Pattern matching algorithms in Blockchain for network fees reduction

Robert Susik†, Robert Nowotniak‡

† Lodz University of Technology, Institute of Applied Computer Science
Al. Politechniki 11, 90–924 Łódź, Poland
rsusik@kis.p.lodz.pl
‡ MetaSolid.tech, Al. Grunwaldzka 56/202, 80-241 Gdańsk, Poland,
rnowotniak@metasolid.tech

August, 2022

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 Ethereum Virtual Machine (EVM), making it a global protocol for distributed applications. The code deployment and each operation performed in EVM cost the network fee called gas, which 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 Ethereum Virtual Machine. 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 comparing to StringUtils (a popular Solidity string library).

keywords: blockchain, pattern, matching, ethereum, string

1 Introduction

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 [31]. Over the years, blockchain gained popularity [13] and became a promising technology that found application in many computer science fields. Several alternative blockchains were introduced (Namecoin1, Litecoin2, Peercoin3, etc.) before the second generation of blockchain was developed.

Ethereum [9], 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 [15] that introduces an Ethereum Virtual Machine (EVM). One of the main advantages of EVM is the support of Turing-complete [39] programming language, which allows for writing decentralized applications based on smart contracts [38]. 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 in, for instance, an “infinite” loop. The mentioned code execution cost may differ depending on the number of operations performed in a transaction. It means the infinite loop is not possible because, in the worst case, the EVM will stop processing the code by raising the “out of gas” error. A single

---

1 www.namecoin.org
2 litecoin.org
3 www.peercoin.net
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 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 is 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. Dapps are 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 [25, 2], voting [30], transportation [30], music industry [27], supply chains [11], reputation systems [3], document versioning [33], and decentralized finance [1]. The interest in Blockchain-based technologies is growing rapidly [40, 6]. Similarly to Web2.0, web3 Dapps can be reached with its alias name via DNS-like services such as ENS 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 Ethereum Virtual Machine (EVM) development, the gas fees seem to be a challenge [28, 29, 22]. 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 [23] in place of Proof-Of-Work [31]).

The transformation process of traditional applications to a Dapp is not well defined and is a subject of study [40]. Along with this process, there is an obvious need for algorithms and libraries on EVM. In [19] the authors performed an interesting analysis of computational costs using gas consumption as the metric. The gas price prediction [34, 29, 22] and transaction fee optimization [26, 11, 21] is an active subject of study.

In this paper, propose an efficient implementation of several exact pattern matching algorithms in the Solidity/YUL language, and evaluate the gas usage of mentioned algorithms on the Ethereum Virtual Machine. 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 several exact string matching algorithms for Ethereum Virtual Machine. (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, section 4 concludes the results and suggests future work.
2 Our 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 $T[0\ldots n-1]$, and a pattern $P[0\ldots m-1]$, $m \leq n$, both over a common alphabet $\sum$ of size $\sigma$, report all occurrences of $P$ in $T$, such that $P[0\ldots m-1] = T[i\ldots i+m-1]$, where $i \leq n-m-1$.

2.2 Algorithms

The string matching algorithms constitute an essential component in many software applications [10]. Over the years, tens of algorithms have been invented, most of which are modifications of the older ones [14][12]. 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 bitwise techniques and 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 the best if the pattern has at

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 [24]. KMP reads the pattern and builds a lookup table $N[0\ldots m-1]$, 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 $O(m)$.

2.2.3 Boyer-Moore-Horspool

Boyer-Moore-Horspool (BMH) algorithm [16] is a simplified variant of Boyer-Moore (BM) [7]. 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 [16]. Searching takes $O(nm)$ time in the worst case, $O(n\log_4(m)/m)$ in the average case, and $O(n/m)$ in the best case.

2.2.4 Rabin-Karp

Rabin-Karp (RK) algorithm [20] is the first that uses rolling-hash for text search purposes. The algorithm calculates a hash for the pattern $P[0\ldots m-1]$ and text window $T[0\ldots m-1]$, then moves the window towards 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 $T[i]$. The average complexity of this algorithm is $O(n+m)$, and $O(nm)$ in the worst case.

2.2.5 Shift-Or

Shift-Or (SO) algorithm [5] simulates Nondeterministic Finite Automata (NFA) [35]. It uses the bitwise techniques and 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 the 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 [8] to reduce the pattern size. The complexity of this algorithm is independent of the pattern length and equals \( O(n[m/w]) \), which gives \( O(n) \) for \( m = O(w) \).

2.2.6 Backward Nondeterministic Dawg Matching

Backward Nondeterministic Dawg Matching (BNDM) [32] is a Directed Acyclic Word Graph simulation implemented with bit-parallel techniques. The algorithm, 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. 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 \( O(n\log_2 m/m) \).

2.2.7 Stringutils

StringUtils [11], is a popular Solidity library for string operations that most developers would copy into their programs and deploy along with their smart contracts [18]. 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). If the \( i \) is the position of the first occurrence of the pattern in the text, then the returned substring is \( T[i \ldots n - 1] \). In order to evaluate its performance, we had to modify it, so it returns all the occurrences of the pattern in the text likewise other presented algorithms.

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) [9] (a personal blockchain for development). The algorithms were implemented in 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 [10]. The smart contract was deployed on the Rinkeby network [11]. 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, 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 the functions available in StringUtils. The tests were performed on datasets (dna, english, proteins, sources) from Pizza & Chilli corpus [1]. Algorithms were tested using multiple patterns sizes \( m \in \{4, 8, 12, 16, 24, 32, 64, 128, 256, 512\} \), and text sizes \( n \in \{1\text{KiB}, 16\text{KiB}, 128\text{KiB}\} \) (substrings of mentioned datasets). 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 \( m \).

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

---

1. trufflesuite.com/ganache
2. github.com/rsusik/pattern-matching-in-blockchain
3. 0x9Fb22d8d82FcF1c5321D5ac75eE917CF986E257
4. github.com/Arachnid/solidity-stringutils
5. pizzachili.dcc.uchile.cl
shifts the text window by one. The cost of \texttt{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 BHM reduces the gas usage by up to 22-folds comparing to StringUtils. Interesting is the fact that theStringUtils has an even worse result than the Naive approach for \texttt{sources}, and \( m = 512 \).

Table 1 shows gas usage and its price (fee) of searching \( m = 512 \) pattern in 128 KiB text. In this case, if we assume the current\footnote{As per 2022.07.10} gas price (about 25 Gwei) and USD/ETH exchange rate (1250 USD), the approximate cost of searching pattern of \( m = 512 \) characters in \texttt{sources} dataset using StringUtils is about 1766 whereas the same using BHM is about 79. The BHM wins for \texttt{proteins} and \texttt{sources}, but in case of \texttt{dna} and \texttt{english} the BNDM dominates. However, only in the case of \texttt{dna} the difference between BHM and BNDM is notable.

Table 2 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 KiB and 16 KiB is more
Table 1: Gas usage (in millions) and fee of searching long pattern \( (m = 512) \) in 128 KiB text

| Algorithm | dna | english | proteins | sources |
|-----------|-----|---------|----------|---------|
| 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 |

Table 2: Gas usage per text character of searching short \( (m = 16) \) pattern

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

We see, in this case, the impact of function 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 preprocessing phase.

In addition to the 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.

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 specifi-
4 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 the 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.

Acknowledgement

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

References

[1] Stringutils. [https://github.com/Arachnid/solidity-stringutils](https://github.com/Arachnid/solidity-stringutils). Accessed: 2022-04-15.

[2] E. M. Adere. Blockchain in healthcare and iot: A systematic literature review. *Array*, page 100139, 2022.

[3] A. S. Almasoud, F. K. Hussain, and O. K. Hussain. Smart contracts for blockchain-based reputation systems: A systematic literature review. *Journal of Network and Computer Applications*, 170:102814, 2020.

[4] H. Amler, L. Eckey, S. Faust, M. Kaiser, P. Sandner, and B. Schlosser. Defining defi: Challenges & pathway. In *2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, pages 181–184. IEEE, 2021.

[5] R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, 1992.

[6] R. Belchior, A. Vasconcelos, S. Guerreiro, and M. Correia. A survey on blockchain interoperability: Past, present, and future trends. *ACM Computing Surveys (CSUR)*, 54(8):1–41, 2021.

[7] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[8] N. Brisaboa, E. Iglesias, G. Navarro, and J. L. Paramá. An efficient compression code for text databases. In *Proc. 25th European Conference on Information Retrieval Research (ECIR'03)*, LNCS 2633, pages 468–481. Springer, 2003.

[9] V. Buterin. Ethereum white paper: A next generation smart contract & decentralized application platform. 2013.

[10] C. Charras and T. Lecroq. *Handbook of exact string matching algorithms*. Citeseer, 2004.

[11] A. Di Sorbo, S. Laudanna, A. Vacca, C. A. Visaggio, and G. Canfora. Profiling gas consumption in solidity smart contracts. *Journal of Systems and Software*, 186:111193, 2022.

[12] S. Faro and T. Lecroq. The exact online string matching problem: A review of the most recent results. *ACM Computing Surveys (CSUR)*, 45(2), mar 2013.

[13] A. G. Gad, D. T. Mosa, L. Abualigah, and A. A. Abohany. Emerging trends in blockchain technology and applications: A review and outlook. *Journal of King Saud University-Computer and Information Sciences*, 2022.

[14] S. Grabowski. New algorithms for exact and approximate text matching. online.

[15] H. Guo and X. Yu. A survey on blockchain technology and its security. *Blockchain: Research and Applications*, page 100067, 2022.

[16] R. N. Horspool. Practical fast searching in strings. *Software: Practice and Experience*, 10(6):501–506, 1980.
[17] A. Hume and D. Sunday. Fast string searching. *Software: Practice and Experience*, 21(11):1221–1248, 1991.

[18] K. Iyer and C. Dannen. *Building Games with Ethereum Smart Contracts*. Springer, 2018.

[19] A. Jabbar and S. Dani. Investigating the link between transaction and computational costs in a blockchain environment. *International Journal of Production Research*, 58(11):3423–3436, 2020.

[20] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[21] M. M. A. Khan, H. M. A. Sarwar, and M. Awais. Gas consumption analysis of ethereum blockchain transactions. *Concurrency and Computation: Practice and Experience*, 34(4):e6679, 2022.

[22] H.-M. Kim, G.-W. Bock, and G. Lee. Predicting ethereum prices with machine learning based on blockchain information. *Expert Systems with Applications*, 184:115480, 2021.

[23] S. King and S. Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper, August*, 19(1), 2012.

[24] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(1):323–350, 1977.

[25] A. Kumar, R. Krishnamurthi, A. Nayyar, K. Sharma, V. Grover, and E. Hossain. A novel smart healthcare design, simulation, and implementation using healthcare 4.0 processes. *IEEE Access*, 8:118433–118471, 2020.

[26] C. Li, S. Nie, Y. Cao, Y. Yu, and Z. Hu. Trace-based dynamic gas estimation of loops in smart contracts. *IEEE Open Journal of the Computer Society*, 1:295–306, 2020.

[27] Y. Li, J. Wei, J. Yuan, Q. Xu, and C. He. A decentralized music copyright operation management system based on blockchain technology. *Procedia Computer Science*, 187:458–463, 2021.

[28] L. Marchesi, M. Marchesi, G. Desteefanis, G. Barabino, and D. Tigan. Design patterns for gas optimization in ethereum. In *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 9–15. IEEE, 2020.

[29] R. Mars, A. Abid, S. Cheikhrouhou, and S. Kallel. A machine learning approach for gas price prediction in ethereum blockchain. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 156–165. IEEE, 2021.

[30] M. B. Mollah, J. Zhao, D. Niyato, Y. L. Guan, C. Yuen, S. Sun, K.-Y. Lam, and L. H. Koh. Blockchain for the internet of vehicles towards intelligent transportation systems: A survey. *IEEE Internet of Things Journal*, 8(6):4157–4185, 2020.

[31] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.

[32] G. Navarro and M. Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In *Proc. Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 14–33. Springer, 1998.

[33] N. Nizamuddin, K. Salah, M. A. Azad, J. Arshad, and M. Rehman. Decentralized document version control using ethereum blockchain and ipfs. *Computers & Electrical Engineering*, 76:183–197, 2019.

[34] G. A. Pierro, H. Rocha, S. Ducasse, M. Marchesi, and R. Tonelli. A user-oriented model for oracles’ gas price prediction. *Future Generation Computer Systems*, 128:142–157, 2022.

[35] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal*
of research and development, 3(2):114–125, 1959.

[36] M. Sober, G. Scaffino, C. Spanring, and S. Schulte. A voting-based blockchain interoperability oracle. In 2021 IEEE International Conference on Blockchain (Blockchain), pages 160–169. IEEE, 2021.

[37] R. Susik, S. Grabowski, and K. Fredriks-son. Revisiting multiple pattern matching. Computing and Informatics, 38(4):937–962, 2019.

[38] N. Szabo. Smart contracts: building blocks for digital markets. EXTROPY: The Journal of Transhumanist Thought,(16), 18(2):28, 1996.

[39] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. J. of Math, 58(345-363):5, 1936.

[40] K. Wu. An empirical study of blockchain-based decentralized applications. arXiv preprint arXiv:1902.04969, 2019.

[41] Y. Zhang, L. Chen, M. Battino, M. A. Farag, J. Xiao, J. Simal-Gandara, H. Gao, and W. Jiang. Blockchain: An emerging novel technology to upgrade the current fresh fruit supply chain. Trends in Food Science & Technology, 2022.

Appendix A

In this section we provide complementary results.
|        |        |        |        |        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|
|        |        | m      | 4      | 8      | 12     | 16     | 24     | 32     | 64     |
| dna    | BMH    | 10.76  | 7.54   | 7.41   | 7.62   | 7.34   | 5.19   | 8.52   | 8.16   | 6.46   | 7.45   |
|        | BNDM   | 27.36  | 13.86  | 10.13  | 8.30   | 5.99   | 4.80   | 2.88   | 1.76   | 1.22   | 1.21   |
|        | KMP    | 47.65  | 46.97  | 47.37  | 47.21  | 48.62  | 44.76  | 50.10  | 49.16  | 48.25  | 46.59  |
|        | Naive  | 60.54  | 60.59  | 60.03  | 59.62  | 62.06  | 61.29  | 62.48  | 61.50  | 61.85  | 61.63  |
|        | RK     | 42.69  | 42.96  | 42.24  | 42.31  | 44.07  | 43.98  | 44.56  | 43.93  | 44.04  | 44.04  |
|        | SO     | 17.30  | 17.12  | 17.26  | 17.27  | 17.32  | 17.81  | 17.46  | 17.12  | 17.37  |        |
|        | StringUtils | 23.01 | 20.35  | 20.64  | 20.60  | 20.70  | 20.84  | 41.63  | 40.53  | 40.92  | 42.05  |
| english| BMH    | 7.25   | 4.38   | 3.27   | 2.71   | 2.36   | 1.99   | 1.65   | 1.32   | 1.17   | 1.05   |
|        | BNDM   | 15.57  | 8.72   | 6.08   | 5.07   | 3.72   | 2.99   | 1.92   | 1.29   | 0.99   | 0.95   |
|        | KMP    | 34.24  | 36.57  | 35.87  | 33.60  | 34.30  | 33.83  | 35.00  | 35.15  | 34.21  | 36.83  |
|        | Naive  | 56.13  | 56.70  | 56.37  | 54.94  | 55.28  | 55.70  | 55.97  | 55.07  | 58.23  | 57.95  |
|        | RK     | 44.29  | 43.87  | 43.71  | 43.40  | 44.20  | 43.73  | 43.22  | 45.42  | 45.84  |        |
|        | SO     | 17.30  | 17.41  | 17.43  | 17.45  | 17.44  | 17.28  | 17.60  | 17.89  | 17.80  | 17.72  |
|        | StringUtils | 20.88 | 21.02  | 21.16  | 20.70  | 20.62  | 20.64  | 41.08  | 41.60  | 41.80  | 43.07  |
| proteins| BMH    | 7.08   | 4.37   | 3.28   | 2.71   | 2.16   | 2.02   | 1.55   | 1.54   | 1.47   | 1.53   |
|        | BNDM   | 14.19  | 8.84   | 6.06   | 5.15   | 3.65   | 3.59   | 2.56   | 2.71   | 1.01   | 4.18   |
|        | KMP    | 34.15  | 34.01  | 34.57  | 34.23  | 33.60  | 34.29  | 35.80  | 35.78  | 36.35  | 36.75  |
|        | Naive  | 58.07  | 56.15  | 56.13  | 59.41  | 62.52  | 62.10  | 57.62  | 57.33  | 56.33  | 57.12  |
|        | RK     | 45.54  | 43.86  | 43.68  | 47.79  | 49.04  | 47.09  | 44.52  | 44.50  | 43.58  |        |
|        | SO     | 17.32  | 17.74  | 17.41  | 17.12  | 17.20  | 17.30  | 17.73  | 17.68  | 17.59  | 17.61  |
|        | StringUtils | 21.01 | 21.19  | 20.81  | 20.58  | 20.53  | 20.57  | 41.12  | 41.26  | 41.41  | 42.13  |
| sources | BMH    | 6.52   | 3.67   | 2.71   | 2.31   | 1.90   | 1.53   | 1.14   | 0.93   | 0.80   | 0.75   |
|        | BNDM   | 12.62  | 7.23   | 5.13   | 4.12   | 3.21   | 2.54   | 1.65   | 1.16   | 0.92   | 0.92   |
|        | KMP    | 33.53  | 32.66  | 32.87  | 34.19  | 32.97  | 33.04  | 32.70  | 32.09  | 32.56  | 33.05  |
|        | Naive  | 56.12  | 56.06  | 54.49  | 55.27  | 56.73  | 55.34  | 54.65  | 55.29  | 55.59  | 55.71  |
|        | RK     | 44.13  | 44.38  | 42.98  | 44.01  | 44.50  | 43.97  | 43.62  | 43.31  | 44.37  | 44.63  |
|        | SO     | 17.70  | 17.42  | 17.49  | 17.25  | 17.26  | 17.51  | 17.18  | 17.01  | 17.17  |        |
|        | StringUtils | 21.01 | 20.79  | 20.75  | 20.60  | 20.55  | 20.70  | 40.48  | 40.15  | 40.31  | 41.47  |

Table 3: Time (in seconds) of searching pattern in 128 KiB text
| Set      | Algorithm | 4   | 8   | 12  | 16  | 24  | 32  | 64  | 128  | 256  | 512  |
|----------|-----------|-----|-----|-----|-----|-----|-----|-----|------|------|------|
| dna      | BMH       | 11.68 | 8.63 | 8.44 | 8.76 | 8.43 | 6.67 | 9.73 | 9.53  | 7.95  | 9.55  |
|          | BNDFM     | 26.78 | 14.64 | 11.24 | 9.49 | 7.28 | 6.30 | 4.45 | 3.48  | 2.97  | 2.96  |
|          | KMP       | 47.95 | 47.36 | 47.52 | 47.47 | 47.69 | 43.61 | 47.41 | 49.22 | 48.62 | 47.65 |
|          | Naive     | 62.32 | 62.50 | 62.51 | 62.29 | 62.47 | 60.55 | 62.61 | 62.26 | 62.47 | 61.99 |
|          | RK        | 43.27 | 43.21 | 43.21 | 43.21 | 43.21 | 43.21 | 43.22 | 43.24 | 43.27 | 43.33 |
|          | SO        | 17.02 | 17.01 | 17.01 | 17.01 | 17.01 | 17.01 | 17.01 | 17.02 | 17.02 | 17.06 |
|          | StringUtils | 23.21 | 22.37 | 22.37 | 22.37 | 22.36 | 22.36 | 45.68 | 47.23 | 50.33 | 56.50 |
| english  | BMH       | 8.20  | 5.69  | 4.78  | 4.29  | 3.94  | 3.61  | 3.34  | 3.04  | 2.91  | 2.83  |
|          | BNDFM     | 15.65 | 9.72  | 7.36  | 6.50  | 5.26  | 4.61  | 3.62  | 3.03  | 2.76  | 2.75  |
|          | KMP       | 34.39 | 35.37 | 34.92 | 33.53 | 33.96 | 34.02 | 34.12 | 34.98 | 34.15 | 35.42 |
|          | Naive     | 56.55 | 57.06 | 56.84 | 56.24 | 56.31 | 56.37 | 56.73 | 56.73 | 56.32 | 56.74 |
|          | RK        | 43.21 | 43.21 | 43.21 | 43.21 | 43.21 | 43.21 | 43.22 | 43.24 | 43.27 | 43.33 |
|          | SO        | 17.01 | 17.01 | 17.01 | 17.01 | 17.01 | 17.01 | 17.02 | 17.02 | 17.03 | 17.06 |
|          | StringUtils | 22.43 | 22.37 | 22.37 | 22.37 | 22.36 | 22.36 | 45.68 | 47.23 | 50.33 | 56.50 |
| proteins | BMH       | 8.08  | 5.68  | 4.65  | 4.21  | 3.80  | 3.71  | 3.30  | 3.25  | 3.23  | 3.21  |
|          | BNDFM     | 14.91 | 9.56  | 7.40  | 6.59  | 5.23  | 5.16  | 4.20  | 4.42  | 2.76  | 5.78  |
|          | KMP       | 34.36 | 34.13 | 34.72 | 34.70 | 33.15 | 34.70 | 35.63 | 34.74 | 35.07 | 36.12 |
|          | Naive     | 56.50 | 56.47 | 56.67 | 56.63 | 56.28 | 56.70 | 57.03 | 56.89 | 57.20 | 58.42 |
|          | RK        | 43.21 | 43.21 | 43.21 | 43.21 | 43.21 | 43.21 | 43.22 | 43.22 | 43.24 | 43.27 |
|          | SO        | 17.01 | 17.01 | 17.01 | 17.01 | 17.01 | 17.01 | 17.02 | 17.02 | 17.03 | 17.07 |
|          | StringUtils | 22.46 | 22.40 | 22.38 | 22.40 | 22.37 | 22.40 | 45.68 | 47.23 | 50.33 | 56.51 |
| sources  | BMH       | 7.74  | 5.20  | 4.33  | 3.93  | 3.51  | 3.23  | 2.86  | 2.64  | 2.54  | 2.53  |
|          | BNDFM     | 13.80 | 8.57  | 6.70  | 5.73  | 4.72  | 4.20  | 3.33  | 2.91  | 2.66  | 2.68  |
|          | KMP       | 33.32 | 32.69 | 32.44 | 33.25 | 33.68 | 32.79 | 33.07 | 32.60 | 32.85 | 33.59 |
|          | Naive     | 56.17 | 55.92 | 55.95 | 56.20 | 56.42 | 55.94 | 56.01 | 56.00 | 55.91 | 56.07 |
|          | RK        | 43.21 | 43.21 | 43.21 | 43.21 | 43.21 | 43.21 | 43.22 | 43.22 | 43.24 | 43.27 |
|          | SO        | 17.01 | 17.01 | 17.01 | 17.01 | 17.01 | 17.01 | 17.01 | 17.02 | 17.03 | 17.06 |
|          | StringUtils | 22.42 | 22.37 | 22.37 | 22.37 | 22.36 | 22.36 | 45.68 | 47.23 | 50.33 | 56.50 |

Table 4: Gas usage (in millions) of searching pattern in 128 KiB text