Utrecht University

BACHELOR THESIS

Intersection non-emptiness for restricted classes of Deterministic Finite Automata

Author:
Pim Veraar

Supervisor:
Dr. Benjamin Rin

Second Reader:
Dr. Johannes Korbmacher

Faculty of Humanities
Artificial Intelligence
7.5 ECTS

July 14, 2021

Abstract

In this thesis, the intersection non-emptiness problem for Deterministic Finite Automata is discussed. By applying certain restrictions to the Deterministic Finite Automata, an algorithm is constructed to analyze whether the given restricted Automata share any common strings. The time complexity of this algorithm is then discussed and compared to previous research on this subject.

Acknowledgements

I would like to thank my supervisor Benjamin Rin for his help and guidance during the writing process of the thesis. I would also like to thank Dominik Klein for his assistance in this project.

Contents

Chapter 1

Introduction

1.1 Background

The deterministic finite automaton (or DFA for short) is a simple yet powerful model of computation. It has an extremely limited memory: with any given input, it uses a single character from that input to determine what to do and then moves on to the next character. At first glance it may appear that there are not many uses for such a restrictive system, but a lot of procedures we encounter in the real world can be done with them. An example of this is simple decision-making done by AI.

All deterministic finite automata consist of *states*, with *transition functions* to tell us how to move between the different states. One of these states is the *start state*, this is the state where the first character of the input is evaluated. However, the DFA does not recognize all characters: the set of characters it does recognize is called the *alphabet* of the DFA.

Definition 1 [1, p.34] A deterministic finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the states,
2. Σ is a finite set called the alphabet,
3. $\delta : Q \times \Sigma \longrightarrow Q$ is the transition function,
4. $q_0 \in Q$ is the start state, and
5. $F \subseteq Q$ is the set of accept states.

One of the interesting things to think about is the set of the strings recognized by multiple DFAs and check if this set is empty or not. This is called the intersection non-emptiness problem (or IE for short). Trivially this can be done by constructing the Cartesian product of the DFAs, resulting in a single DFA that effectively simulates both DFAs [1, p. 46]. The result is a new DFA that recognizes the intersection of the separate regular languages. To check if the resulting set is empty, there can exist no path from the start state to an accept state. If there is, it means that there are strings which are recognized by both DFAs. The procedure of constructing the Cartesian product results in a DFA of n^k states, where n is the amount of states in the k number of DFAs. Then for checking for a path from the start state to an accept state, we at worst have to go through every state in the Cartesian product to confirm if there are shared strings between the two DFAs. From this we can define the time complexity of this algorithm to say something about its performance.

Defining the time complexity of an algorithm can be done by *asymptotic analysis* [1, p.277]. It is a way to ascribe an asymptotic upper bound to the runtime of an algorithm, it will always be lower or equal to this upper bound. We say that for two

functions f and g it holds that $f(n) = O(g(n))$, if there exist positive integers c and m such that for every integer n it holds that $n \geq m$ and $f(n) \leq cg(n)$. So this upper bound is regardless of any constant; it is true that both $n = O(n)$, $2n = O(n)$ and $3n = O(n)$ for example. Writing the asymptotic upper bound as $O(g(n))$ is called the *big-O notation*. There is also the *small-O notation*. Here we say that for two functions f and g it holds that $f(n) = o(g(n))$, if for any real number $c > 0$ a number m exists, where for every $n \geq m$ it holds that $f(n) < cg(n)$. So the difference between the two is that the small-o notation is used for strict asymptotic upper bounds. So $n = O(n)$, but $n \neq o(n)$. However it is true that $n = O(n^2)$ and $n = o(n^2)$. Using this together with the analysis of the Cartesian product construction, we can see that this method would result in a time complexity of $O(n^k)$.

1.2 Research question

But can solving intersection non-emptiness be done faster? Outright proving this has been a challenge for many decades, since it would refute the Exponential Time Hypothesis. This hypothesis states that several NP-complete problems cannot be solved in subexponential time in the worst case, and this includes the non-intersection emptiness problem. This is too broad for the scope of this study, but if we narrow down some of the sets given in Definition 1 there might be some interesting results. Examples of such restrictions on the DFA will be discussed in a moment, as this topic has already been worked on as well. So is it possible to solve intersection non-emptiness of k DFAs with n states and certain restrictions in $o(n^k)$ time?

1.3 Relevance to AI

The reason this is interesting to look at is that studying these cases may help us understand versions of DFAs that are more complex. Although they are thought to be very well understood, there are still a lot of things we do not know about them. As already mentioned, DFAs can be used for decision-making problems. Further exploring this topic could serve to support future development in AI.

1.4 Previous research

Previous work on this subject has shown that by applying certain restrictions to Definition 1 of the DFA given above, we can get better results than by constructing the Cartesian product. One such restrictions is to let the alphabet Σ only contain one symbol. These versions of DFAs are called unary deterministic finite automata (UDFA). These UDFAs all have a very similar structure, since they can have only one transition arrow per state.

Definition 2 A DFA $(Q, \Sigma, \delta, q_0, F)$ is *unary* if Σ has cardinality 1.
In what follows, we will assume that unary DFAs (UDFAs) have no states that are inaccessible from the start state, as such states have no effect on the language of a DFA. The following observation is taken from M. Wehar [2].

Observation 1 Every UDFA has a form consisting of:

1. a *handle* (a set of states that can never be visited more than once in any computation) and

2. a *cycle* (a nonempty cyclic set of states comprising all the states that can be visited more than once in a computation).

This structure can be seen in Figure 1.1 below. We can see why this is true as follows. Observe that all DFAs have at least one nonempty cycle, since δ is a total function (the automaton is deterministic) and the number $|Q|$ of states is finite. Let $D = (Q, 1, \delta, q_0, F)$ be a UDFA. Since D has a unary alphabet, we can see that any computation by D is uniquely determined by the input's length. Call this property *input-length determinism*. Observe that a computation contains a loop (there exists a state that is entered at least twice) if and only if the input length is at least $|Q|$. For, if the input is at least this long, then a loop is required by the pigeonhole principle; conversely, if there is a loop, then the number of unique states entered during the computation must be maximal, because computations are input-length determined and thus any computation from a longer input would halt in some state within the same loop. Since the number is maximal and we assumed that no states are inaccessible from the start state, it must be exactly $|Q|$.

It follows that D must have exactly one cycle. All computations on inputs of length at least $|Q|$ will loop upon reading the $|Q|^{\text{th}}$ symbol of the input. The set of states not part of this cycle is called the handle. By the preceding, we see that the handle must be a linear series of states beginning with q_0 and ending just before the first state in the cycle (i.e., the state where D would halt if it were to read the string $1^{|Q|}$).

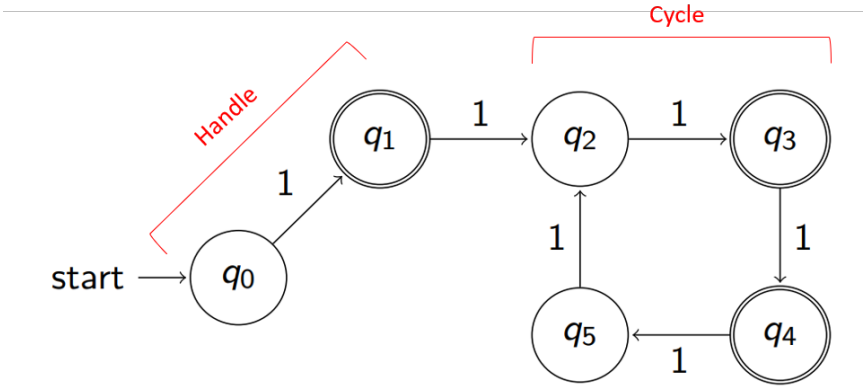

FIGURE 1.1: A visualization of an UDFA[2].

In a study by Oliveira and Wehar [3] it was found that intersection non-emptiness for two UDFAs can be solved in $O(n)$ time, and for three UDFAs in $O(n^{1.1865})$ time. They pointed out that this problem has similarities to solving a system of modular equations. For example, the UDFA in the figure below has a solution for all strings of a length congruent to 3 mod 4. After evaluating these strings we would end up in accept state q_3. We do have to take into account the handle of the UDFA: strings of a length congruent to 0 mod 4 are only allowed if that length is greater than zero. Otherwise, according to the modular equation, q_0 would be a valid solution. This is not an accept state however, so the empty string (a string of length 0) would be rejected in this UDFA. But if we concern ourselves only with the cycle part of the UDFA, the intersection non-emptiness problem is essentially reduced to solving a system of modular equations

$$\begin{cases} x = a \bmod m \\ x = b \bmod n \end{cases} \tag{1.1}$$

Both of these equations represent an UDFA. For the top equation, a represents the position of the accept state along the cycle and m represents the length of the cycle. Now, UDFA-IE denotes intersection non-emptiness for UDFAs.

1.5 Further Background

A helpful tool for solving a system of modular equations is the Chinese Remainder Theorem. It states that we can uniquely determine a solution to the system if for every pair of modular equations the moduli are coprime. This statement was first discovered by Chinese mathematician Sūnzǐ. Later in 1247, Chinese mathematician Qin Jiushao discovered a general form of the Chinese Remainder Theorem that makes use of a well-defined set of steps (or as we call it now: an algorithm) to solve the system of modular equations. This method was later rediscovered by Victor-Amédée Lebesgue in 1859. The general Chinese Remainder Theorem is a way to solve a system of modular equations even if every pair of moduli is not necessarily coprime. However, this can only be done if certain requirements are met. The following lemmas will be assumed to proof the next theorem.

Lemma 1 Let a, b and c be integers. If $a|b$ and $b|c$, then $a|c$. [4, p. 55]
Lemma 2 Let a, b and c be integers. If $a|b$ and $a|c$ then $a|(b-c)$. [5, p. 9]

Theorem 1 *Equation 1.1 has a solution if and only if* $gcd(m,n) \mid (a-b)$. *[5, p.51]*

Proof: By definition of modular equations, we know that $m \mid (x-a)$ and $n \mid (x-b)$. By definition of the greatest common divisor, we also know that $\gcd(m,n) \mid m$ and $\gcd(m,n) \mid n$. Because $\gcd(m,n) \mid m$ and $m \mid (x-a)$, by Lemma 1 it follows that $\gcd(m,n) \mid (x-a)$.
Likewise, it holds that $\gcd(m,n) \mid (x-b)$. From $\gcd(m,n) \mid (x-b)$ and $\gcd(m,n) \mid (x-a)$ it follows by Lemma 2 that $\gcd(m,n) \mid ((x-b)-(x-a))$. Rewriting this results in $\gcd(m,n) \mid (a-b)$. □

This just proves the existence of a solution, but what if we are interested in what the solution is? This is were we introduce Qin's method. It aims to combine two equations into a single unique solution. Doing this pairwise repeatedly, we reduce the size of the system by one equation each step. For this, we use Bézout's identity:

Lemma 3 (Bézout's identity) Let a and b be integers with greatest common divisor d. Then there exist integers x and y called the Bézout coefficients such that $ax + by = d$. Using this together with Equation 1.1 we can write

$$mu + nv = g \tag{1.2}$$

where m and n are the moduli, u and v are the Bézout coefficients and $g = \gcd(m,n)$. Rewriting Equation 1.2 gives us

$$\frac{m}{g}u + \frac{n}{g}v = 1 \tag{1.3}$$

$$\frac{m}{g}u = 1 - \frac{n}{g}v \tag{1.4}$$

Theorem 2 *Given that the system in Equation 1.1 has a solution, a solution x is given by $a\frac{n}{g}v + b\frac{m}{g}u$ where u and v are the Bézout coefficients of the moduli and g the greatest common divisor of the moduli.*

Proof: Filling in Equation 1.4 gives us

$$x = a\frac{n}{g}v + b(1 - \frac{n}{g}v)$$

$$x = b + (a - b)\frac{n}{g}v$$

$$x = b + \frac{(a-b)}{g}vn \tag{1.5}$$

In the last step the $(a - b)$ is moved to the top of the fraction and n to the back to make it more clear what is now going on. The usefulness of Theorem 1 becomes apparent: v is a Bézout coefficient and will therefore be an integer by definition, while $\frac{(a-b)}{g}$ will be in integer only if $\gcd(m, n) \mid (a - b)$. This is the case, since we assumed there would be a solution to the system. Then it follows that $b + \frac{(a-b)}{g}vn \equiv b \bmod n$. The same thing can be done to show this satisfies the other modular equation.

However, the end result is also a modular equation, where the modulus is given by the *least common multiple* of m and n.[6, p. 60] The end result of reducing two equations from the system into one will be

$$\frac{anv + bmu}{g} \bmod \operatorname{lcm}(m, n) \tag{1.6}$$

□

1.6 Scope and Structure

The goal of this study will be to apply similar restrictions to the DFAs and find different ways to improve on the already known bounds for solving intersection non-emptiness. In the next chapter I will construct an algorithm for solving this problem with other restrictions to the DFAs besides alphabet size. Specifically we will restrict the maximum number of accept states and defining its time complexity. After this I will discuss my findings and possible future research.

Chapter 2

Results

Another interesting restriction, besides a limitation on the size of the alphabet like in UDFAs, is to fix the number of accept states allowed. This naturally has implications for the representation of a UDFA as a system of modular equations.

2.1 3-UDFA-IE with 1 accept state

We start with an easy case where we are given 3 UDFA's with all one accept state each. Here we can lay the groundwork for how the algorithm will work and what its time complexity is. Later we will expand on these findings by increasing the number of UDFAs and accept states.

Theorem 3 *Intersection non-emptiness for 3 UDFAs with 1 accept state is decidable in $O(n \log n)$ time.*

Proof: As we have discussed previously, the conversion from the UDFAs to a system of modular equations only works for the cycle of the UDFA, not the handle. This can be solved by trying out a string of every length up to the maximum length of the handles of all the UDFAs. So, first we try out the empty string to see if it is accepted by all 3 UDFAs. If this is not the case, we try out the string of length 1. We continue this process until evaluating the string results in a state that is in the cycle for all UDFAs. If we find that there is a string that is accepted by all UDFAs this way, intersection non-emptiness is already proven and we can stop. We can also stop if one of the UDFAs has their only accept state in the handle, if no string is recognized by all 3 UDFAs in this procedure. Because by definition of the handle, checking the accept state once is enough to determine intersection non-emptiness, since we cannot return to this accept state.

If for no strings an accept state is reached for all UDFAs this way, we can construct the appropriate system of modular equations. In the case of 3 UDFAs, this would be represented as

$$\begin{cases} x = a \bmod m \\ x = b \bmod n \\ x = c \bmod p \end{cases} \tag{2.1}$$

Now, by the general Chinese Remainder Theorem, we know that for every pair of equations it has to be true that the greatest common divisor of the moduli divides the difference of the remainders. For 3 UDFAs this would mean that

$$\gcd(m,n)|(a-b) \wedge \gcd(m,p)|(a-c) \wedge \gcd(n,p)|(b-c). \tag{2.2}$$

Although it might not be the most efficient, a way of checking divisibility is to simply divide and check the result. If this divisibility test is true for every pair, we

know by Theorem 1 that there is a solution to the system of modular equations. This solves the intersection non-emptiness problem. The solution of the system is not relevant for the intersection non-emptiness problem, either there are strings that are accepted by all UDFAs or there are none. But it is worth noting that the result x of the system represents the length of the string that is accepted by all UDFAs, *if we would start at the beginning of the cycle*. We of course have to take into account the adjustment that needs to be made to x for our assumption to leave out the states in the handle, if we want x to truly represent the length of this universally accepted string.

Analyzing the time complexity of this algorithm, we consider the worst-case scenario. In the first part of the algorithm, where we try to find a solution in the handles of the UDFAs, worst-case we have to check all n states if one of the UDFAs has a cycle that is only its last state. Thus, this step takes $O(n)$ time. The next step of the algorithm is testing for divisibility. To perform these tests we need to first find the greatest common divisor of the moduli and then do a division. The gcd can be calculated by the *Euclidean algorithm*, which at worst takes $O(\log n)$ time [7, p. 360]. If we use efficient ways for division and storing the information of the UDFA [8], this step takes $O(n \log n)$ time. As we can see from Equation 2.2, we have to do this step 3 times in the case of 3 UDFAs. This does not influence the time complexity of this step, as it is constant. So overall, the time complexity of the entire algorithm is determined by the time complexity of the divisibility tests, making the time complexity of this algorithm $O(n \log n)$ as well.

Alternatively we could apply Qin's method to reduce the system of modular equations by constructing Equation 1.6 and repeat this step. We can calculate the gcd and Bézout coefficients by using the *Extended Euclidean algorithm*. It has the same time complexity as the Euclidean algorithm, but it uses more space for the calculation of the Bézout coefficients. The lcm can be calculated with multiplication and division, as $\text{lcm}(m,n) = \frac{ab}{\gcd(m,n)}$. Then using efficient ways for multiplication and division, this step has a time complexity of $O(n \log n)$. Compared to the pairwise divisibility tests, we only have to take a $O(n \log n)$ time step 2 times instead of 3. But since this is a constant, there won't be a difference overall in time complexity of the algorithm between doing divisibility tests pairwise and applying Qin's method. The time complexity of the algorithm will thus be $O(n \log n)$. □

2.2 k-UDFA-IE with 1 accept state

Now it might be interesting to look cases where we are given k UDFAs with 1 accept state. This implies that we can have a lot more UDFAs now, resulting in a possibly bigger system of modular equations.

Theorem 4 *Intersection non-emptiness for a constant k UDFAs with 1 accept state is decidable in $O(n \log n)$ time.*

Proof: This proof works similarly to the proof to Theorem 3, but now we have a system of k modular equations

$$\begin{cases} x = a \bmod m \\ x = b \bmod n \\ \vdots \\ x = z \bmod q \end{cases} \tag{2.3}$$

where k is a constant given to us. As shown in the proof to Theorem 3, the time complexity of the algorithm is determined by the divisibility tests. In the case of k modular equations that we have to check pairwise, $\binom{k}{2}$ divisibility tests have to be done. If we use Qin's method, we have to take $k-1$ steps. However we see again that this is a constant, as the premise assumed k is constant. This means that the time complexity of the algorithm still is $O(n \log n)$. □

2.3 k-UDFA-IE with 2 accept states

So far we have only considered cases where the UDFAs have only 1 accept state. But what if we consider UDFAs that have multiple accept states? A simple case to start with is to allow a maximum of 2 accept state per UDFA. This means that the UDFA can have 2 accept states, but it's still possible for UDFAs to have only 1 accept state.

Theorem 5 *Intersection non-emptiness for a constant k UDFAs with a maximum of 2 accept states is decidable in $O(n \log n)$ time.*

Proof: If we allow UDFAs to have more than 1 accept state, it will alter the system of modular equations. For example, if the first UDFA has 2 accept states both in its cycle, the system of modular equations would be

$$\begin{cases} x = a_1 \bmod m \text{ OR } x = a_2 \bmod m \\ x = b_1 \bmod n \text{ OR } x = b_2 \bmod n \\ x = c_1 \bmod p \text{ OR } x = c_2 \bmod p \\ \vdots \\ x = z_1 \bmod q \text{ OR } x = z_2 \bmod q \end{cases} \tag{2.4}$$

Notice that the moduli for the equations on the first line of the system are the same, as it represents the size of the cycle (and it is still the same UDFA). If one of the UDFAs has only one accept state, for example the second one, we could say that this corresponds to the assertion that $b_1 = b_2$. Each line of this system is not really an equation but a disjunction of equations, we cannot simply use the general Chinese Remainder Theorem like before. Instead we have to execute the formulated algorithm with 1 disjunct of each line. The number of such possible combinations is 2^k, which at worst is the number of divisibility tests we have to do if all the UDFAs had 2 accept states. But this is constant, so the time complexity is still $O(n \log n)$. □

2.4 k-UDFA-IE with j accept states

An obvious next step to take is to see what happens if we introduce a new constant j, which will represent the maximum number of accept states. It is still possible here for UDFAs to just have 1 accept state, but some equations in the system will consist of a lot of disjuncts.

Theorem 6 *Intersection non-emptiness for a constant k UDFAs with a maximum of a constant j accept states is decidable in $O(n \log n)$ time.*

Proof: UDFAs with maximum j accept states will result in a system of modular equations with up to j disjuncts

$$\begin{cases} x = a_1 \bmod m \text{ OR } x = a_2 \bmod m \text{ OR ... OR } x = a_j \bmod m \\ x = b_1 \bmod n \text{ OR } x = b_2 \bmod n \text{ OR ... OR } x = b_j \bmod n \\ x = c_1 \bmod p \text{ OR } x = c_2 \bmod p \text{ OR ... OR } x = c_j \bmod p \\ \vdots \\ x = z_1 \bmod q \text{ OR } x = z_2 \bmod q \text{ OR ... OR } x = z_j \bmod q \end{cases} \tag{2.5}$$

Adjusting the maximum accept states j will affect the number of divisibility checks we have to do. In the worst case there are k UDFAs with all the maximum j accept states, which would result in j^k possible combinations of modular equations. This means we would have to do j^k divisibility checks. Although this might seem like a lot, both k and j are specified to be constant, so their result will be constant as well. The time complexity of the algorithm will thus be $O(n \log n)$. □

Chapter 3

Discussion

As we have seen from the results, intersection non-emptiness is decidable in $O(n \log n)$ time for DFAs with an alphabet of size 1 and limited accept states. This is faster than the $O(n^k)$ time needed to construct the Cartesian product, where n was the amount of states in the DFA and k the amount of DFAs. Even though it is not faster for solving intersection non-emptiness with 2 UDFAs compared to the linear time Oliveira and Wehar found in 2018, it is a slight improvement for solving it with 3 UDFAs compared to $O(n^{1.1865})$ they found. However, the UDFAs used in this thesis are even further restricted, also putting a limit on the number of accept states. Without a constant number of accept states given, the number of accept states would be dependent on the total amount of states n. The implication will be that the algorithm would now have a time complexity of $O(n^2 \log n)$, which is worse than the previously known bounds by Wehar. Problems might also arise when we do not have a constant k UDFAs. What if we allow any number of UDFAs with a constant j accept states. Suddenly this problem becomes much harder, as n could now also be the number of UDFAs.

But can we really be so pessimistic? Surely not all these variables can be n. If the number of accept states is n, than applying our algorithm would still be fast, since every string would be an accepted string in UDFA's where all states are accept states. It might also be too harsh to expect both the number of UDFAs and the number of states to be n.

An interesting thing about intersection non-emptiness for UDFAs is their deep connection to number theory. The conversion to the system of modular equations and solving that system heavily relies on very fundamental concepts of number theory, like the general Chinese Remainder Theorem. Because of this reliance, future work on this subject could be focused on the number theory elements in this thesis. For example, the divisibility tests of the algorithm determines the overall time complexity. But is it really necessary to do this? In the end, we do not care for what the actual outcome of the division is, we just need to know if the result will be an integer. A simple example: if we want to know if 2 divides some really big number, we would just have to look at the last digit and check if that is even. Is something like this possible, where we test for divisibility and receive a simple yes or no answer, for all numbers? This might significantly speed up the algorithm.

Other possible future research might include something similar to this study, where restrictions are applied to deterministic finite automata to solve the intersection non-emptiness problem. As we have discussed in the introduction, DFAs are thought to be well understood, but maybe there are other problems related to DFAs that remain unsolved. An example of this would be the seperating words problem [9, p. 105]. Given two strings, what is the smallest DFA that would accept one string but reject the other? This problem was first formulated in 1986 [10], and recently there has been found a new lowest upper bound for solving this problem [11]. It is possible that this result could be expanded on even further.

References

[1] M. Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2012. ISBN: 9781285401065.

[2] M. Wehar. *DFA Intersection Problems*. 2019. URL: `http://www.michaelwehar.com/documents/mcafl2019.pdf`.

[3] M. de Oliveira Oliveira and M. Wehar. "Intersection Non-emptiness and Hardness Within Polynomial Time". In: *Developments in Language Theory* (2018). Ed. by M. Hoshi and S. Seki, pp. 282–290.

[4] E.D. Bloch. *Proofs and Fundamentals: A First Course in Abstract Mathematics*. Undergraduate Texts in Mathematics. Springer New York, 2011. ISBN: 9781441971272.

[5] D.C. Marshall et al. *Number Theory Through Inquiry*. MAA textbooks. Mathematical Association of America, 2007. ISBN: 9780883857519.

[6] G.A. Jones and J.M. Jones. *Elementary Number Theory*. Springer Undergraduate Mathematics Series. Springer London, 2012. ISBN: 9781447106135.

[7] D.E. Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Pearson Education, 2014. ISBN: 9780321635761.

[8] D. Harvey and J. van der Hoeven. "Integer multiplication in time O(n log n)". In: *Annals of Mathematics* 193.2 (2021), pp. 563–617.

[9] J. Shallit. *A Second Course in Formal Languages and Automata Theory*. A Second Course in Formal Languages and Automata Theory. Cambridge University Press, 2009. ISBN: 9780521865722.

[10] P. Goralčík and V. Koubek. "On discerning words by automata". In: *Automata, Languages and Programming* (1986). Ed. by Laurent Kott, pp. 116–122.

[11] Z. Chase. "A New Upper Bound for Separating Words". In: *ArXiv* abs/2007.12097 (2020).