Approximate Streaming Regular Pattern Matching

Adrien Mathieu

February 29, 2024

The goal of the internship is to generalize the algorithm that solves the streaming regular pattern matching problem from [Dud+22] (which we will call the *base article*), by searching for *approximate matches*.

The result of the internship is an algorithm that works if we allow one mismatch, although its time complexity is far from ideal. Part of this solution has been formalized; the rest of the solution has been checked in details, but hasn't been properly written. Besides this

- a minor technical problem has been identified and patched in the base article;
- a an idea for improving the time complexity of the algorithm presented in the base article has been discussed, but not formalized.

Problem Statement

The problem we are trying to generalize is that of regular pattern matching, where, given a regular expression (the *pattern*) and a string (the *text*), we want to know which positions of the text that end substrings which matches the pattern.

Definition 1 (Streaming model)**.** *The streaming model is akin to the usual RAM-word model, except that the text is fed to the algorithm letter by letter, and the algorithm has no access to any character of the text but the one it is currently being presented. This means that any additional character that the algorithm wishes to remember has to be counted in its space complexity.*

Additionally, before receiving the first character of the text, the algorithm is receives the pattern and is free to do any amount of precomputations.

The algorithm should answer whether there is a match ending at the character that is currently being provided to it before seeing the next character. The time complexity is expressed per character.

Since it takes a linear space complexity to simply store all the information seen so far, we aim at a sublinear space complexity (otherwise the algorithm has the same space

complexity as in the read-only model), both in the size of the text and in the size of the pattern.

The algorithm from the base article solves the streaming regular pattern matching problem. We aim at generalizing the problem further by searching for substrings of the text that are at Hamming distance at most k from a string that is matched by the pattern, where k is also a parameter of the problem. We provide here a solution for the case $k = 1$, which is heavily based on the exact algorithm.

Solution overview

Given a regular expression R, we build a non-deterministic automaton that recognized the same language, and has the following property: every node either has any number of outgoing ε-transitions, or has exactly one outgoing transition labeled with a character. We therefore think of the nodes linked together by a transition labeled with a character has a single, atomic string. For instance, the automaton for the regular expression `ab(acb|bcba)ac` is

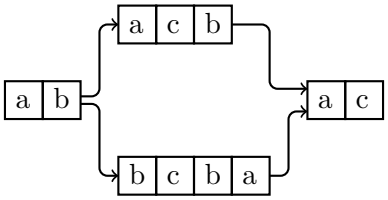


Figure 1: Automaton of the regex `ab(acb|bcba)ac`

where only the ε-transitions are represented. The four atomic strings of this regular expression are `ab`, `acb`, `bcba` and `ac`.

Definition 2 (Preceding atomic string)**.** *We say that an atomic string A* precedes *an other atomic string B (or a prefix of B) if there is a path labeled only by ε-transitions from the end of A to the beginning of B in the automaton.*

Example 3. *In Figure 1, the atomic string* `acb` *and* `bcba` *both precede* `ac`*, but not* `ab`*.*

The solution of the base paper is built upon a subroutine that simply solves the streaming pattern matching problem (that is, that finds *occurrences of a string* rather than matches of a regular expression in a streaming fashion), by stitching together the information about occurrences of the atomic strings.

Our solution uses a similar subroutine, but that finds approximate occurrences. We adapt the exact algorithm from the base paper to work with approximate occurrences rather than exact ones. Most of the invariants in the said article hold (modulo minor modifications) except for a single corner case, for which we have only recovered a workaround in the case $k = 1$. Hence be believe that our strategy could be improved to work for any k.

The general strategy of our algorithm is the same as in the base algorithm. We will present here the differences.

1. Finding and Storing Occurrences

Instead of finding exact matches for the canonical prefixes, we can use an approximate equivalent (such as presented in [CKP] or [BK23]).

A key insight used for the exact algorithm, used to handle the case of periodic strings, is [Dud+22, Observation 3.7], which states

Observation 4 (stems from [FW65])**.** *Let P and X be two strings, with $|X| \leq 2|P|$. If P has at least three occurrences in X, then P must be periodic and the set of occurrences of P in X forms an arithmetic progression with difference ρ, where ρ is the period of P.*

we replace this result with an approximate counterpart

Observation 5 (Theorem 3.1 and Theorem 3.2, from [CKW20])**.** *Let P and X be two strings, with $|X| \leq 2|P|$, and $k \in \mathbb{N}^*$. If there at least $1152k + 1$ k-mismatch occurrences of P in X, then these occurrences form at most $6k$ distinct arithmetic progressions, with step ρ where ρ is a number that depends only on P.*

While we search for approximate occurrences of periods of periodic canonical prefixes, we only exact streaks. When we find an occurrence of a period with a mismatch ending a streak, we continue the streak for a small number of periods, but in parallel we start a new streak. All the mismatches in a streak of P can only be within $|P|$ characters of the end of the streak.

2. Storing and Finding Witnesses

Given a periodic canonical prefix P, and a streak S of $\Delta(P)$, its compact representation of witnesses is similar to that of the base algorithm, except for witnesses of canonical prefixes Q which overlap S, and such that both halves Q_1 and Q_2 of Q are periodic, with the same period as P. In this case, we store:

- all occurrences of Q_2 that make Q overlap with S (they form arithmetic progressions);
- an (exact) streak for occurrences of Q_1 that make Q overlap with S.

This is important because, unlike in the exact case, there can be more than a single overlapping occurrence per canonical prefix for a given streak. The edge case we have distinguished above is the only one where there can be an unbounded number of such occurrences.

Since, for each witness, we also store the number of mismatches in its partial match, we need to update the procedure to find witnesses: if r is an occurrence of a canonical prefix P with k_1 mismatches, and if there is a witness r' of a preceding atomic string of P with

k_2 mismatches, such that $k_1 + k_2 \leq k$, such that $r' + |P| = r$, then r is a $k_1 + k_2$-mismatch witness of P. This is very similar to the way the base algorithm works for finding whether an occurrence is a witness. The only difference is due to the aforementioned difference in the compact representation of witnesses. There is a single edge case where the witness we need is stored in this different form:

- the witness we look for is of a canonical prefix $Q = Q_1 Q_2$;
- there is a mismatch between Q_2 and the streak;
- Q_1 and Q_2 are both periodic, with the same period as the streak;
- the witness of Q is overlapping with the streak.

To cover this case, we

- iterate over all overlapping occurrences of Q_2 (we have stored their arithmetic progression);
- check, using the streak of Q_1, whether that occurrence of Q_2 is a witness;
- check, if that occurrence is a witness, whether it makes the current occurrence a witness.

3. Streak Queries

The streak queries in the base algorithm rely on the following theorem to be implemented efficiently:

Theorem 6 ([Dud+22, Theorem 2.8])**.** *There exists an algorithm which, given a directed multigraph G with non-negative integer weights on edges, its two nodes v_1 and v_2 and a number x, decides if there is a walk from v_1 to v_2 of total weight x in $\mathcal{O}((|E(G| + |V(G)|^3)x \operatorname{polylog} x)$ time and $\mathcal{O}((|E(G) + |V(G)|^3) \operatorname{polylog} x)$ space, and succeeds with probability at least $1/2$.*

We replace it with the following generalization, whose proof is given in the Appendix A.

Theorem 7. *There exists an algorithm which, given a directed multigraph G, a weight function $w : E(G) \to \mathbb{N}$, a cost function $c : E(G) \to \mathbb{N}$, to nodes s and t, a target weight $\tilde{w} \in \mathbb{N}$ and a target cost $\tilde{c} \in \mathbb{N}$ such that $\tilde{w} = \Omega(\tilde{c})$, decides whether there is a walk p from s to t in G such that $w(p) = \tilde{w}$ and $c(p) = \tilde{c}$ in $\mathcal{O}(|V(G)|^2 \cdot (|E(G) + |V(G)|^3|) \cdot \tilde{w}\tilde{c}^2 \cdot \operatorname{polylog}(\tilde{w}, \tilde{c}))$ time and $\mathcal{O}((|E(G)| + |V(G)|^3)\tilde{c}^2 \operatorname{polylog} \tilde{w})$ space with probability at least $1/2$.*

This allows us to constraint the number of mismatches in our queries, in addition to the distance between the witness and the occurrence.

4. Other Work

We report here tangential work that has been carried out during the internship.

4.1. Minor problem in the base article

The algorithm described in the base article did not distinguish identical atomic strings that appeared several times in the regular expression, leading to false-positives. For instance, for the automaton shown in Figure 2, after having read AB, the algorithm would store a witness for B, and therefore after having read ABD, the algorithm would (most likely) report a match, even though ABD is not actually a walk from the start of the automaton to the end of D.

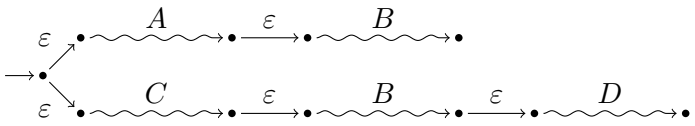


Figure 2: Automaton illustrating the bug in the base article

The patch for this bug is relatively straightforward: the representation of a canonical prefix also contains its ending node in the automaton, so that two occurrences of a canonical prefix in the automaton can be distinguished. Then, when considering preceding canonical prefixes, we only consider those that actually correspond to a preceding node in the automaton.

4.2. Improving the time complexity of the base algorithm

The running time of the base algorithm is $\tilde{\mathcal{O}}(nd^5)$ per character. The n factor is due to a single subroutine of the algorithm: the one which, given a streak, a witness at the beginning of the streak, and an occurrence toward the end of the streak, returns whether that occurrence is a witness considering only the given witness as a starting point. The time complexity of that query is proportional to the distance between the witness and the occurrence given as arguments, which is itself (at most) the size of the streak. Since a streak might run for the whole text, the time complexity of this subroutine is $\tilde{\mathcal{O}}(n)$.

Two ideas have been mentioned to improve this:

- regularly update the witnesses that are remembered in a streak, so that instead of being at the beginning of the streak, the are not too far from its end;
- truncate the streaks after a certain size, and remember more, shorter streaks.

If done properly, both ideas should lead to a distance between the witnesses and the occurrence that is bounded by the size of the canonical prefix of which it is the streak we are considering. All in all, this means turning the factor n into m, which is good because we expect m to be much smaller than n (even though this complexity is still not tight).

References

[BK23] Sudatta Bhattacharya and Michal Koucký. "Streaming K-Edit Approximate Pattern Matching via String Decomposition". In: ed. by Kousha Etessami, Uriel Feige, and Gabriele Puppis. Vol. 261. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl Leibniz-Zentrum für Informatik, 2023, 22:1–22:14. ISBN: 978-3-95977-278-5. DOI: 10.4230/LIPIcs.ICALP.2023.22.

[CKP] Raphael Clifford, Tomasz Kociumaka, and Ely Porat. "The Streaming K-Mismatch Problem". In: pp. 1106–1125. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611975482.68.

[CKW20] Panagiotis Charalampopoulos, Tomasz Kociumaka, and Philip Wellnitz. "Faster Approximate Pattern Matching: A Unified Approach". In: Nov. 2020, pp. 978–989. DOI: 10.1109/FOCS46700.2020.00095.

[Dud+22] Bartomiej Dudek, Pawe Gawrychowski, Garance Gourdel, and Tatiana Starikovskaya. "Streaming Regular Expression Membership and Pattern Matching". In: Proceedings. Society for Industrial and Applied Mathematics, Jan. 2022, pp. 670–694. DOI: 10.1137/1.9781611977073.30.

[FW65] N. J. Fine and H. S. Wilf. "Uniqueness Theorems for Periodic Functions". In: 16.1 (1965), pp. 109–114. ISSN: 0002-9939. DOI: 10.2307/2034009. JSTOR: 2034009.

Appendix

A. Proof of Graph Subroutine Theorem

Recall the following facts:

Fact 8 (cf. [Dud+22, Theorem 5.2]). *There exists an algorithm which, given $y \in \mathbb{N}$, finds in $\mathcal{O}(y \operatorname{polylog} y)$ time and $\mathcal{O}(\log y)$ space a prime $p \in \mathbb{P}$ and an $\omega \in \mathbb{F}_p$ such that, for any $N = 2^{\mathcal{O}(y \log y)}$, with probability at least $1/2$, we have:*

1. *there exists a t, such that $y \leq t = \mathcal{O}(y \operatorname{polylog} y)$ and ω is a t-th root of the unity in $\mathbb{F}_p$;*
2. *$p = \mathcal{O}(y^2 \operatorname{polylog} y)$;*
3. *$p \nmid N$.*

Fact 9 (cf. [Dud+22, Theorem 5.1]). *Let $p \in \mathbb{P}$ a prime, $t \geq 1$ and $\omega \in \mathbb{F}_p$ a t-th root of the unity in $\mathbb{F}_p$. Let C be a circuit over $\mathbb{F}_p^t$, with convolution and addition gates, and singleton inputs (that is, every input node of C has at most one nonzero entry), which outputs $\operatorname{out} C \in \mathbb{F}_p^t$ without overflowing.*

Given $x \in [t-1]$, $(\operatorname{out} C)[x]$ can be computed in $\mathcal{O}(|C| t \operatorname{polylog} p)$ time and $\mathcal{O}(|C| \log p)$ space.

We will now prove Theorem 7.

Proof of Theorem 7. We note $[n] = \{0, \dots, n\}$ for $n \in \mathbb{N}$. We say that a walk p is a (d, l)-walk if its weight is d and if its cost is l.

Small $\tilde{w}$ For this first case, assume $\tilde{w} \leq |V(G)|$. Let us compute arrays C_n for $n = 0, \dots, \lceil \log \tilde{w} \rceil$, indexed by nodes $u, v \in V(G)$ and $l \in [\tilde{c}]$ where $C_n[u, v, l]$ is a bit-vector of length $\tilde{w} + 1$ such that:

1. $C_n[u, v, l][d] = 1$ implies that there exists a walk of weight (d, l) from u to v;
2. for every $d \leq 2^n$, if there exists a walk of weight (d, l) from u to v in G, we have $C_n[u, v, l][d] = 1$.

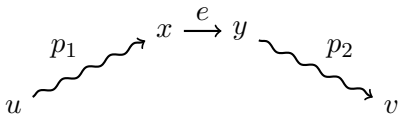
We define the $(\textsc{or}, \textsc{Convolution}_x)$-product $\odot$ of multidimensional arrays if bit vectors, truncated after the first $\tilde{w} + 1$ bits: for any $u, v \in V(G)$, $l \in [\tilde{c}]$, $d \in [\tilde{w}]$,

$$(A \odot B)[u, v, l][d] = \bigvee_{\substack{w \in V(G) \\ i \in \{0, \dots, d\} \\ j \in \{0, \dots, l\}}} A[u, w, j][i] \wedge B[w, v, l - j][d - i]$$

Once we have computed C_0, we can compute inductively C_n with

$$C_{n+1} := C_n \odot C_0 \odot C_n$$

Lemma 10. *For any (non-empty) path $p = e_0 \dots e_r$, there exists an edge e_n such that the paths $p_1 := e_0 \dots e_{n-1}$ and $p_2 := e_{n+1} \dots e_r$ have weight at most $\frac{w(p)}{2}$.*



Proof. Consider p_1 the longest prefix of p of weight at most $\frac{w(p)}{2}$, and p_2 the longest suffix of p of weight at most $\frac{w(p)}{2}$. If they overlap, then one can pick e_n to be anywhere in their overlap, and we have what we wanted.

Otherwise, let p_1 be $e_0 \dots e_{n-1}$ and p_2 be $e_{m+1} \dots e_r$. If $n = m$, we have cut the path p in exactly two paths an a single edge dividing them, with the weight of p_1 and p_2 being at most half of that of p. If $n < m$, by definition $w(p_1 e_n) > \frac{w(p)}{2}$ and $w(e_m p_2) > \frac{w(p)}{2}$, so $w(p) \geq w(p_1 e_1) + w(e_2 p_2) > w(p)$, which is absurd. □

Because of Theorem 10, by induction, the (C_n) satisfy the announced properties.

The answer to our query is then simply stored in $C_{\lceil \log x \rceil}[s, t, \tilde{c}][\tilde{w}]$. Without taking into account the time to build C_0, which fits in the space and time bounds of the theorem, this runs in $\mathcal{O}(|V(G)|^2 \tilde{w} \tilde{c})$ space and $\mathcal{O}(|V(G)| \tilde{c} \tilde{w} \operatorname{polylog} \tilde{w})$ time using the fast Fourier transform.

Big $\tilde{w}$ Let us now assume $\tilde{w} \geq |V(G)|$. As in the previous case, we are going to build iteratively a sequence of arrays D_n that store the information about longer and longer paths between any two vertices in G. There are three main differences with the C_n, though:

- the D_k are going to count the number of paths between two nodes, not just store a bit to indicate whether there is one: in the end, we just check whether the number of paths is nonzero to know whether there is one;
- we are not going to build the D_k as is. Instead, we are going to build them *modulo a certain* $p \in \mathbb{P}$ and, if we are lucky enough the number of $(\tilde{w}, \tilde{c})$)-walks from s to t will not be divisible by p (if there is at least one such path), so we can still check whether the result is nonzero modulo p to check whether there is a path;
- in C_n, there can be nonzero entries that represent paths of length above 2^k, whose weight is large. With D_ns, we are going to be more picky when counting paths to avoid those heavy walks. To do so, we are going to multiply the weight of the walks we consider by $1 + \varepsilon$ at each step, rather than 2, with a well-chosen ε.

Let $\varepsilon = \frac{1}{\log x}$, which is strictly smaller than 1 for x large enough. Let D_0 be defined as follows: for any $u, v \in V(G)$, for any $l \in [\tilde{c}]$ and $d \in [\tilde{w}]$, we let $D_0[u, v, l][d] = 1$ if $d \in \{0, 1\}$ and there is a (d, l)-walk from u to v. Otherwise, $D_0[u, v, l][d] = 0$. To define D_n for $n > 0$, we first need to define an n-good triplet (n_1, n_2, n_3): such a triplet is n-good if

(a) $(1+\varepsilon)^{n_1-1} + (1+\varepsilon)^{n_2-1} + (1+\varepsilon)^{n_3-1} \leq (1+\varepsilon)^n$

(b) $2 \cdot (1+\varepsilon)^{\max\{n_1,n_3\}-1} \leq (1+\varepsilon)^n$

Then, for $n > 0$, we define D_n with

$$D_n := \biguplus^*_{(n_1,n_2,n_3)\ n\text{-good}} D_{n_1} \odot B_{n_2} \odot D_{n_3}$$

where B_n describes all pairs of nodes connected by a (d,l)-edge, with $d \leq (1+\varepsilon)^n$. Since, if (n_1,n_2,n_3) is n-good, $n_i < n$ for $i = 1,2,3$, this definition is sound.

This refines our search: instead of considering any edge to connect two paths of weight 2^n, to form any path of weight at most 2^{n+1}, which makes us consider some paths with arbitrary high weight, we only connect paths in D_{n_1} and D_{n_3} by edges in B_{n_2}, ensuring that their total weight is bounded. More precisely, we are going to show that, for every $n = 0, \ldots, \lceil \log_{1+\varepsilon} x \rceil$:

1. $D_n[u,v,l][d] > 0$ implies that there exists a (d,l)-walk from u to v in G;
2. for each $d \leq (1+\varepsilon)^n$, if there exists a (d,l)-walk from u to v in G, then $D_n[u,v,l][d] > 0$;
3. $D_n[u,v,l][d] = 0$ for all $d > (1+\varepsilon)^n \cdot (1+\varepsilon)^{2n\cdot\log(1+\varepsilon)}$.

The two first properties follow, by induction, from Theorem 10. The following lemma proves the third property.

Lemma 11. *For every n and every n-good triple (n_1,n_2,n_3), the largest weight of a walk in $D_{n_1} \odot B_{n_2} \odot D_{n_3}$ is at most $(1+\varepsilon)^n \cdot (1+\varepsilon)^{2n\cdot\log(1+\varepsilon)}$.*

Proof. By induction on n. This is clearly true for D_0, as its only non-zero entries are for $d \in \{0,1\}$.

For $n > 0$, let (n_1,n_2,n_3) be a n-good triple and, without loss of generality, assume $n_1 \leq n_3$. The walks in $D_{n_1} \odot B_{n_2} \odot D_{n_3}$ have weight at most

$$\begin{aligned}
&(1+\varepsilon)^{n_1} \cdot (1+\varepsilon)^{2n_1\cdot\log(1+\varepsilon)} + (1+\varepsilon)^{n_2} + (1+\varepsilon)^{n_3} \cdot (1+\varepsilon)^{2n_3\cdot\log(1+\varepsilon)} \\
&\qquad \leq ((1+\varepsilon)^{n_1} + (1+\varepsilon)^{n_2} + (1+\varepsilon)^{n_3}) \cdot (1+\varepsilon)^{2n_3\cdot\log(1+\varepsilon)} \\
&\qquad \leq (1+\varepsilon)^n \cdot (1+\varepsilon)^{2n_3\cdot\log(1+\varepsilon)} && \text{by condition (a)} \\
&\qquad \leq (1+\varepsilon)^{n+1} \cdot (1+\varepsilon)^{2n\cdot\log(1+\varepsilon)}
\end{aligned}$$

□

We just have to show how to efficiently compute the D_n. Let us start with the case $n = 0$.

Lemma 12. *D_0 can be computed in $\mathcal{O}(|V(G)|^2(|E(G)| + |V(G)|^3)\tilde{c}^2 \operatorname{polylog}(\tilde{w},\tilde{c}))$ time and $\mathcal{O}((|E(G)| + |V(G)|^3)\operatorname{polylog}\tilde{c})$ space.*

Proof. We compute D_0 by starting with the case $d = 0$. For any $u, v \in V(G)$ and $l \in [\tilde{c}]$, by [Dud+22, Theorem 3.8], we can find whether there is a path of cost l from u to v in the graph where we have kept only the edges of null weight in G in $\mathcal{O}((|E(G)| + |V(G)|^3)\tilde{c} \operatorname{polylog} \tilde{c})$ time and $\mathcal{O}((|E(G)| + |V(G)|^3) \operatorname{polylog} \tilde{c})$ space. All in all, this takes $\mathcal{O}(|V(G)|^2(|E(G)| + |V(G)|^3)\tilde{c}^2 \operatorname{polylog} \tilde{c})$ time. Since we want to succeed with probability at least $1/2$, we repeat this computation $\Theta(\log(|V(G)|^2\tilde{w})) = \mathcal{O}(\log \tilde{w})$ times.

For the case $d = 1$, we note that

$$D_0[u, v, l][1] = \bigvee_{\substack{(w,w',1,i)\in E(G) \\ j\in[l-i]}} D_0[u, w, j][0] \wedge D_0[w', v, l - j - i][0]$$

which can be computed in $\mathcal{O}(|V(G)|^2\tilde{c} \cdot |E(G)|\tilde{c}) = \mathcal{O}(|V(G)|^2|E(G)|\tilde{c}^2)$ time. □

To compute D_n efficiently, we compute it modulo p, for a well-chosen $p \in \mathbb{P}$, by applying the Theorem 9. Since the only information we are interested into is whether a certain entry of D_r is nonzero, with good probability, if that entry is nonzero, then it will be nonzero modulo p, ie. p does not divide it, where r is the last n for which we need to compute D_n, ie. $r = \lceil \log_{1+\varepsilon}(x) \rceil$. Before proceeding, we need to show two lemmas that bound the number of D_n we need to compute, and the number of entries we should actually store for each of them (we don't care for entries that are guaranteed to be zero, but the computations must not overflow to apply the Theorem 9, so we must ensure that the nonzero entries are all $\mathcal{O}(x)$).

Lemma 13. $r = \mathcal{O}(\log^2 x)$

Proof. For big enough x, ε is small enough that $\log(1 + \varepsilon) > \varepsilon$.

$$\begin{aligned} r = \lceil \log_{1+\varepsilon} x \rceil &\leq 1 + \frac{\log x}{\log(1+\varepsilon)} \\ &\leq 1 + \frac{\log x}{\varepsilon} && \text{by the aforementioned fact} \\ &= \mathcal{O}(\log^2 x) \end{aligned}$$

□

Lemma 14. *Let* $y = \left\lceil (1+\varepsilon)^r \cdot (1+\varepsilon)^{2r\cdot\log(1+\varepsilon)} \right\rceil$. *We have that* $y = \mathcal{O}(x)$.

Proof. For any ε, $\log(1 + \varepsilon) < 2\varepsilon$, hence

$$\begin{aligned} y &= (1+\varepsilon)^r \cdot (1+\varepsilon)^{2r\cdot\log(1+\varepsilon)} \\ &\leq (1+\varepsilon)^{r\cdot(4\varepsilon+1)} \\ &\leq (1+\varepsilon)^{(1+\log_{1+\varepsilon} x)\cdot(4\varepsilon+1)} \\ &\leq x^{4\varepsilon+1} \cdot (1+\varepsilon)^{4\varepsilon+1} \\ &= \mathcal{O}(x \cdot 2^{4\cdot\log x \cdot \frac{1}{\log x}}) \\ &= \mathcal{O}(x) \end{aligned}$$

□

By Theorem 8, we can find a prime $p = \mathcal{O}(y^2 \operatorname{polylog} y)$, a $t \in \mathbb{N}$ such that $y \leq t = \mathcal{O}(y \operatorname{polylog} y)$ and a $\omega \in \mathbb{F}_p$ such that, for any $N = 2^{\mathcal{O}(y \log y)}$, with probability at least $1/2$, we have:

1. ω is a t-th root of the unity;
2. $p \nmid N$

We will therefore use arrays in $\mathbb{F}_p^t$ to perform our computations. They will not overflow, as $t \geq y$ and by Theorem 14, and $t = \mathcal{O}(y \operatorname{polylog} y) = \mathcal{O}(x \operatorname{polylog} x)$.

Lemma 15. *All values in D_r are bounded by $2^{\mathcal{O}(x \log x)}$.*

Proof. Let f be the tight monotonous function such that, for every $k \in [r]$, $f(k)$ is an upper bound on the values in D_n, and let $g = |V(G)|$. Observe that a single product $A \odot B$ of arrays with entries bounded by $a_{\max}$ (respectively, $b_{\max}$) results in a matrix with entries bounded by $g \cdot y \cdot \tilde{c} \cdot a_{\max} \cdot b_{\max}$. Hence, as values in B_k are in $\{0, 1\}$, we have the bound

$$\begin{aligned} f(k) &\leq \sum_{(k_1,k_2,k_3)\ k-\text{good}} f(k_1) \cdot (yg\tilde{c})^2 \cdot f(k_3) \\ &\leq k^3 (yg\tilde{c})^2 f(k_{\max})^2 \\ &\leq W f(k_{\max}) \end{aligned}$$

where $k_{\max}$ is the largest possible value of k_i that can be part of a k-good triple, and $W = r^2(yg\tilde{c})^2 = \mathcal{O}(\tilde{w}^7)$ as $y = \mathcal{O}(x)$, $\tilde{w} = \Omega(\tilde{c})$ and $\tilde{w} \geq g$.

By definition of a k-good triple, we have

$$\begin{aligned} 2 \cdot (1+\varepsilon)^{k_{\max}-1} &< (1+\varepsilon)^k \\ k_{\max} + \log_{1+\varepsilon} 2 - 1 &< k \end{aligned}$$

Since k and $k_{\max}$ are integers, we even have $k_{\max} \leq k - c$ where $c = \lceil \log_{1+\varepsilon} 2 - 1 \rceil$. By monotonicity of f, we have, for $k = 0, \ldots, r$,

$$f(k) \leq \begin{cases} W \cdot f(k-c)^2 & k \geq c \\ W & k < c \end{cases}$$

By induction, this means that we have, for $k \in [r]$, $f(k) < W^{2^{\left\lceil \frac{k}{c} \right\rceil + 1} - 1}$. As $W = \mathcal{O}(\tilde{w}^7)$, we have

$$f(r) < W^{2^{\left\lceil \frac{k}{c} \right\rceil + 1} - 1} = W^{O(2^{r/c})} < 2^{O(2^{r/c} \log \tilde{w})}$$

Let us finally show that $\frac{r}{c} = \log x + \mathcal{O}(1)$:

$$\begin{aligned}\frac{r}{c} &\leq \frac{\log_{1+\varepsilon}(x)+1}{\log_{1+\varepsilon}(2)-1} = \frac{\frac{\log x}{\log(1+\varepsilon)}+1}{\frac{\log 2}{\log(1+\varepsilon)}-1} \\ &= \frac{\log x + \log(1+\varepsilon)}{1-\log(1+\varepsilon)} \leq \frac{\log x + 2\varepsilon}{1-2\varepsilon} \\ &= \frac{\log^2 x + 2}{\log x - 2} = \log x + \mathcal{O}(1)\end{aligned}$$

So we finally have $f(r) = 2^{\mathcal{O}(x \log x)}$. □

In particular, there is a probability of at least $1/2$ that, if $D_r[s,t,\tilde{c}][\tilde{w}] \neq 0$, $p \nmid D_r[s,t,\tilde{c}][\tilde{w}]$. We can therefore use Theorem 9 to compute $D_r[s,t,\tilde{c}][\tilde{w}] \pmod p$, and output whether it is nonzero.

Let us now estimate the size of the circuit, to check that the complexity is within the stated bounds. We compute $r = \mathcal{O}(\log^2 x)$ arrays D_n, for each of which we process $\mathcal{O}(r^3)$ k-good triples which perform two $\odot$ products each. Every $\odot$ product introduces $\mathcal{O}(|V(G)|^3\tilde{c}^2)$ gates. Hence the total number of gates is $\mathcal{O}(|E(G)| + \tilde{c}^2 \cdot |V(G)|^2 \cdot \operatorname{polylog}(\tilde{w},\tilde{c}))$.

By Theorem 9, we compute $D_r[u,v,l][d]$ in $O(|C|t \operatorname{polylog} p)$ time and $\mathcal{O}(|C| \log p)$ space, where $t = \mathcal{O}(\tilde{w} \operatorname{polylog} \tilde{w})$, $p = \mathcal{O}(y^2 \operatorname{polylog} y)$, $y = \Theta(\tilde{w})$ and $|C| = \mathcal{O}((|E(G) + \tilde{c}^2|V(G)|^3) \log \tilde{w})$. Hence, it takes $\mathcal{O}((|E(G) + \tilde{c}^2|V(G)|^3) \log^2 \tilde{w})$ space and $\mathcal{O}((|E(G) + \tilde{c}^2|V(G)|^3)\tilde{w} \operatorname{polylog} \tilde{w})$ time. □