and pay transaction fees. For each transaction, the user needs to specify the upper bound of gas that can be consumed by the transaction. An advantage of such an approach is that it helps to avoid the situation where all the user's resources are wasted, for instance, an "infinite" loop. The mentioned code execution cost may differ depending on the number of operations performed in a transaction. It means that the infinite loop is not possible because, in the worst case, the EVM will stop processing the

[1] www.namecoin.org.
[2] litecoin.org.
[3] www.peercoin.net.

code by raising the "out of gas" error. A single computational step (which we can compare to a single CPU cycle) costs one unit of gas, and a single operation usually takes more than one step. For instance, an operation (ADD) that sums two 32-byte integer numbers costs three units of gas. On the contrary, there are a few operations that cost nothing, such as RETURN. Apart from the execution cost, each byte of the transaction data costs 5 units of gas.

Several languages are available for writing smart contracts, such as Solidity, YUL, Serpent, and Vyper. The former, Solidity, is the most popular [7] and recommended object-oriented programming language for Ethereum. YUL and Serpent are high-level assembly languages, and Vyper puts emphasis on simplicity and security (the syntax similar to Python, with inheritance removed). All of the mentioned languages are translated to EVM stack-based bytecode language that, once deployed in blockchain, can be executed by a transaction transferring Ether (the fuel) to the contract address.

Development of blockchain high-level programming languages opened new opportunities to create more complex smart contracts, which combined with user interfaces form applications called Dapps (Decentralized applications). It is aligned with the web3 concept where the applications are decentralized and always available. However, more complex apps consume more gas which causes higher costs.

1.2 Motivation

Blockchain finds a wide range of applications in areas such as healthcare [8, 9], voting [10], transportation [11], music industry [12], supply chains [13], reputation systems [14], document versioning [15], and decentralized finance [16]. The interest in Blockchain-based technologies is growing rapidly [7, 17]. Similarly to Web2.0, web3 Dapps can be reached with its alias name via DNS-like services such as ENS,[4] Unstoppable domains,[5] or Namecoin that are supported by web browsers (web browser extensions or dedicated web browsers to navigate and browse blockchain-based applications). This new type of application has web/mobile apps, backend (smart contract) data sources (Oracle contract), or data storage (i.e., IPFS). In a general case, it is possible to implement almost any application with the use of blockchain technology. However, there are technical (i.e., stack depth and size) and financial (gas is expensive compared to CPU time) limitations. While the first one may be solved with future EVM development, the gas fees seem to be a challenge [18–20]. Currently, there are approaches that significantly reduce the cost of running code in blockchain, such as Polygon, Solana, or Ethereum 2 (a new "consensus layer," which leverages *Proof-Of-Stake* algorithm [21] in place of *Proof-Of-Work* [1]).

The transformation process of traditional application to a Dapp is not well defined and is a subject of study [7]. Along with this process, there is an obvious need for algorithms and libraries on EVM. In [22], the authors performed an

[4] ens.domains.

[5] unstoppabledomains.com.

interesting analysis of computational costs using gas consumption as the metric. The gas price prediction [19, 20, 23] and transaction fee optimization [24–26] are an active subject of study.

As the Dapp gains popularity, it is likely that more and more complex applications are about to be implemented. In the case of blockchain-based applications, the algorithms should be carefully considered as a wrong decision may cause unnecessary costs making the software expensive or even useless. In software development, pattern matching algorithms are often used for data validation, spell checking, spam filtering, intrusion detection, bioinformatics, and more.

This paper proposes an efficient implementation of several exact pattern matching algorithms in the Solidity/YUL language. Over the years, tens of exact string algorithms were invented, most of which are modifications of the older ones. We consider six exact string matching algorithms, including the Naive approach, Knuth–Morris–Pratt, which is one of the first algorithms to solve the problem in linear time, Boyer–Moore–Horspool that represents a significant class of string matching algorithms based on a bad-character rule that compares the pattern and text window backward (from the end of the pattern), Rabin–Karp which is an approach that efficiently uses hashing (rolling hash) to filter out the positions that do not match the pattern, Shift-Or, which simulates Nondeterministic Finite Automata with the bitwise techniques, and Backward Nondeterministic Dawg Matching algorithm that does a bit-parallel simulation of the suffix automaton (see Sect. 2.2 for more details). We believe that such a set represents (and is fundamental to) most of the online string matching algorithms and can be used for further research in this field. We evaluate the algorithms' gas fee and execution time for different parameters (such as pattern length, alphabet size, and text size). We show that some of those algorithms significantly reduce the gas fee and execution time compared to the existing Solidity library.

The following contributions of this work can be enumerated: (i) We adapt and implement six exact string matching algorithms for EVM. (ii) We present the performance of all implemented algorithms in the Ethereum blockchain environment. (iii) We show the gas fee and execution time reduction comparing to popular Solidity library.

Section 2 defines the problem of exact pattern matching and describes all the implemented algorithms. Section 3 presents the results of performed experiments in terms of gas usage. Finally, Sect. 5 concludes the results and suggests future work.

2 Proposed approach

2.1 Problem

Exact string matching is one of the most explored problems in computer science. The problem can be stated as follows: For a given text $T[0 \dots n-1]$, and a pattern

$P[0 \dots m-1]$, $m \leq n$, both over a common alphabet $\sum$ of size σ, report all occurrences of P in T, such that $P[0 \dots m-1] = T[i \dots i+m-1]$, where $i \leq n-m$.

2.2 Algorithms

The string matching algorithms constitute an essential component in many software applications [27]. Over the years, tens of algorithms have been invented, most of which are modifications of the older ones [28, 29]. We adapted and implemented several classic exact text matching algorithms. The Solidity language and EVM specification (and limitation) implied changes to original algorithms implementation. We adapted and optimized the algorithms to take advantage of 32-byte word and reduce the number of expensive instructions (i.e., bitwise shifts) as much as possible. Those changes are rather technical tricks that do not change the algorithm's logic. We discussed the changes most impacting the performance in Sect. 4. Many instructions (i.e., SIMD,[6] AVX) that are available in modern CPUs are not available in EVM yet, and thus, some improvements cannot be implemented.

2.2.1 Naive algorithm

The Naive (also called Brute-Force) approach to this problem is to scan text T using a window of size m. The window starts at position 0 and moves toward the end of the string (say from left to right). At each step, the content of the window is compared with the pattern character by character. If all m characters match, the position is reported. If there is a mismatch, the window is shifted by one position to the right. The complexity is $O(nm)$ in the worst case, and if $\sigma \geq 2$, then the average complexity equals $O(n)$, which was experimentally supported in [30].

Algorithm 1 Naive

```
1: function NAIVE(P[0...m−1], T[0...n−1])
2:     for i ← 0...n−m do
3:         if T[i...i+m−1] = P[0...m−1] then
4:             report i
```

2.2.2 Knuth–Morris–Pratt

One of the first solutions that reduce the number of character comparisons to find a pattern in the text is Knuth–Morris–Pratt (KMP) algorithm [31]. This approach assumes that the pattern P contains information about how many text T characters can be skipped in the next step. KMP reads the pattern and builds a lookup table

[6] A proposal (EIP-616) to provide SIMD in EVM was created by Greg Colvin in 2017 but has not been implemented yet.

$N[0 \dots m-1]$, which contains information on how many characters can be skipped if a mismatch occurs. The algorithm sequentially compares characters between pattern P and text T from left to right. Once all m characters are matched, the position is reported. If a mismatch occurs, the algorithm reads how many characters can be skipped from table N. KMP compares between n and $2n-1$ characters, the search complexity is $O(n)$, and the N table is done in $O(m)$.

Algorithm 2 Knuth-Morris-Pratt

```
1:  function KMP(P[0...m−1], T[0...n−1])
2:      i ← 0; j ← −1; N[0] ← −1                          ▷ Preprocessing phase
3:      while i < m do
4:          if j > −1 and P[i] ≠ P[j] then
5:              j ← N[j]
6:              continue
7:          i ← i + 1; j ← j + 1
8:          if P[i] = P[j] then
9:              N[i] ← N[j]
10:         else
11:             N[i] ← j
12:
13:     i ← 0; j ← 0                                      ▷ Search phase
14:     while i < n do
15:         if j > 0 and P[j] ≠ T[i] then
16:             j ← N[j]
17:             continue
18:         i ← i + 1; j ← j + 1
19:         if j ≥ m then
20:             report i − j
21:             j ← N[j]
```

2.2.3 Boyer–Moore–Horspool

Boyer–Moore–Horspool (BMH) algorithm [32], presented in Algorithm 3, is a simplified variant of Boyer–Moore (BM) [33]. The algorithm compares characters from right to left, and if a mismatch occurs, then the window is shifted according to so-called *bad-character heuristic* [32].

This rule says that if the last symbol in the text frame ($c = T[i+m-1] \neq P[m-1]$) does not occur in the pattern, the text frame can be moved by m positions, on the other hand, if such a symbol appears in the pattern at position $pos = \text{last}(c, P[0 \dots a-1])$, where $\text{last}(c, s)$ is the function that returns the last position of character c in string s, the text frame can be moved by $m - 1 - pos$. The algorithm uses these observations to filter out (skip) text regions where the pattern cannot occur. The function that implements this rule takes a parameter as a character read from the text and returns the value indicating how many characters can be skipped (up to a maximum of m).

Searching takes $O(nm)$ time in the worst case, $O(n \log_\sigma(m)/m)$ in the average case, and $O(n/m)$ in the best case.

Algorithm 3 Boyer-Moore-Horspool

```
1: function BMH(P[0...m−1], T[0...n−1])
2:     Initialize J[0...255] with m                          ▷ Preprocessing phase
3:     for i ← 0 ... m−2 do
4:         J[P[i]] ← m−1−i
5:     i ← 0
6:     while i ≤ n−m do                                      ▷ Search phase
7:         c ← T[i+m−1]
8:         if P[m−1] = c and P[0 ... m-1] = T[i ... i + m - 1] then
9:             report i
10:        else
11:            i ← i + J[c]
```

2.2.4 Rabin–Karp

The Rabin–Karp (RK) algorithm [34] is the first one that uses a *rolling hash* for text search purposes. The algorithm (Algorithm 7) calculates a hash for the pattern $P[0 \ldots m-1]$ and text window $T[0 \ldots m-1]$, then moves the window toward the end of the text. At each step, i, the hash is recalculated by adding the character that enters the window $T[i+m]$ and removing the one that moves outside the window $T[i]$.

This solution's critical element is the hash function's efficiency. In the context of this approach, a good hash function ensures that each text character is processed only once. Such a function is often referred to as a rolling hash. This term implies that to compute the value of $h(T[i+1 \ldots i+m])$, we can utilize a previously calculated value of $h(T[i \ldots i+m-1])$. As a parameter, we provide the character that moves outside the text frame in the next step ($T[i]$), along with the character that appears as the last element within the text frame ($T[i+m]$).

The average complexity of this algorithm is $O(n+m)$, and $O(nm)$ in the worst case.

Algorithm 4 Rabin-Karp

```
1: function RK(P[0...m−1], T[0...n−1])
2:     h_p ← h(P[0...m−1])
3:     h_t ← h(T[0...m−1])
4:     for i ← 0...n−m do
5:         if h_t = h_p then
6:             report i
7:         h_t ← update(h_t, T[i], T[i+m])          ▷ T[i] removed, T[i+m] inserted
```

2.2.5 Shift-Or

Shift-Or (SO) algorithm [35] simulates Nondeterministic Finite Automata (NFA) [36]. The authors employed an approach where they process the text by comparing characters from left to right. The algorithm is divided into two stages: preprocessing and search. Both stages are presented in Algorithm 5. In the preprocessing stage, for each alphabet symbol, a bitwise mask $B[x]$ is created (where x is the alphabet symbol). This mask contains a value of 0 at the i-th bit if $P[i] = x$. The bitwise mask represents a transition table, where a value of 0 for $B[x]$ at the i-th bit indicates a transition from state i to state $i + 1$ for symbol x. Subsequently, during the search process, the algorithm utilizes a state vector D (a bit vector) initially filled with 1 on each bit. Successively reading the text's characters, the algorithm performs a left bitwise shift operation on the state vector D and a bitwise OR operation with the bitwise mask of the current symbol. These operations determine whether a transition (or lack thereof) occurs to the next state (8-th line). The automaton reports the occurrence of the pattern in the text when a value of 0 appears on the $(m - 1)$-th bit (9-th line), which corresponds to the transition to the final state.

Such a bitwise technique is efficient if $m \leq w$, where w is the machine word size. In EVM, the machine word size is 256-bit, so the algorithm performs best if the pattern has at most 256 characters. The algorithm can find a larger pattern, but in that case, it searches for the prefix (of size w) and verifies the reported positions. One workaround for this limitation was presented in [37] where authors used End-Tagged Dense Code [38] to reduce the pattern size. The algorithm also found application in circular pattern matching [39]. The complexity of this algorithm is independent of the pattern length and equals $O(n\lceil m/w \rceil)$, which gives $O(n)$ for $m = O(w)$.

Algorithm 5 Shift-Or

```
1:  function SO(P[0...m−1], T[0...n−1])
2:      for i ← 0...σ−1 do                          ▷ Preprocessing phase
3:          B[i] ← ∼0
4:      for i ← 0...m−1 do
5:          B[P[i]] ← B[P[i]] & ∼(1 ≪ i)
6:      D ← ∼0; mm ← 1 ≪ (m−1); i ← 0
7:      while i < n do                               ▷ Search phase
8:          D ← (D ≪ 1) | B[T[i]]
9:          if (D & mm) ≠ mm then
10:             report i − m + 1
11:         i ← i + 1
```

2.2.6 Backward nondeterministic dawg matching

Backward Nondeterministic Dawg Matching (BNDM) [40] is a Directed Acyclic Word Graph simulation implemented with bit-parallel techniques presented. The algorithm presented in Algorithm 6, like BMH, compares the characters from the

last character in the window, and if the character does not occur in the pattern, the window is shifted by $m - x$ (x is the length of the suffix that matches) characters forward.

The search process is divided into two stages: preprocessing and searching. In the first stage, the algorithm creates a table B of size σ, where each element represents a bitwise mask with pattern symbols' positions. If symbol c does not occur in the pattern, the symbol's bitwise mask ($B[c]$) has zeros on all bits. Conversely, if symbol c occurs in the pattern at position i, the i-th bit of the bitwise mask is set. Next, the algorithm moves to the search stage. It moves a text frame (of size m) from left to right, reading the characters from the frame, starting at the last character (from the right side, i.e., $m - 1 \ldots 0$). If the character of the frame and pattern match, the least significant bit of the $B[T[i + m - 1]]$ is set to 1; otherwise, it is set to 0. The position of the symbol occurrence is stored in variable d using the operation $d \leftarrow d\&B[T[i + m - 1]]$. This information represents the current state. Transition involves left-shifting d variable ($d \leftarrow d \ll 1$). This process continues until $d\&B[c]$ returns 0 or until the $(m - 1)$-th bit (counting from the least significant) of variable d is set to 1. If part of the pattern's characters has been matched, the frame is shifted forward by $m - x$ (where x is the length of the matched suffix).

Like SO, the max pattern length depends on the machine word size. The complexity is $P(n/m)$ in the best case and $O(nm)$ in the worst case and $O(n \log_\sigma m/m)$.

Algorithm 6 Backward Nondeterministic DAWG Matching

```
1:  function BNDM(P[0...m−1], T[0...n−1])
2:      B[0...σ−1] ← 0                                  ▷ Preprocessing phase
3:      for i ← m−1...0 do
4:          B[P[i]] ← B[P[i]] | (1 ≪ (m−1−i))
5:      j ← 0
6:      while j ≤ n−m do                                ▷ Search phase
7:          i ← m−1
8:          d ← ∼0
9:          while i ≥ 0 and d ≠ 0 do
10:             d ← d & B[T[j+i]]
11:             if d ≠ 0 and i ≤ 0 then
12:                 report j
13:             d ← d ≪ 1
14:             i ← i−1
15:         j ← j+(i+2)
```

2.2.7 StringUtils

StringUtils [41] is a popular Solidity library for string operations that most developers would copy into their programs and deploy along with their smart contracts [42]. There are several functions supported, but we are primarily interested in the "find" operation, which searches the first occurrence of pattern P in text T and returns the "slice" (a data structure representing the substring of the text).

The algorithm works as follows. If the pattern fits into a 256-bit (32 bytes) machine word, the first mode of the algorithm is activated, which reads subsequent chunks (32 bytes) of text, filters them using a mask, and compares them with the pattern. The algorithm then moves forward one character. This mode resembles the Naive algorithm. When $m >= 32$, the algorithm works similarly to the Rabin–Karp algorithm. First, it calculates the hash for the pattern, and then, moving the text frame calculates the hash of the text frame at each position and compares it with the pattern's hash. Here, the keccak256 function is used as a hash function. It is not a rolling hash, which means that the hash is calculated for all characters of the text frame.

Algorithm 7 StringUtils

```
1:  function StringUtils(P[0...m−1], T[0...n−1])
2:      if m ≤ 32 then
3:          for i ← 0...n−m do
4:              if P[0...m−1] = T[i...i+m−1] then
5:                  report i
6:              i ← i+1
7:      else
8:          h_p ← keccak256(P[0...m−1])
9:          h_t ← keccak256(T[0...m−1])
10:         for i ← 0...n−m do
11:             if h_t = h_p then
12:                 report i
13:             h_t ← keccak256(T[i...i+m−1])
```

3 Experimental results

In order to evaluate the performance of the algorithms, we performed various experiments. We tested the algorithms in the Ethereum network using Ganache v6.12.2 (ganache-core: 2.13.2)[7] (a personal blockchain for development). The algorithms were implemented in the Solidity language interleaved with inline assembly statements written in YUL. All source codes were compiled with Solc v0.8.11 compiler with optimizer enabled for 200 runs and shared publicly on Github.[8] The smart contract was deployed on the Rinkeby network.[9] The experiments were executed on a machine equipped with Intel(R) Core(TM) i5-3570 CPU 3.4 GHz, (256 KB L1, 1 MB L2, and 6 MB L3 memory), 16 GB of DDR3 1333 MHz RAM, and running under Fedora 28 64-bit OS. As a competing algorithm for comparison, we took a widely used and popular StringUtils[10] library. We are unaware of any other fast implementation of exact string matching algorithms on the blockchain than

[7] trufflesuite.com/ganache.
[8] github.com/rsusik/pattern-matching-in-blockchain.
[9] 0x9Fb22d8d82FcF1c5321D5acf75eE917CF936E257.
[10] github.com/Arachnid/solidity-stringutils.

Table 1 Input parameters of the implemented algorithms

Input parameter	Possible values
Datasets	`dna`, `english`, `proteins`, `sources`
Pattern length	4, 8, 12, 16, 24, 32, 64, 128, 256, 512
Text length	1 KiB, 16 KiB, 128 KiB

the functions available in StringUtils. The tests were performed on datasets from Pizza and Chilli corpus.[11]

Algorithms were tested using multiple pattern sizes and text sizes (substrings of mentioned datasets). All the parameters are shown in Table 1. We generated 11 patterns for each test case and presented the median value (gas or execution time) of searching them. In the first set of analyses, we investigated the impact of the alphabet, text size, and pattern size on gas usage. We noticed a considerable difference in gas usage for different m.

All tests have been performed on the same machine and environment. We used the same libraries and compiler versions for all codes. The EVM is deterministic, meaning that the transaction takes the same amount of gas for the same input data, so the results are repeatable. Gas usage is public information that is available to everyone in the blockchain. To read the gas usage of executed transactions, we used the brownie API. Despite the gas usage, we also measured the execution time of each algorithm. The execution time may vary slightly depending on the machine and the environment.

In Fig. 1, we can clearly see that the StringUtils function increases rapidly once the pattern size exceeds 32 characters. It can be easily explained, the StringUtils highly depends on the fact that the machine word size in EVM is 32 bytes. For patterns with at most 32 characters ($m \leq 32$), the algorithm packs all the pattern characters into a 32-byte variable, compares it against the masked text window and finally, shifts the text window by one position. If the pattern is longer than 32 characters, the algorithm calculates the hash (`keccak256`) of the pattern, compares it against the hash of the text window and then, shifts the text window by one. The cost of `keccak256` depends on the length of the input, which is why the cost grows if m increases. On the other hand, the Boyer–Moore–Horspool algorithm takes advantage of longer patterns as it allows to make larger jumps (if the first character does not exist in the pattern, the algorithm skips m positions of the text). We find that the BHM wins in almost all cases, and only BNDM is comparable. The most striking fact to emerge from these results is that the BMH reduces gas usage by up to 22-fold compared to StringUtils. Interesting is the fact that the StringUtils has an even worse result than the Naive approach for `sources` and $m = 512$.

Table 2 shows gas usage and its price (fee) of searching $m = 512$ pattern in 128 KiB text. In this case, if we assume the current[12] gas price (about 25 Gwei) and

[11] pizzachili.dcc.uchile.cl.

[12] As per 2022.07.10.

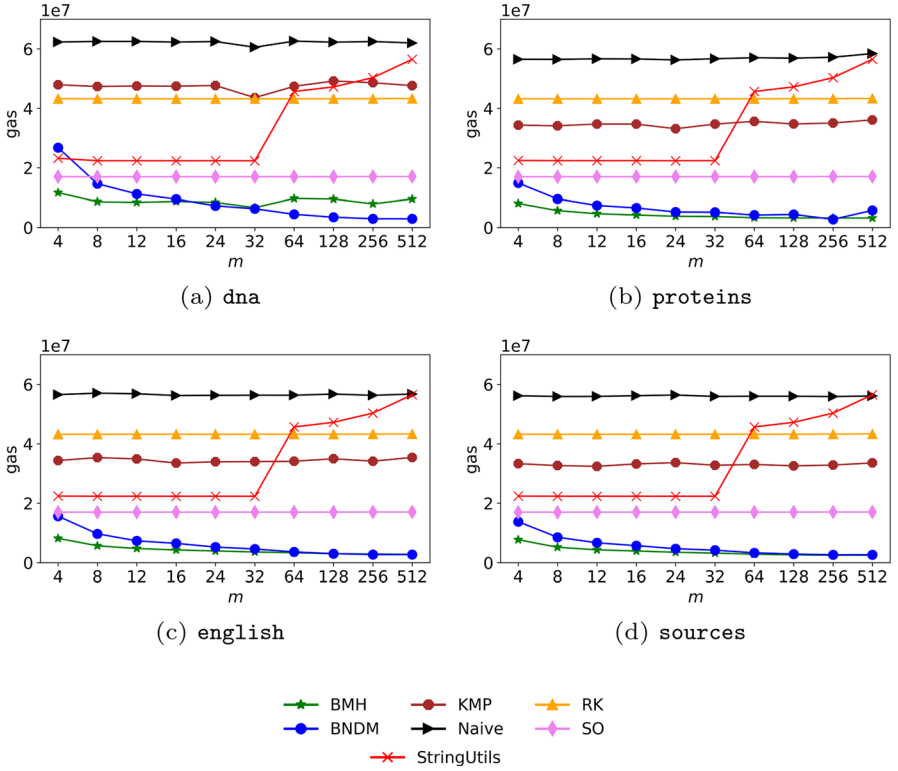

Fig. 1 Gas usage in function of pattern size for $n = 128$ KiB

Table 2 Gas usage (in millions) and fee of searching long pattern ($m = 512$) in 128 KiB text (the best results are in bold)

Text	dna		English		Proteins		Sources	
	Gas	Fee	Gas	Fee	Gas	Fee	Gas	Fee
Algorithm								
BMH	9.55	$298.34	2.83	$88.43	**3.21**	**$100.29**	**2.53**	**$78.92**
BNDM	**2.96**	**$92.49**	**2.75**	**$85.99**	5.78	$180.47	2.68	$83.65
KMP	47.65	$1488.91	35.42	$1106.95	36.12	$1128.80	33.59	$1049.69
Naive	61.99	$1937.23	56.74	$1773.07	58.42	$1825.76	56.07	$1752.33
RK	43.33	$1353.93	43.33	$1353.93	43.34	$1354.31	43.33	$1353.93
SO	17.06	$533.21	17.06	$533.21	17.07	$533.35	17.06	$533.21
StringUtils	56.50	$1765.74	56.50	$1765.74	56.51	$1765.93	56.50	$1765.74

USD/ETH exchange rate (1250 USD), the approximate cost of searching a pattern of $m = 512$ characters in `sources` dataset using StringUtils is about $1766 whereas the same using BMH is about $79. The BMH wins for `proteins` and `sources`,

Table 3 Gas usage per text character of searching short ($m = 16$) pattern

Text	Algorithm n (KiB)	BMH	BNDM	KMP	Naive	RK	SO	StringUtils
dna	1	111.89	124.19	358.08	478.56	357.66	176.04	195.31
	16	67.97	74.18	366.65	473.92	331.00	132.12	171.79
	128	66.84	72.44	362.15	475.25	329.66	129.77	170.64
English	1	77.18	95.52	290.64	450.87	357.66	176.04	195.31
	16	34.57	52.27	261.74	431.39	331.00	132.12	171.79
	128	32.70	49.62	255.79	429.09	329.66	129.77	170.64
Proteins	1	77.70	96.22	297.96	452.59	357.66	176.04	195.31
	16	34.30	49.81	259.82	430.62	331.00	132.12	171.79
	128	32.12	50.30	264.74	432.02	329.68	129.78	170.91
Sources	1	86.59	105.59	289.87	456.54	357.66	176.04	195.31
	16	32.03	48.07	244.69	425.78	331.00	132.12	171.79
	128	30.01	43.70	253.65	428.74	329.66	129.77	170.64

but in case of `dna` and `english` the BNDM dominates. However, only in the case of `dna` the difference between BMH and BNDM is notable.

We can notice that the time and gas consumption of both BMH and BNDM are decreasing when m is growing. The reason behind this is that both algorithms take advantage of long patterns as the potential text frame movement can be larger. The maximum number of skipped positions in those algorithms equals m.

Table 3 presents gas usage per text character. All the algorithms, despite Naive, are more expensive for very short texts (1 KiB) than for longer ones (16 KiB and 128 KiB). The gas usage per character falls with the text size increase. The difference between 1 and 16 KiB is more considerable than between 16 and 128 KiB. We see, in this case, the impact of the function's initial steps and the preprocessing phase on transaction cost. Both take the same amount of resources if the pattern size is fixed. It is also the reason why it only slightly affects the Naive algorithm, which does not have a preprocessing phase.

In addition to gas usage, we measured searching time. Figure 2 presents median time (in seconds) of searching patterns in 128 KiB text. As expected, the time and gas are correlated. However, we noticed some discrepancies between them. For instance, in the case of `sources` dataset, the StringUtils needs 22 times more gas than BMH to find a pattern of $m = 512$ characters, whereas it needs 55-fold more time, which means the result in terms of time is 150% higher than for the gas usage. Nevertheless, the difference is much smaller for very short patterns ($m = 4$, and the same other parameters). The StringUtils time and the gas usage are 3.22 and 2.90 higher than BMH (respectively), resulting in about 11% more in time than gas consumption.

In our experiment, we measured the gas usage and execution time of the algorithms. The first one is deterministic, which means we always get the same result for a particular input, while the second may be different as it depends on other factors (such as the machine load, hardware specification, or I/O access time).

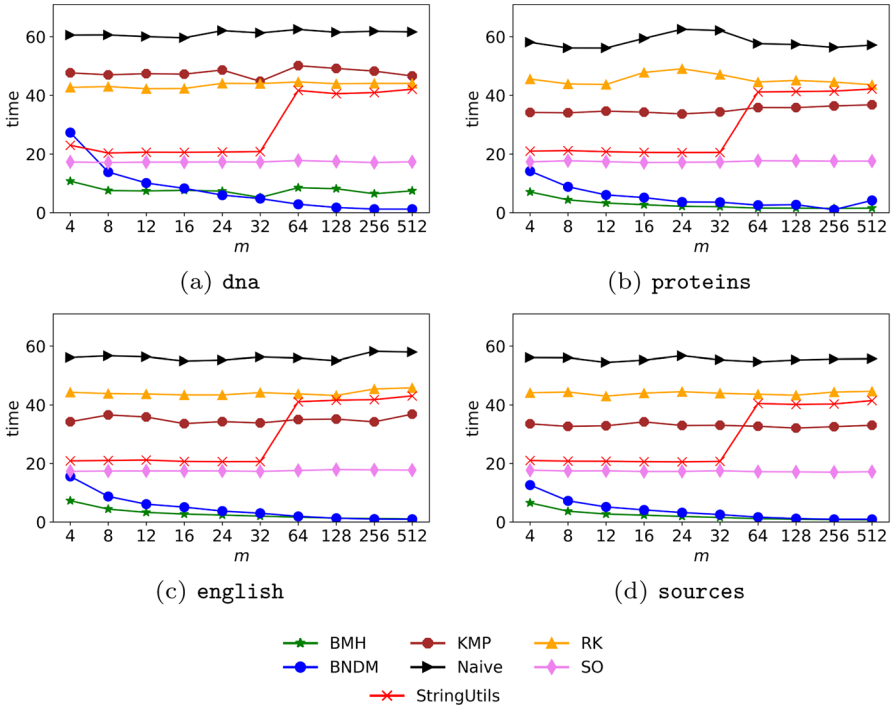

Fig. 2 Time (in seconds) in function of pattern size for $n = 128$ KiB

Table 4 Results of the paired t test for the execution time of the algorithms

Algorithm	M	SD	t	p value	Effect size
BNDM	2.31	3.78	24.58	2.05×10^{-110}	0.87
BMH	1.70	2.28	26.70	3.40×10^{-126}	0.95
SO	6.86	7.56	20.29	3.61×10^{-80}	0.39
KMP	14.57	16.97	− 15.87	1.00	− 0.21
NAIVE	22.10	25.57	− 27.87	1.00	− 0.53
RK	17.01	19.50	− 23.37	1.00	− 0.34

Thus we performed the statistical tests only for the execution time. We defined the alternative hypothesis as follows, the mean of the distribution underlying the first sample (StrUtil) is less than the mean of the distribution underlying the second sample (one of other algorithms). All the algorithms were executed to search for the same patterns, so the sample was the same for all algorithms. It means we have to peform a paired t test to compare the means of the search time of the algorithms. We performed the paired t-test between StrUtil and all other algorithms, which is presented in Table 4.

It can be seen that for BNDM ($M = 2.31$, SD = 3.78), BMH ($M = 1.70$, SD = 2.28), SO ($M = 6.86$, SD = 7.56), the p value is near zero $t(1319) = 24.58$,

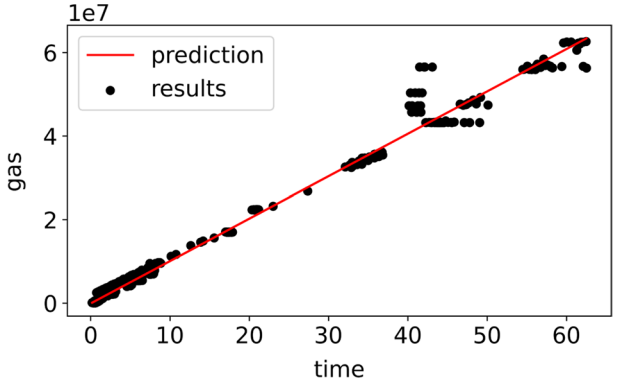

Fig. 3 Gas usage in function of execution time (in seconds)

$p < 0.001$ (2.05×10^{-110}), $t(1319) = 26.70$, $p < 0.001$ (3.40×10^{-126}), $t(1319) = 20.29$, $p < 0.001$ (3.61×10^{-80}), respectively, which means we can reject the null hypothesis and accept the alternative hypothesis. The test indicates that the results are statistically significant in terms of searching time. Thus, the BNDM, BMH, and SO algorithms are statistically significantly faster than the StrUtil algorithm. Additionally, we calculated the effect size (Cohen's d) for the above algorithms, which is equal to $d = 0.87$, $d = 0.95$, and $d = 0.39$, respectively. It means that the effect size is large for BNDM and BMH, and medium for SO. On the other hand, for NAIVE ($M = 22.10$, SD $= 25.57$), KMP ($M = 14.57$, SD $= 16.97$), RK ($M = 17.01$, SD $= 19.50$), the p-value is equal to 1 $t(1319) = -27.87$, $p = 1.0$, $t(1319) = -15.87$, $p = 1.0$, $t(1319) = -23.37$, $p = 1.0$, respectively, which means we cannot reject the null hypothesis and accept the alternative hypothesis. Thus, the StrUtil algorithm is statistically significantly faster than the NAIVE, KMP, and RK algorithms. It means the statistical tests confirm the results obtained in the practical experiment.

Finally, we accumulated all the results and displayed them in a single chart. Figure 3 presents the gas usage in the function of the execution time along with the trend line. We can clearly see that the execution time and the gas usage are correlated. The coefficients of linear regression allow estimating the cost of code execution approximately. We found that one second of code execution (assuming our environment specification) would cost about $31.71.

4 Discussion

Given the limitations of the Solidity language, such as the inability to directly reference memory, we often resort to using assembly inserts written in YUL in various parts of our code. One crucial optimization technique we employ involves efficiently reading consecutive characters. For instance, when reading text character by character in Solidity, each subsequent read of $t[i]$ (i.e., the i-th character from text t) is equivalent to reading 32 bytes, which is the size of a machine word (256 bits). However, this method proves inefficient as we use only one

byte (character size). In practice, we can read 32 bytes only once and process 32 characters (subsequent bytes), saving 31 memory reads. This can be achieved using the YUL language and several bitwise operations. It means we can read the 32-byte chunk by calling the `mload` operation, and then, to read the i-th byte, we perform a bit shift to the right by $248 - (i * 8)$ on the chunk and then apply the 0xFF mask. Additionally, we employ the loop unrolling technique to reduce the number of operations, which significantly reduces gas usage. Verifying found positions is crucial in some text matching algorithms (such as RK). To optimize this process, we first count the hashes from the pattern before starting the search. Then, once we find the position in the text, we compare the hash of the text frame with that of the pattern. While this is a standard practice for comparing strings in Solidity, it involves a performance compromise. Though theoretically, we could compare entire blocks of text using the `xor` function, which would require multiple iterations and executions of the function, incurring additional costs. Moreover, the gas cost of the `keccak256` function, which we use for hashing, increases with the input size. Hence, despite the initial overhead of calculating the hash for the pattern, it proves more efficient in the long run, especially when there are many verifications. Otherwise, if we compare the strings (e.g., using the `xor` operation), we would need to iteratively read subsequent chunks of both the pattern and text and compare them successively. Additionally, the last 32-byte chunk must be masked, requiring extra operations. Despite the higher gas cost associated with hash comparison, it remains cheaper than comparing the pattern and text frame, particularly for long patterns. This is because we calculate the hash for the pattern only once and reuse it multiple times. However, if the operation is performed only once, i.e., we only want to compare short strings a and b ($a = b$), it might be cheaper to do it using bitwise operations instead of comparing hashes. Unfortunately, the EVM architecture limits the stack size and depth, posing challenges for complex contract code. In such cases, it is necessary to determine which algorithms stay within the stack depth limit experimentally. While bytecode optimizers might help optimize the algorithms for more complex contracts, our experiments did not explore this aspect. Nonetheless, it is possible to bypass this limitation by saving the result of nested functions in variables and then calling another function with the value of the temporary variable. However, this incurs additional memory costs per variable. Fortunately, the EVM's determinism ensures consistent behavior across executions.

5 Conclusion

In this work, we adapted exact pattern matching algorithms to EVM architecture, implemented the algorithms in Solidity language combined with YUL assembly, and performed extensive tests using the Ethereum blockchain. We empirically proved that gas usage could be significantly reduced in all the test cases. The experiments confirmed the technical (smaller computational time) and financial (smaller costs) advantages of the proposed approach. We demonstrated that the cost of searching

patterns could be reduced by up to 22 times ($78.92 vs $1765.74) and the execution time by up to 55 times (41.47s. vs 0.75s.) when compared to StringUtils.

In our view, these results constitute a good initial step toward implementing these algorithms as a Solidity pattern matching library. The EIP-616 proposal to provide SIMD operations may also open further improvements to the proposed solution.

Acknowledgements We thank Sz. Grabowski (*Lodz University of Technology*) for helpful discussions and valuable suggestions.

Author contributions R.S. and R.N. wrote the manuscript. R.S. implemented the solution. R.N. tested the solution. All authors reviewed the manuscript.

Funding Not applicable.

Data availability All data and materials are publicly available at github.com/rsusik/pattern-matching-in-blockchain.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

Ethical approval Not applicable.

References

1. Nakamoto S (2008) Bitcoin: a peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf. Accessed 15 Apr 2022
2. Gad AG, Mosa DT, Abualigah L, Abohany AA (2022) Emerging trends in blockchain technology and applications: a review and outlook. J King Saud Univ-Comput Inf Sci 34:6719–6742
3. Buterin V (2013) Ethereum white paper: a next generation smart contract & decentralized application platform. https://ethereum.org/en/whitepaper. Accessed 15 Apr 2022
4. Guo H, Yu X (2022) A survey on blockchain technology and its security. Blockchain Res Appl 3:100067
5. Turing AM (1936) On computable numbers, with an application to the entscheidungsproblem. J Math 58(345–363):5
6. Szabo N (1996) Smart contracts: building blocks for digital markets. EXTROPY J Transhumanist Thought (16) 18(2):28
7. Wu K (2019) An empirical study of blockchain-based decentralized applications. arXiv preprint arXiv:1902.04969
8. Kumar A, Krishnamurthi R, Nayyar A, Sharma K, Grover V, Hossain E (2020) A novel smart healthcare design, simulation, and implementation using healthcare 4.0 processes. IEEE Access 8:118433–118471
9. Adere EM (2022) Blockchain in healthcare and IoT: a systematic literature review. Array 14:100139

10. Sober M, Scaffino G, Spanring C, Schulte S (2021) A voting-based blockchain interoperability oracle. In: 2021 IEEE International Conference on Blockchain (Blockchain). IEEE, pp 160–169
11. Mollah MB, Zhao J, Niyato D, Guan YL, Yuen C, Sun S, Lam K-Y, Koh LH (2020) Blockchain for the internet of vehicles towards intelligent transportation systems: a survey. IEEE Internet Things J 8(6):4157–4185
12. Li Y, Wei J, Yuan J, Xu Q, He C (2021) A decentralized music copyright operation management system based on blockchain technology. Procedia Comput Sci 187:458–463
13. Zhang Y, Chen L, Battino M, Farag MA, Xiao J, Simal-Gandara J, Gao H, Jiang W (2022) Blockchain: an emerging novel technology to upgrade the current fresh fruit supply chain. Trends Food Sci Technol 124:1–12
14. Almasoud AS, Hussain FK, Hussain OK (2020) Smart contracts for blockchain-based reputation systems: a systematic literature review. J Netw Comput Appl 170:102814
15. Nizamuddin N, Salah K, Azad MA, Arshad J, Rehman M (2019) Decentralized document version control using ethereum blockchain and ipfs. Comput Electr Eng 76:183–197
16. Amler H, Eckey L, Faust S, Kaiser M, Sandner P, Schlosser B (2021) Defi-ning defi: challenges & pathway. In: 2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS). IEEE, pp 181–184
17. Belchior R, Vasconcelos A, Guerreiro S, Correia M (2021) A survey on blockchain interoperability: past, present, and future trends. ACM Comput Surv (CSUR) 54(8):1–41
18. Marchesi L, Marchesi M, Destefanis G, Barabino G, Tigano D (2020) Design patterns for gas optimization in ethereum. In: 2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE). IEEE, pp 9–15
19. Mars R, Abid A, Cheikhrouhou S, Kallel S (2021) A machine learning approach for gas price prediction in ethereum blockchain. In: 2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC). IEEE, pp 156–165
20. Kim H-M, Bock G-W, Lee G (2021) Predicting ethereum prices with machine learning based on blockchain information. Expert Syst Appl 184:115480
21. King S, Nadal S (2012) Ppcoin: peer-to-peer crypto-currency with proof-of-stake. Self-published paper, August 19(1)
22. Jabbar A, Dani S (2020) Investigating the link between transaction and computational costs in a blockchain environment. Int J Prod Res 58(11):3423–3436
23. Pierro GA, Rocha H, Ducasse S, Marchesi M, Tonelli R (2022) A user-oriented model for oracles' gas price prediction. Future Gener Comput Syst 128:142–157
24. Li C, Nie S, Cao Y, Yu Y, Hu Z (2020) Trace-based dynamic gas estimation of loops in smart contracts. IEEE Open J Comput Soc 1:295–306
25. Di Sorbo A, Laudanna S, Vacca A, Visaggio CA, Canfora G (2022) Profiling gas consumption in solidity smart contracts. J Syst Softw 186:111193
26. Khan MMA, Sarwar HMA, Awais M (2022) Gas consumption analysis of ethereum blockchain transactions. Concurr Comput Pract Exp 34(4):6679
27. Charras C, Lecroq T (2004) Handbook of exact string matching algorithms. King's College, London
28. Grabowski S (2009) New algorithms for exact and approximate text matching. Technical University of Lodz Publisher
29. Faro S, Lecroq T (2013) The exact online string matching problem: a review of the most recent results. ACM Comput Surv (CSUR). https://doi.org/10.1145/2431211.2431212
30. Hume A, Sunday D (1991) Fast string searching. Softw Pract Exp 21(11):1221–1248
31. Knuth DE, Morris JH, Pratt VR (1977) Fast pattern matching in strings. SIAM J Comput 6(1):323–350
32. Horspool RN (1980) Practical fast searching in strings. Softw Pract Exp 10(6):501–506
33. Boyer RS, Moore JS (1977) A fast string searching algorithm. Commun ACM 20(10):762–772
34. Karp RM, Rabin MO (1987) Efficient randomized pattern-matching algorithms. IBM J Res Dev 31(2):249–260
35. Baeza-Yates RA, Gonnet GH (1992) A new approach to text searching. Commun ACM 35(10):74–82
36. Rabin MO, Scott D (1959) Finite automata and their decision problems. IBM J Res Dev 3(2):114–125
37. Susik R, Grabowski S, Fredriksson K (2019) Revisiting multiple pattern matching. Comput Inform 38(4):937–962

38. Brisaboa N, Iglesias E, Navarro G, Paramá JL (2003) An efficient compression code for text databases. In: Proceedings of 25th European Conference on Information Retrieval Research (ECIR'03). LNCS 2633. Springer, Berlin, pp 468–481
39. Susik R, Grabowski S, Deorowicz S (2014) Fast and simple circular pattern matching. In: Man–Machine Interactions, vol 3. Springer, pp 537–544
40. Navarro G, Raffinot M (1998) A bit-parallel approach to suffix automata: fast extended string matching. In: Proceedings of Annual Symposium on Combinatorial Pattern Matching (CPM). Springer, Berlin, pp 14–33
41. StringUtils. https://github.com/Arachnid/solidity-stringutils. Accessed 15 Apr 2022
42. Iyer K, Dannen C (2018) Building games with ethereum smart contracts. Springer, Berlin