Pushdown Model Checking above the Cubic Bottleneck

A. R. Balasubramanian*
Max Planck Institute for Software Systems
(MPI-SWS), Kaiserslautern, Germany
Email: bayikudi@mpi-sws.org

Dmitry Chistikov
University of Warwick
Coventry, United Kingdom
Email: d.chistikov@warwick.ac.uk

Rupak Majumdar
Max Planck Institute for Software Systems
(MPI-SWS), Kaiserslautern, Germany
Email: rupak@mpi-sws.org

***Abstract*—It is well known that various problems in program analysis and the verification of recursive programs can be reduced to *pushdown model checking*. In this problem, we are given as input a pushdown automaton (PDA), representing the program, and a description of undesirable behaviors given by an intersection of NFAs, and the problem is to decide if there is a behavior of the PDA that belongs to the set of undesirable behaviors. It is well-known that there is an algorithm for this problem that runs in time $O(n^{2k}|\Sigma| + n^{3k})$, where n is the maximum number of states of the PDA and the NFAs, Σ is the common alphabet of these machines, and $k-1$ is the number of NFAs used to specify the violations. Despite the importance of this problem, no better algorithm is known for it since the 1960s.**

In this paper, we provide an explanation for this lack of progress using the lens of *fine-grained complexity theory*. More precisely, we prove that if the (combinatorial) $3k$-clique hypothesis is true, then there is no algorithm that solves pushdown model checking in time $O((n^{2k}|\Sigma| + n^{3k}))^{1-\varepsilon}$ for any $\varepsilon > 0$. Hence, our result implies that any better algorithm for pushdown model checking than the existing ones would lead to a breakthrough for the $3k$-clique problem. Our lower bound applies even in the case when all the machines are deterministic, and even when the PDA is simply a deterministic one-counter machine. Furthermore, using the same hypothesis, we also show that pushdown model checking over constant-sized alphabets cannot be solved in time faster than $O(n^{3(k-1)-\varepsilon})$ for $\varepsilon > 0$.

Finally, we also investigate the possibility of an $O(N^{3k-\varepsilon})$ time algorithm for pushdown model checking where N is the total bit size of the given input. We formulate a new hypothesis, the 2NPDA(k) hypothesis, that helps explain the lack of $O(N^{3k-\varepsilon})$ time algorithms for pushdown model checking. To corroborate this hypothesis, we show a web of linear-time reductions between the 2NPDA(k) hypothesis, pushdown model checking, and other problems in language theory and automata theory.

I. Introduction

Many problems in program analysis, formal language theory and verification of recursive programs are reducible to *pushdown model-checking*: given as input a PDA (over a constant-sized stack alphabet) and a specification of bad behaviors given as a set of NFAs, the goal is to check if there is a behavior in the language of the PDA that lies in the intersection of the languages of the NFAs. It is well-known that the *model complexity* of the problem, which treats the PDA as the input and the NFAs as constant-sized, is $O(n^3)$ time, where n is the number of states of the given PDA [18], [41], [32], [6], [19], [31]. This cubic algorithm simply does a product construction of the given NFAs with the PDA to get another PDA and then performs reachability analysis on this latter PDA.

When we consider the *combined complexity* of the problem, where both the PDA and the set of NFAs are considered part of the input, the best algorithm runs in time $O(n^{2k}|\Sigma| + n^{3k})$ where n is the maximum number of states in any of the given NFAs and the PDA, Σ is the common input alphabet of all of the machines and $k-1$ is the number of NFAs used to encode the specification. This algorithm uses the algorithm from the previous paragraph. Intuitively, it takes time $O(n^{2k}|\Sigma|)$ to do the product construction of the NFAs with the given PDA to get another PDA with $O(n^k)$ states, and then it takes $O(n^{3k})$ time to perform reachability analysis on this new PDA. Going beyond constant-sized specifications is important in several program analysis applications, such as checking set constraints with regular annotations [27], pointer analysis [33], and model checking for certain security properties [14].

Despite decades of research, no substantially better algorithm is known for either problem. For model complexity, the best current bound is $O(n^3/\log n)$ time [12]. No truly subcubic algorithms are known; this is usually referred to as the "cubic bottleneck" for program analysis [24]. Similarly, for combined complexity, the trivial algorithm above is the best known.

In this paper, we explain this lack of progress through the lens of *fine-grained complexity theory*. This subfield of complexity theory is one of its recent successes: it aims to explain the absence of faster algorithms (than existing ones) for various polynomial-time solvable problems: that is, $(*)$ why a problem can be solved in time $O(n^c)$ but not $O(n^{c-\varepsilon})$ for any $\varepsilon > 0$. By identifying a small number of relevant hypotheses and linking many open questions of the kind $(*)$ with them, fine-grained complexity has provided tight conditional lower bounds for problems in a variety of domains: graph theory, stringology, formal language theory, databases, data structures, and dynamic algorithms [39], [7].

We make the following contributions:

1) We prove that significantly faster algorithms for pushdown model checking would lead to breakthrough al-

* A part of the work was done when the author was at TUM, Germany

gorithms for the k-Clique problem (and thus refute a well-established hypothesis).

2) We show that the hypothesis that pushdown model checking problems require time n^{3k} is closely related (by linear-time inter-reductions) to another natural hypothesis, on the time complexity of languages recognized by two-way nondeterministic pushdown automata with k heads (2NPDA(k), for short).
3) We give a language-theoretic consequence of our reductions: a new construction of hardest languages for the class of languages recognized by 2NPDA(k), for each k.

1. Conditional lower bounds from k-Clique: One of the most important hypotheses used in fine-grained complexity theory is the so-called k-Clique hypothesis ($k \geq 3$): For each $\varepsilon > 0$, there is no algorithm that detects the existence of cliques of size k in graphs on n vertices in time $O(n^{\omega k/3-\varepsilon})$. (Here ω is the matrix multiplication exponent: the infimum of all c such that there is an $O(n^c)$ algorithm to multiply two $n \times n$ matrices.) Closely related to this hypothesis is the *combinatorial* k-Clique hypothesis which asserts that, for each $\varepsilon > 0$, there is no combinatorial[1] algorithm that solves k-Clique in time $O(n^{k-\varepsilon})$. These two hypotheses have been used to show conditional lower bounds for a variety of problems such as context-free language recognition and RNA folding [2], parsing tree adjoining grammars [9], Klee's measure problem [10], maximum-weight box problem in computational geometry [5], etc. In fact, the best conditional lower bounds for the *model complexity* of pushdown model checking, are based on the 3-Clique hypothesis [11], [2], [28], [15]. However, these lower bounds do not imply anything for the *combined complexity* of pushdown model checking.

Our first main result is to provide a tight conditional lower bound on the combined complexity of pushdown model checking problem based on the clique hypotheses. We show that, unless the $3k$-Clique hypothesis (resp. combinatorial $3k$-Clique hypothesis) is false, there is no (combinatorial) algorithm that solves pushdown model checking in time $O((n^{(\omega-1)k}|\Sigma| + n^{\omega k})^{1-\varepsilon})$ (resp. in time $O((n^{2k}|\Sigma|+n^{3k})^{1-\varepsilon})$). Hence, our result proves that, unless the combinatorial $3k$-Clique hypothesis is false, no algorithm can be better than the known algorithm for this problem. In fact, our lower bound applies even to the special case of the problem in which all the $k-1$ NFAs are DFAs, and the PDA is a deterministic one-counter machine.

Our proof (of Theorem 2 in Section III) generalizes the well-known encoding of triangle finding using PDAs, but requires several new ideas, in order to go beyond triangles. First, a large input alphabet (with up to n^k letters) enables us to "name" k-cliques. With the help of a counter (counting up to n^k), we can uniquely store and retrieve a k-clique. We use the counter along with the NFAs to find three k-cliques whose nodes are all neighbors to each other. Our key trick here is to use small automata (n states each) to increment and decrement the counter all the way to these large numbers, as well as check the neighborhood relation between nodes of different k-cliques.

We then turn our attention to the case of pushdown model checking when the input alphabet of all the machines is constant-sized, i.e., when $|\Sigma|$ is a constant. This is an important special case: the complexity of language-theoretic problems is often studied under this assumption. Because $|\Sigma|$ is a constant, we have that $O(n^{2k}|\Sigma| + n^{3k}) = O(n^{3k})$, which is the best known running time in this setting. For this case (handled in Theorem 5 in Section IV), based on the $3k$-Clique hypothesis, we rule out the existence of $O(n^{\omega(k-1)-\varepsilon})$ time algorithms. Similarly, using the combinatorial $3k$-Clique hypothesis, we also prove that $O(n^{3(k-1)-\varepsilon})$ time combinatorial algorithms cannot exist for this problem. Since the alphabet size is fixed, we can no longer use the trick of naming k-cliques. To go beyond this, our construction initially uses linear-sized input and stack alphabets in order to find three k-cliques whose nodes are all neighbors to each other. For this purpose, various specialized gadgets are constructed to store and keep track of (multiple) k-cliques in the stack as well as to check that nodes in different k-cliques are neighbors. We then carefully encode the linear-sized alphabets into binary and convert the original construction into one over constant-sized alphabets, with only a logarithmic blowup in the state space. Note that our results in this regime have a gap of $O(n^{3+(3-\omega)k})$ between the upper and lower bounds in the general case and $O(n^3)$ in the combinatorial case.

We also note that, for the special case of $k = 2$, that is, the intersection non-emptiness problem for the language of 1 PDA and 1 NFA, a conditional lower bound follows from a result in [1]. They consider the special case of the problem in which the PDA is replaced by a straight-line program (a context-free grammar that generates a single word only). The problem is to decide whether a given NFA accepts this *compressed word*. They prove that, unless the combinatorial k-Clique hypothesis is false, there is no $\varepsilon > 0$ for which there exists an algorithm for this problem running in time $O(\min\{pq^3, Nq^2\}^{1-\varepsilon})$, where p is the size of the compressed representation of the word (think the number of states in the PDA), q is the number of states in the NFA, and $N \leq 2^n$. The Nq^2 term matches a decompress-and-solve algorithm which is not available for general PDA. In the regime $p = q = n$, this lower bound is of order n^4.

2. New hypotheses: It is not known whether fast matrix multiplication algorithms can be used for faster pushdown model checking. Standard existing hypotheses appear to be insufficient for explaining the hardness even for $k = 1$ (language non-emptiness of pushdown automata). The best (non-combinatorial) lower bound is $\Omega(n^\omega)$ from k-Clique [2], and it has been shown that the strong exponential-time hypothesis (SETH), perhaps the most well-known hypothesis in fine-grained complexity, cannot be used to beat this bound, unless breakthrough results in circuit complexity appear [15]. Note that the $3k$-Clique hypothesis only asserts the non-existence of algorithms with runtime as a function of the number of nodes and not of the whole input. Intuitively, this cannot help explain

[1]While the notion of *combinatorial algorithms* is not rigorous, these are algorithms that do not use fast matrix multiplication techniques.

whether pushdown model checking admits an algorithm with runtime $O(N^{3k-\varepsilon})$ where N is the overall bit size of the input for the pushdown model checking: the input includes all transitions in the machines, and their number could be quadratic in the number of states.

Thus, new hypotheses may be required to explain the absence of faster algorithms. Indeed, for $k = 1$ the recent NFA acceptance hypothesis [8] gives an n^3 lower bound for pushdown model checking for *dense* PDAs.

For $k > 1$, we introduce (in Section V) a new 2NPDA(k) *hypothesis*. It asserts that there is no $\varepsilon > 0$ for which some algorithm running in time $O(|w|^{3k-\varepsilon})$ can decide if a given word w is accepted by a (fixed) two-way non-deterministic pushdown automaton with k heads (2NPDA(k)). Intuitively, a 2NPDA(k) is a machine (see [23], [26]) which has access to a stack and its input is written on a read-only input tape. This machine has k heads on the input tape using which it can query the letters of k positions on the input tape. Based on this query, it can update its state, the positions of these k heads and also its stack content. It is a folklore result that there is an algorithm for the 2NPDA(k) acceptance problem that runs in time $O(|w|^{3k})$ for any fixed 2NPDA(k). (For example, Rytter [35] refers to Aho, Hopcroft, and Ullman [3], even though only 2NPDA with $k = 1$ head are considered there.) However, no faster algorithm is known for this problem. Based on this lack of progress, we introduce this hypothesis as a generalization of the 2NPDA hypothesis ($k = 1$) introduced by Neal [30] and Heintze and McAllester [24]. Furthermore, if $k \geq 2$, 2NPDA(k) language recognition is not known to admit even $O(|w|^{3k}/\log|w|)$ algorithms, unlike for $k = 1$ [12], [37]. To the best of our knowledge, algorithms with this complexity are only known for the special case of loop-free automata [37]; see also related results in [35], [36].

As a way to strengthen the believability of this hypothesis, we provide a web of linear-time reductions between the 2NPDA(k) language recognition problem and a variety of other problems in language theory and program analysis, one of which is the pushdown model checking problem. So, under the 2NPDA(k) hypothesis, we show that there is no algorithm running in time $O(N^{3k-\varepsilon})$ for pushdown model checking. Hence, this gives rise to a hierarchy of program analysis problems, one for each $k > 1$, that go beyond the famous cubic bottleneck that is established in the literature.

3. Consequences and applications: The lens of fine-grained complexity provides some purely language-theoretic consequences. Using our chain of linear-time reductions between 2NPDA(k) recognition and pushdown model checking, we get that, for each $k \geq 1$, there is a *hardest* 2NPDA(k) language, that is, a fixed language $L_0^{(k)}$ recognized by a 2NPDA with k heads such that for every 2NPDA(k) language L there is a homomorphism h such that $w \in L$ iff $h(w) \in L_0^{(k)}$, for all non-empty words w. Previous constructions of hardest languages used language-theoretic constructions [29], [34].

II. Preliminaries

A. PDA and NFAs

A pushdown automaton (PDA) consists of a finite set of control states and a stack into which it can push/pop elements. Initially, the PDA begins in some designated initial control state and reads the input word w one letter at a time. As it reads each letter, its transition relation allows it to move from one control state to another whilst pushing/popping elements from its stack. At the end of reading w, if the machine is in one of a designated set of final states and its stack content is empty, then it is said to accept w. The language of the machine is the set of all words that it accepts.

Formally, a PDA is a tuple $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ where Q is a finite set of control states, Σ is the input alphabet, Γ is the stack alphabet, $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \times Q \times \Gamma^*$ is the transition relation, q_0 is the initial state and F is a set of final states. A transition of the form (p, a, α, q, β) dictates that, in order for this transition to be used, the PDA must be at state p, read the letter a (or read no letter if $a = \varepsilon$) and then it must pop α from the stack, move to state q and push β into the stack. Note that β could be ε, which means that the net effect is simply popping α from the stack. Similarly, β could also be of the form $\alpha\zeta$ for some letter ζ, which means that the net effect is simply pushing ζ into the stack. We assume that Γ contains a designated "end of stack" symbol Z_0 such that no transition of P replaces Z_0 on the stack with a different symbol, or pushes Z_0 on the stack when the top of the stack is not Z_0, i.e., for any transition (p, a, α, q, β), β contains Z_0 iff $\alpha = Z_0$ and $\beta = Z_0\gamma$ for some $\gamma \in (\Gamma \setminus Z_0)^*$. Throughout the paper, we will assume that the stack alphabet Γ is of constant size, i.e., $|\Gamma| = O(1)$ unless specifically stated otherwise.

A *configuration* of the PDA is a pair (p, γ) where p is a state and $\gamma \in \Gamma^*$ is the stack content. For any transition $t = (p, a, \alpha, q, \beta)$, there is a step from a configuration of the form $(p, \gamma\alpha)$ to $(q, \gamma\beta)$, which represents the changes made to the stack as dictated by t. We will denote this step by $(p, \gamma\alpha) \xrightarrow{t} (q, \gamma\beta)$ or simply $(p, \gamma\alpha) \xrightarrow{a} (q, \gamma\beta)$ when only the component $a \in \Sigma \cup \{\varepsilon\}$ of the transition t is important.

A transition (p, a, α, q, β) is called an ε transition if $a = \varepsilon$. We say that a configuration (q, γ) can reach a configuration (q', γ') by using ε transitions if there are configurations $(q_0, \gamma_0), (q_1, \gamma_1), \ldots, (q_k, \gamma_k)$ such that $(q, \gamma) = (q_0, \gamma_0) \xrightarrow{\varepsilon} (q_1, \gamma_1) \xrightarrow{\varepsilon} (q_2, \gamma_2) \xrightarrow{\varepsilon} \ldots (q_k, \gamma_k) = (q', \gamma')$.

The initial configuration is (q_0, Z_0). A run of the PDA on a word $w = w_1 w_2 \ldots w_n \in \Sigma^*$ is a sequence of configurations of the form $(q_0, \gamma_0), (q_1, \gamma_1), \ldots, (q_n, \gamma_n)$ such that (q_0, γ_0) is the initial configuration and, for each $i \geq 0$, there exist configurations $(q_i', \gamma_i'), (q_i'', \gamma_i'')$ such that (q_i, γ_i) can reach (q_i', γ_i') by ε transitions, $(q_i', \gamma_i') \xrightarrow{w_i} (q_i'', \gamma_i'')$ and (q_i'', γ_i'') can reach (q_{i+1}, γ_{i+1}) by ε transitions. A run is said to be accepting if γ_n is empty and $q_n \in F$. The *language* of a PDA P, denoted by $\mathcal{L}(P)$, is the set of all words that it accepts, i.e., the set of all words on which it has an accepting run. PDA are known to accept exactly the set of context-free languages.

A one-counter automaton (OCA) is a PDA with a stack alphabet containing only one letter (say α) apart from Z_0. Note that the content of a stack is then uniquely determined by the number of times α appears in it. Hence, in this case the stack can be thought of as a counter, where pushing/popping α corresponds to incrementing/decrementing the counter, respectively; and popping Z_0 corresponds to testing whether the counter is zero. Hence, the accepting condition for a run in an OCA is that the state at the end is an accepting state and the counter has reached the value 0.

A PDA is deterministic if its transition relation is a partial function, i.e., δ is of the form $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightharpoonup Q \times \Gamma^*$ and, moreover, for each $q \in Q$, either $\delta(q, a, \alpha)$ is defined for all $a \in \Sigma$, $\alpha \in \Gamma$ and $\delta(q, \varepsilon, \alpha)$ is undefined for all $\alpha \in \Gamma$, or $\delta(q, a, \alpha)$ is undefined for all $a \in \Sigma$, $\alpha \in \Gamma$ but $\delta(q, \varepsilon, \alpha)$ is defined for all $\alpha \in \Gamma$. A deterministic PDA (resp. deterministic OCA) will be succinctly referred to as DPDA (resp. DOCA).

An NFA is a PDA in which there are no ε transitions and no stack operations are performed, i.e., no push or pop happens to the stack. (Formally, $\alpha = \beta = Z_0$ in all transitions (p, a, α, q, β).) A DFA is an NFA for which the transition relation is actually a function. NFAs and DFAs accept exactly the set of regular languages.

B. Intersection Non-Emptiness problems

This paper focuses on the following class of problems, one for *each fixed* $k \geq 1$:

$\mathbf{PDA \cap NFA^{k-1}}$ Intersection Non-Emptiness

Fix: Stack alphabet Γ.
Input: PDA P and $k-1$ NFAs $A_1, \ldots, A_{k-1}$, all over a common input alphabet Σ.
Decide: Is the intersection $\mathcal{L}(P) \cap \mathcal{L}(A_1) \cap \ldots \cap \mathcal{L}(A_{k-1})$ non-empty?

When $k = 1$, this is simply checking if the given PDA P has a non-empty language, which is the well-known PDA non-emptiness problem.

In this paper, we let the stack alphabet Γ be fixed (as it is in applications related to program analysis). In contrast, the common input alphabet of the automata is provided as part of the input, as it forms part of the description of the PDA and each NFA. In applications, the input alphabet needs to be rich enough to specify possible actions or events in the system. In theory, it is often instructive to consider constant-sized alphabets too, e.g., $|\Sigma| = 2$. Furthermore, there are applications related to PDAs in which the underlying language is fixed, which automatically fixes the alphabet as well. For instance, for the CFL Reachability problem (which we generalize and study as the CFL k-Intersection Reachability problem in Section V-B in our paper), the underlying context-free language (and hence the input alphabet) is fixed.

In general, the model checking problem may use multiple NFAs to encode a specification. We give a small example from the domain of model checking for security properties [14], [13]. A program is modelled as a PDA, and security properties as a set of NFAs. For example, a security property is "a program should drop privileges from all its user IDs before calling certain system functions." This property is expressed as multiple NFAs: one NFA tracking if certain system calls have been made, and the others tracking which user IDs have root privilege. The common alphabet consists of system calls, which are executed by the program (PDA) and also cause the property NFAs to change state. For example, a call to drop root privilege from a user ID moves the corresponding NFA to a state in which that user ID does not have root privilege.

Furthermore, sometimes an alternative way of encoding a single big specification (an NFA of size n^k for some k) might be to decompose it into a product of multiple smaller NFAs (intersection of k NFAs, each of size n). There are NFAs that cannot be decomposed in this manner, but, a priori, it might have been possible that if the specification NFA has a nice structure, it could be decomposed into that form and the problem could have been solved faster than the general case. Our results in the next section (Theorem 2) show that even this restricted setting is as difficult as the original version.

If the number of NFAs, k, is unbounded (not fixed), then intersection non-emptiness becomes complete for EXPTIME. (Indeed, a $T(n)$-time Turing machine can be simulated by an auxiliary pushdown automaton (AuxPDA) with $O(\log T(n))$ bits of storage [16]. Language recognition for such an AuxPDA is reducible to the intersection non-emptiness of a usual PDA with $O(\log T(n))$ DFAs, each responsible for one cell of the storage.) Shortest words in the intersection may in this scenario be doubly exponentially long [4].

The main contributions of this paper are to provide conditional lower bounds for the $\mathrm{PDA} \cap \mathrm{NFA}^{k-1}$ intersection non-emptiness problem under different settings, some of which match the known upper bounds, suggesting that no improvement over the known algorithm is possible. These conditional lower bounds are based on popular hypotheses from the field of fine-grained complexity theory. We now proceed to describe these hypotheses and then our contributions.

C. The k-Clique hypothesis

This is one of the central hypotheses in fine-grained complexity theory. Formally, the k-Clique hypothesis states that, for each $\varepsilon > 0$, there is no algorithm that, given a graph G on some n vertices, correctly decides if G has a clique of size k in time $O(n^{\omega k/3 - \varepsilon})$. A similar hypothesis is the so-called combinatorial k-Clique hypothesis, which states that there is no combinatorial algorithm that solves k-Clique in time $O(n^{k-\varepsilon})$ for any $\varepsilon > 0$. As mentioned in the Introduction, while the notion of combinatorial algorithms is not rigorous, these are roughly taken to be algorithms that do not use fast matrix multiplication techniques.

III. Lower Bounds for $\mathrm{PDA} \cap \mathrm{NFA}^{k-1}$ Intersection Non-Emptiness

Before we present our main result, which is a (conditional) lower bound for $\mathrm{PDA} \cap \mathrm{NFA}^{k-1}$ Intersection Non-Emptiness, let us first recall the known upper bounds for this problem.

Theorem 1 (Upper Bounds)**.** *The* $\mathrm{PDA} \cap \mathrm{NFA}^{k-1}$ *intersection non-emptiness problem over an input alphabet* Σ *and with* n *the maximum number of states among all the given machines can be solved in time*

- $O(n^{2k}|\Sigma| + n^{\omega k})$ *if the given PDA is an OCA;*
- $O(n^{2k}|\Sigma| + n^{3k})$ *if the given PDA is not an OCA or if only combinatorial algorithms are allowed.*

Proof. Let P be the given PDA and let $A_1, \ldots, A_{k-1}$ be the given NFAs. The proof of the theorem follows from two observations: First, PDA non-emptiness can be solved in $O(n^3)$ time [6] and furthermore, if the PDA is an OCA, then it can be solved in $O(n^\omega)$ time [22]. Second, for any PDA P and any NFA A, in $O(n^4|\Sigma|)$ time, we can construct a PDA with $O(n^2)$ states and the same stack alphabet as P, such that this new PDA recognizes $\mathcal{L}(P) \cap \mathcal{L}(A)$. This is the usual Cartesian product construction between a PDA and an NFA (see, e.g., Hopcroft, Motwani, and Ullman's textbook [25, Section 7.3.4]). Hence, by repeatedly doing the product construction, in time $O(n^{2k}|\Sigma|)$, we can construct a PDA with $O(n^k)$ states having the same stack alphabet as P which recognizes $\mathcal{L}(P) \cap \mathcal{L}(A_1) \cap \ldots \cap \mathcal{L}(A_{k-1})$. Then, we can perform a non-emptiness check on this PDA in time $O(n^{3k})$ or in time $O(n^{\omega k})$ if it is an OCA. □

The above algorithm has been known for almost 50 years, and no polynomial improvements have been made for this problem. We provide a conditional lower bound for this problem, showcasing the difficulty of any improvement.

Theorem 2 (Lower Bounds)**.** *If the* $3k$*-Clique hypothesis is true, the* $\mathrm{PDA} \cap \mathrm{NFA}^{k-1}$ *intersection non-emptiness problem over an input alphabet* Σ *with* n *the maximum number of states among all the given machines cannot be solved in time*

- $O((n^{(\omega-1)k}|\Sigma| + n^{\omega k})^{1-\varepsilon})$ *for any* $\varepsilon > 0$*;*
- $O((n^{2k}|\Sigma| + n^{3k})^{1-\varepsilon})$ *for any* $\varepsilon > 0$*, if only combinatorial algorithms are allowed.*

Both lower bounds already hold when the given PDA is a DOCA and all the NFAs are DFAs.

This theorem provides a tight lower bound for combinatorial algorithms: no improvements using such algorithms are possible unless the combinatorial $3k$-Clique hypothesis is false.

Remark 3. *Our lower bounds admit multivariate counterparts, where the number of states of PDA and NFAs has different orders of magnitude. Suppose in the input to the problem the PDA has at most* p *states and each NFA has at most* q *states. The existing algorithm from Theorem 1 delivers the upper bound* $O(s^2|\Sigma| + s^3)$*, where* $s = pq^{k-1}$ *is the number of states in the product automaton. Thus, our lower bound implies that there is no combinatorial algorithm for this problem running in time*

$$O\big(((pq^{k-1})^2|\Sigma| + (pq^{k-1})^3)^{1-\varepsilon}\big)$$

for any $\varepsilon > 0$*. Indeed, even in the regime* $p = q = n$ *such an algorithm would contradict Theorem 2. Similar conclusions can be drawn from other lower bounds that we prove in this paper. We will henceforth not make them explicit.*

A. Proof idea of Theorem 2

The formal proof of the theorem is given in Appendix A. Here we give the main ideas and intuitions behind the proof.

Recall: Triangle finding: The main idea behind this proof is a generalization of the idea used to detect triangles, i.e., 3-cliques in a graph by means of an OCA. Let us first recall this idea. Let G be some graph and without loss of generality, let the nodes of this graph be $\{0, 1, \cdots, n-1\}$. Furthermore, we can assume that the graph contains no self-loops, since removing self-loops does not destroy the property of having a triangle. We can check for the existence of a triangle in this graph by using the following DOCA $\mathcal{M}$: First $\mathcal{M}$ reads a letter corresponding to some node a, moves to a state labelled by $(a, 0)$ and increments the counter by a. Then it reads a letter corresponding to some node b and moves from $(a, 0)$ to $(b, 1)$ only if b is a neighbor of a. (Checking that b is a neighbor of a can be hard-coded into the transitions of $\mathcal{M}$.) Then, from $(b, 1)$, it reads a letter corresponding to some node c and moves from $(b, 1)$ to $(c, 2)$ only if c is a neighbor of b. Then, it reads some letter d and moves from $(c, 2)$ to $(d, 3)$ only if d is a neighbor of c. Finally, from $(d, 3)$, it reads any letter, decrementing the counter by d. If the counter is 0 at the end, then $\mathcal{M}$ accepts, otherwise it rejects.

Note that any triangle $\{a, b, c\}$ in the graph can be converted into an accepting path in $\mathcal{M}$ by inputting the sequence a, b, c, a. On the other hand, any accepting path in $\mathcal{M}$ is of the form: $(a, 0), (b, 1), (c, 2), (d, 3)$, where $(a, b), (b, c), (c, d)$ are edges in G. Further the only updates that happen along the way to the counter are an increment by a at the beginning, and a decrement by d at the end. Recall that for a run to be accepted, the counter must be 0 at the end, and so it follows that $d = a$ and so $\{a, b, c\}$ is a triangle in G. Hence the language of $\mathcal{M}$ is non-empty if and only if G has a triangle.

We will now expand upon this idea and prove the lower bound for the general case. More precisely, given a graph G on n nodes (without self-loops), we will use a DOCA $\mathcal{M}_0$, $k-1$ many DFAs $\mathcal{M}_1, \cdots, \mathcal{M}_{k-1}$ (where $\mathcal{M}_0, \mathcal{M}_1, \cdots, \mathcal{M}_{k-1}$ will all have $O(n)$ states) and an input alphabet of size $O(n^k)$ to detect $3k$ cliques in graphs.

Finding three k*-cliques:* We can think of a $3k$-clique as 3 different k-cliques C_1, C_2, C_3 of size k each such that every node in each C_i is connected to every node in each C_j for $i \neq j$, i.e., $C_i \cup C_j$ for $i \neq j$ is a $2k$-clique. Our construction attempts to find such k-cliques in the following manner: First, it finds a k-tuple of nodes $C_1 := (v_0, \ldots, v_{k-1})$, stores each node of C_1 in one of the machines and also stores C_1 as a whole in the counter of the OCA *in a unique way*. Some questions arise at this point.

Q: How can we uniquely store a tuple of nodes of size k as a single number?

A: We map each k-tuple of nodes $(v_0, \ldots, v_{k-1})$ to the number $n^{k-1}v_{k-1} + n^{k-2}v_{k-2} + \cdots + v_0$. Note that no two tuples are mapped to the same number. ◁

Q: The above representation can lead to numbers as high as $n^k - 1$. However, each machine can only have $O(n)$ states. How can we increment a counter to that high a value?

A: For this purpose, we construct gadgets that serve as a base-n counter over k digits. These gadgets will have two important properties: First, each gadget G_i will have exactly n states, one for each number from 0 to $n-1$, with transitions which are either self-loops or only taking place between successor states (modulo n). The second property is that, for each G_i, a transition between successor states in G_i can occur if and only if a run of length n traversing all the states occurs in G_{i-1}. This means that if we execute the gadgets $G_0, G_1, \ldots, G_i$ and at some point G_i stops at the state v_i and $G_{i-1}, \ldots, G_0$ all stop at the state 0, then we have executed a path of length exactly $n^i v_i$.

Formally, input letters of each gadget G_i are $\{\#_0, \ldots, \#_{k-1}\}$. Each state $j \in \{0, 1, \ldots, n-1\}$ in G_i will stay at j if it reads any letter from $\#_0, \ldots, \#_{i-1}$, move to $j+1$ if $j < n-1$ and it reads $\#_i$ and finally move to 0 if $j = n-1$ and it reads any letter from $\#_{i+1}, \ldots, \#_{k-1}$. Each gadget initially has start state 0.

Suppose we now take k copies of our gadgets and execute the first copy of $G_0, G_1, \ldots, G_{k-1}$ until G_{k-1} reaches some state v_{k-1} and all the other gadgets reach 0. At this point, suppose we stop the first copy by reading a special letter, which can only be read when $G_0, \ldots, G_{k-2}$ are at the state 0. (If read from any other state, they will move to a rejecting sink state.) After reading this special letter, we "freeze" the value v_{k-1} in G_{k-1} (i.e., we will always remember this value in the remaining copies of G_{k-1}) and then execute the second copy of the gadgets $G_0, G_1, \ldots, G_{k-2}$. Then, once the second copy of G_{k-2} reaches some state v_{k-2} and all the other gadgets reach 0, we stop the second copy by reading another special letter, freeze the value v_{k-2} in G_{k-2} and move on to the third copies of $G_0, G_1, \ldots, G_{k-3}$ and so on. Hence, once we have finished executing all the k copies, we must have a run of length exactly $n^{k-1}v_{k-1} + n^{k-2}v_{k-2} + \cdots + v_0$ for some $v_{k-1}, \ldots, v_0$ that are stored in the last copies of the gadgets. So, if we simply incremented the counter every time a step is executed, at the end, the counter value will be exactly $n^{k-1}v_{k-1} + n^{k-2}v_{k-2} + \cdots + v_0$. ◁

Having found this tuple $C_1 := (v_0, \ldots, v_{k-1})$, we now check that it is a k-clique of the graph G.

Q: How can we verify that C_1 is indeed a k-clique of G?

A: Recall that when the collection is found, each machine stores one node of C_1. Furthermore, we are allowed to have an input alphabet of size n^k. Hence, for each k-clique we will have a letter in our input alphabet. Then, we force the i^{th} machine (which stores v_i) to read one of these letters from its current state only if the i^{th} node in this letter is v_i. (If it reads some other letter, it will move to a rejecting sink state.) This ensures that if all the machines successfully read some letter, then C_1 is a k-clique. ◁

Having now found a k-clique C_1, we now find another k-clique C_2 such that $C_1 \cup C_2$ is a $2k$-clique and store each node of C_2 in one of the machines.

Q: How can we find such a C_2?

A: Each node of C_1 is stored in some machine. Now, we force the i^{th} machine to read a letter corresponding to some k-clique C_2 only if the node stored in the i^{th} machine is a neighbor of every node in C_2 and if it is indeed the case, then the i^{th} machine forgets its current node and starts storing the i^{th} node of C_2. Note that since no self-loops are present in G, if all the machines successfully read the same letter corresponding to some k-clique C_2, then we are guaranteed that $C_1 \cap C_2 = \emptyset$, $C_1 \cup C_2$ is a $2k$-clique and all the machines now store nodes of C_2. ◁

Having now found a k-clique C_2, we now find another k-clique C_3 such that $C_2 \cup C_3$ is a $2k$-clique by the same method as before. Then, we once again find another k-clique C_1' such that $C_3 \cup C_1'$ is a $2k$-clique. Now, if we verify that $C_1 = C_1'$, then we have successfully found a $3k$-clique.

Q: How can we verify that $C_1 = C_1'$?

A: Recall that after having incremented the counter to uniquely store C_1, we have not modified it at all. Hence, the current value of the counter is the encoding of C_1. So, if we decrement the counter by the encoding of C_1' and accept if the counter is zero, then we could verify that $C_1 = C_1'$. ◁

Q: How do we decrement the counter by the value corresponding to C_1'?

A: Recall that to increment the counter to the value of C_1, we constructed k copies of the gadgets $G_0, \ldots, G_{k-1}$ and executed them till the states in the last copy of each gadget G_i stored the i^{th} node of C_1. Hence, if we execute the copies of these gadgets *in reverse*, beginning with the i^{th} machine storing the i^{th} node of C_1' and decrement the counter every time we take a step, then this would decrement the counter by exactly the value corresponding to C_1'. ◁

This completes all the main ideas behind the proof of the theorem. All the machines described above have $O(n)$ states (where the constant depends on k, but recall that k is also a fixed constant) and the alphabet size is $O(n^k)$. Furthermore, each of these machines can be constructed in time $O(n^{k+1})$.

Now, suppose $\text{PDA} \cap \text{NFA}^{k-1}$ intersection non-emptiness can be solved in time $O((n^{(\omega-1)k}|\Sigma| + n^{\omega k})^{1-\varepsilon})$ for some $\varepsilon > 0$. Then, we can solve the $3k$-Clique problem in time $O(n^{\omega k - \varepsilon})$ as follows: Given a graph G, first construct the machines $\mathcal{M}_0, \ldots, \mathcal{M}_{k-1}$ described above in time $O(n^{k+1})$ and then solve the $\text{PDA} \cap \text{NFA}^{k-1}$ intersection non-emptiness problem on this instance. The overall time taken is $O(|G| + n^{k+1}) + O((n^{(\omega-1)k}|\Sigma| + n^{\omega k})^{1-\varepsilon}) = O(n^{\omega k(1-\varepsilon)})$, which contradicts the $3k$-Clique hypothesis. Similarly, any $O((n^{2k}|\Sigma| + n^{3k})^{1-\varepsilon})$ time algorithm for $\text{PDA} \cap \text{NFA}^{k-1}$ intersection non-emptiness would contradict the combinatorial $3k$-Clique hypothesis.

IV. $\text{PDA} \cap \text{NFA}^{k-1}$ Intersection Non-Emptiness - The Case of Constant Alphabets

The reduction in the last section used an alphabet that grew with the input graph. In this section, we turn our attention to instances of the $\text{PDA} \cap \text{NFA}^{k-1}$ non-emptiness

problem where the input alphabet of the underlying machines is fixed. More precisely, we fix an alphabet Σ in advance and only consider instances of the $\mathrm{PDA} \cap \mathrm{NFA}^{k-1}$ non-emptiness problem over this fixed alphabet Σ. As a consequence of Theorem 1, we get the following upper bound.

Corollary 4. *The $\mathrm{PDA} \cap \mathrm{NFA}^{k-1}$ non-emptiness problem over a fixed input alphabet with n the maximum number of states among all the given machines can be solved in time*

- *$n^{\omega k}$ if the given PDA is an OCA*
- *n^{3k} if the given PDA is not an OCA or if only combinatorial algorithms are allowed.*

No polynomial improvement over this algorithm is known in the literature. We now provide a lower bound that is almost tight in the case of combinatorial algorithms and suggests that big improvements over this algorithm are unlikely.

Theorem 5. *If the $3(k-1)$-Clique hypothesis is true, the $\mathrm{PDA} \cap \mathrm{NFA}^{k-1}$ problem over a fixed input alphabet with n the maximum number of states among all the given machines cannot be solved in time*

- *$O(n^{\omega(k-1)-\varepsilon})$ for any $\varepsilon > 0$.*
- *$O(n^{3(k-1)-\varepsilon})$ for any $\varepsilon > 0$, if only combinatorial algorithms are allowed.*

Both lower bounds already hold when the given PDA is a DPDA.

We note that the lower bound here is a factor of $n^{3+(3-\omega)k}$ away in the general case and a factor of n^3 away in the combinatorial case from the respective upper bounds.

A. Proof idea of Theorem 5

The formal proof of the theorem is given in Appendix B. We now give the main ideas and intuitions behind the proof.

Let us fix a number $3(k-1)$. Let G be some graph (without self-loops) over nodes $\{0, \ldots, n-1\}$. We will construct a DPDA $\mathcal{M}_0$ and $k-1$ many NFAs $\mathcal{M}_1, \ldots, \mathcal{M}_{k-1}$ such that G has a $3(k-1)$-clique if and only if there is a word w in the intersection of the languages of all $\mathcal{M}_i$. For the purposes of presentation, we will first describe this construction with *linear-sized* input and stack alphabets, i.e., the size of the input and stack alphabets will not be a constant. Then, by a careful analysis of the construction, we will convert it into one over constant-sized input and stack alphabets with a logarithmic blowup in the state space. We now proceed to the construction with the linear-sized alphabets.

The very high-level idea behind these machines is similar to the construction that we saw in Theorem 2. Intuitively, the machines will first find a collection of $k-1$ vertices C_1, check that C_1 is a $(k-1)$-clique, then find a collection of $k-1$ vertices C_2 and check that every node in C_1 is connected to every node in C_2. Then, they will check that C_2 is a $(k-1)$-clique, find a collection of $k-1$ vertices C_3 and check that every node in C_2 is connected to every node in C_3. Then they will do a similar procedure with C_3 and find another collection C_1'. Then they will finally check that $C_1' = C_1$, which will prove that $C_1 \cup C_2 \cup C_3$ is a $3(k-1)$-clique. We stress that while the high-level idea behind this construction and the one given in Theorem 2 are the same, the actual implementation details vary significantly. In particular, new ideas are needed in order to store the cliques into the stack and circumvent the large alphabet size of the construction from Theorem 2. We will present these ideas now.

At any given point, each machine will store either a node of the graph G in its state or store a special symbol $\Diamond$ denoting that it is not storing any node. The intuition behind the NFAs $\mathcal{M}_1, \ldots, \mathcal{M}_{k-1}$ is that, at each point, each $\mathcal{M}_i$ will store one of the nodes of the collection of $k-1$ nodes that is currently being examined, i.e., one of the nodes in either C_1, C_2, C_3 or C_1'. The intuition behind the PDA $\mathcal{M}_0$ is two-fold. First, the stack of $\mathcal{M}_0$ will help store the cliques C_1, C_2, C_3, C_1' of G, along with some store other information. Second, the states of $\mathcal{M}_0$ should be thought of as a "scratchpad", in that it will help store some auxiliary information that will be needed for the NFAs.

We will encode each node $i \in \{0, \ldots, n-1\}$ by itself, i.e., the input and the stack alphabets will have as letters all the numbers between 0 and $n-1$. In addition to these letters, the input alphabet will also have letters of the form $\{\overline{i} : 0 \leq i \leq n-1\}$. The intuition is that, whenever i is read as an input letter, either the stack does not change or i will be pushed onto the stack. Similarly, whenever $\overline{i}$ is read as an input letter, i will be popped from the stack. Furthermore, the input alphabet will have $\#$ and @ as two other additional letters.

We now describe the construction of the machines. We recall that each machine will store either a node of the graph or $\Diamond$ in its state at all times. The machines will work together in three different parts and each part will itself comprise three different sub-parts. We begin by describing the first part, whose goal is to check if a collection of nodes C_1 is a $(k-1)$-clique and if so, find another collection C_2 such that every node in C_1 is connected to every node in C_2. This is done in three sub-parts.

Part 1, Sub-Part I: The Setup: In the first sub-part, we will store $k-1$ nodes in the stack of $\mathcal{M}_0$ in a specific manner. The PDA $\mathcal{M}_0$ begins this sub-part by remembering $\Diamond$ in its initial state. Further each NFA $\mathcal{M}_i$ begins by remembering some node x_i in its state. (This can be thought of as non-deterministically selecting a state for each $\mathcal{M}_i$ with some node x_i stored in that state; later on we will see how this restriction can be removed). The goal of the first sub-part is to setup the stack in a specific way so that each x_i is pushed into the stack exactly $i-1$ times. This is done in the following manner. Since x_2 is stored in the state of $\mathcal{M}_2$, we can force the input letter that is read at this point to be x_2. Indeed, we only have to create a copy of the current state and have exactly one transition which leads from the original state to the copy by reading x_2. This will ensure that x_2 is the only possible input letter that could be read at this point. We also ensure that upon reading x_2, the PDA $\mathcal{M}_0$ pushes it onto the stack. Similarly, since x_3 is stored in the state of $\mathcal{M}_3$, we can force the next *two* input letters to be x_3, by adding two copies of the current state of $\mathcal{M}_3$ and appropriate transitions. We can

also ensure that the letter x_3 is pushed into the stack twice. In this way, we can ensure that each letter x_i is pushed into the stack $i-1$ times. Once this sub-part is done, each $\mathcal{M}_i$ will store the node x_i in its state, $\mathcal{M}_0$ will store $\Diamond$ in its state, and the stack of $\mathcal{M}_0$ (from the top) will contain $k-2$ many copies of x_{k-1}, $k-3$ copies of x_{k-2} and so on all the way till one copy of x_2. This completes the first sub-part.

Part 1, Sub-Part II: The Check: In the second sub-part, we will use the special structure of the nodes $x_1, \ldots, x_{k-1}$ that are stored in the stack to check that these nodes indeed form a $(k-1)$-clique. This is done in the following manner. Note that, at the end of the first sub-part, the node x_1 is stored in $\mathcal{M}_1$ and the node x_{k-1} is stored at the top of the stack. Using just this information, we will devise a gadget that checks that x_1 is a neighbor of x_{k-1} in the following way. Since x_1 is stored in $\mathcal{M}_1$, we can force the next input letter to be x_1, similar to how we did it in the first sub-part. Upon reading this input letter, we can make $\mathcal{M}_0$ remember it in its state. (Hence at this point, both $\mathcal{M}_0$ and $\mathcal{M}_1$ remember x_1 and x_{k-1} is at the top of the stack.) Now, we force the next input letter to be the special letter $\#$. Upon reading $\#$, $\mathcal{M}_1$ will move to some neighbor x' of x_1. (At this point, $\mathcal{M}_0$ stores x_1, $\mathcal{M}_1$ stores x' and the top of the stack stores x_{k-1}.) We can now force the next input letter to be $\overline{x'}$ and we also force $\mathcal{M}_0$ to pop x' from the top of the stack. For both these things to simultaneously happen, it must be the case that $x' = x_{k-1}$ and hence that x_1 and x_{k-1} are neighbors. (If this successfully happens, then at this point, $\mathcal{M}_0$ stores x_1, $\mathcal{M}_1$ stores $x' = x_{k-1}$ and the stack now contains one fewer x_{k-1} at the top.) Now, we force the input letter to be $\#$, upon reading which $\mathcal{M}_1$ moves to a state remembering $\Diamond$. From there, because $\mathcal{M}_0$ remembers x_1, we can force the input letter to be x_1, upon reading which $\mathcal{M}_1$ will move back to storing x_1 and $\mathcal{M}_0$ will move back to storing $\Diamond$. In this way, we have checked that x_1 and x_{k-1} are neighbors and the only information that we lost along the way was a copy of x_{k-1} at the top.

By the structure of the first sub-part, it follows that we now have $k-3$ more copies of x_{k-1} remaining in the stack of $\mathcal{M}_0$. Hence, we can now reformulate the same gadget from the above paragraph to check that x_2 and x_{k-1} are neighbors, x_3 and x_{k-1} are neighbors and so on.

Note that, after exhausting all the copies of x_{k-1} from the stack, we are left with $k-3$ copies of x_{k-2}. This is then sufficient to check that x_{k-2} is a neighbor of $x_1, x_2, \ldots, x_{k-3}$. Then we do the same check for x_{k-3}, x_{k-4} and so on all the way till x_2. This ensures that $x_1, \ldots, x_{k-1}$ is a $(k-1)$-clique. Note that, at the end of this computation, each $\mathcal{M}_i$ stores x_i and $\mathcal{M}_0$ stores $\Diamond$. This completes the second sub-part.

Part 1, Sub-Part III: The Exploration: In the third sub-part, we will find $k-1$ more nodes $y_1, \ldots, y_{k-1}$ and check that each x_i is connected with each y_j. This is done in the following manner. Initially, we read some node y_1, store it in the state of $\mathcal{M}_0$ and push it onto the stack. Since y_1 is now stored in $\mathcal{M}_0$, we can ensure that the next k input letters are all y_1 and also that all these k input letters are pushed onto the stack. Then, by using the gadget from the second sub-part, we can check that $x_1, x_2, \ldots, x_{k-1}$ are all neighbors of y_1. By construction of this gadget, at the end of this check, each $\mathcal{M}_i$ will still store x_i, $\mathcal{M}_0$ will store $\Diamond$ and the stack will contain one copy of y_1 (since we pushed k many copies of y_1 and only popped $k-1$ many copies). We now repeat what we did before to read another node y_2 k times and ensure that $x_1, \ldots, x_{k-1}$ are all neighbors with y_2. This will end with the stack containing one copy of y_2 and then one copy of y_1. Continuing this we can get $y_3, \ldots, y_{k-1}$ such that each x_i is a neighbor of each y_j, and the stack contains one copy of y_{k-1}, one copy of y_{k-2} and so on all the way till y_1.

Now, by popping the nodes on the stack, we can ensure that the next $k-1$ input letters are $y_{k-1}, \ldots, y_2, y_1$ in that order. While popping y_i (which happens upon reading $\overline{y_i}$), we will store y_i in the state of $\mathcal{M}_i$. Hence, at this point, each $\mathcal{M}_i$ will store y_i and $\mathcal{M}_0$ will store $\Diamond$. This completes the third sub-part and also the first part.

Parts 2 and 3: At the beginning of the first part, we started with a node x_i in the state of each $\mathcal{M}_i$. At the end of the first part, we have ensured that $x_1, \ldots, x_{k-1}$ is a $(k-1)$-clique, found $k-1$ more nodes $y_1, \ldots, y_{k-1}$ such that each x_i is a neighbor of each y_j and stored each y_i in $\mathcal{M}_i$.

The second and third parts are obtained by repeating the same procedure as the first part from where it stopped. More precisely, in the second part, the machines will check that $y_1, \ldots, y_{k-1}$ is a $(k-1)$-clique, find $k-1$ more nodes $z_1, \ldots, z_{k-1}$ such that each y_i is a neighbor of each z_j and then store each z_i in $\mathcal{M}_i$. Then, in the third part, the machines will check that $z_1, \ldots, z_{k-1}$ is a $(k-1)$-clique, find $k-1$ more nodes $x'_1, \ldots, x'_{k-1}$ such that each z_i is a neighbor of each x'_j and store x'_i in each $\mathcal{M}_i$.

Hence, at the end of the third part, each $\mathcal{M}_i$ stores the node x'_i. By construction of the three parts, it would then follow that if each $x'_i = x_i$, then the nodes $x_1, \ldots, x_{k-1}, y_1, \ldots, y_{k-1}, z_1, \ldots, z_{k-1}$ together form a $3(k-1)$-clique. So, under the assumption that in each machine we store the same node at the end as the one that we started off with, there is a word in the intersection of the languages of all the machines if and only if the given graph has a $3(k-1)$-clique. A natural question arises at this point.

Q: How can we get rid of this assumption?

A: Before we begin the first part, we add a zeroth part (which we call the prologue) in which we read $k-1$ nodes, check that the i^{th} node read is the same as the node in the state of $\mathcal{M}_i$ and push each node onto the stack as we read it. In this way, the prologue ensures that when we begin the first part with nodes $x_1, \ldots, x_{k-1}$, they are already on the stack.

Then we proceed to execute the first part, second part and third part as mentioned above. Note that nowhere in any of these parts did we ever need the stack to be empty to make a transition. Hence, for the execution of these three parts, it does not matter what stack content we began with, and so, even with the addition of the zeroth part, the execution of these three parts will be exactly the same as described before, except for the following fact: At the end of the three parts,

each $\mathcal{M}_i$ stores x'_i and the stack of $\mathcal{M}_0$ contains the same content as the end of the prologue.

Now, we add another last part (which we call the epilogue) which reads $k-1$ letters and attempts to pop the node from the stack corresponding to the input letter as it is read and ensures that the i^{th} letter that is popped is the same as the node stored in $\mathcal{M}_{k-i}$. This will ensure that the nodes that were put in the stack during the prologue are the same as the nodes $x'_1, \ldots, x'_{k-1}$, which is what we wanted to verify. $\lhd$

This completes the construction of our reduction except for the fact that we have linear-sized input and stack alphabets.

Q: How can we convert the linear-size alphabets into constant alphabets?

A: The idea is to encode each node i using its binary representation. So, now each node can be represented only by 0's and 1's. Furthermore, in the transitions, we replace pushing the node i onto the stack with pushing its most significant bit (msbf) representation onto the stack. Similarly, we replace popping the node i from the stack with popping its least significant bit (lsbf) representation from the stack. When this encoding is done in a naive way, for each transition labelled with some node, this would incur an extra logarithmic amount of states. Overall, this would then give us an extra $O(m \log n)$ states where m is the number of edges in the graph G. Since m could be n^2, this is not efficient enough for our reduction.

We now sketch how to circumvent this naive method with a more efficient procedure resulting in only an extra $O(n \log n)$ states. The crucial observation for this reduction in the state-space is the following one, stated here informally: *Every* state q storing a node i in each of the machines in our construction obeys one of the following three conditions.

- It only allows to either read i or $\bar{i}$ as input (but not both): In this case, we can only read i or $\bar{i}$ from that state. Hence, we only need $O(\log n)$ more states to encode the node i, i.e., we need $O(\log n)$ more states to check that we are reading i or $\bar{i}$ and then we can non-deterministically choose any of the outgoing transitions from that state.
- It allows for reading any letter from $\{i : 1 \leq i \leq n\}$ or $\{\bar{i} : 1 \leq i \leq n\}$ (but not both) and it goes to the same state irrespective of which letter is read: In this case even though we can read any node, the final state reached is the same. Hence, we only have to ensure that a valid binary representation of *some node* is read, i.e., some string in $\{0,1\}^{\log n}$ is read, which can be ensured by having $O(\log n)$ more states. Once such a string is read, we know that we can move to exactly one state.
- It only allows to read $\#$ as input: In this case we do not need any more states, as we are only replacing the encoding of the nodes i.

This means that for any state storing some node, we only need $O(\log n)$ more states. The second observation for this reduction is that the number of states which do not store any node, i.e., store $\Diamond$, is only a function of k. Furthermore, any state that stores $\Diamond$ has only one outgoing transition for each node of the graph. This means that we can afford to spend $O(\log n)$ states replacing each outgoing transition from these states and in the end, we would end up spending $O(n \log n)$ states for each state storing $\Diamond$. Since the overall number of states storing $\Diamond$ is a function of k only, the total number of states we introduce this way is still $O(n \log n)$ (because $3(k-1)$ was a fixed constant to begin with). $\lhd$

Finally, we end up with a construction in which each of the machines have $O(n \log n)$ states and the input and stack alphabets are constant-sized. Furthermore, all of these machines can be constructed in time $O(|G| + n \log n)$. Using this, we can then show that if we can solve $\mathrm{PDA} \cap \mathrm{NFA}^{k-1}$ in $O(n^{\omega(k-1)-\varepsilon})$ time (resp. in $O(n^{3(k-1)-\varepsilon})$ time) for some $\varepsilon > 0$, then $3(k-1)$-clique can be solved in $O(n^{\omega(k-1)-\varepsilon/2})$ time (resp. in $O(n^{3(k-1)-\varepsilon/2})$ time). This proves Theorem 5.

V. 2NPDA(k) and Equivalences with $\mathrm{PDA} \cap \mathrm{NFA}^{k-1}$

In the previous sections, we have shown lower bounds for the $\mathrm{PDA} \cap \mathrm{NFA}^{k-1}$ non-emptiness problem based on the number of states of the underlying machines. However, this does not preclude the possibility that an $O(N^{3k-\varepsilon})$ time algorithm exists for this problem (for some $\varepsilon > 0$), where N is the total number of bits required to encode all the machines. Note that N might be quadratic in the number of states. In *dense* machines, the number of transitions is quadratic in the number of states. Even when $k = 1$, i.e., PDA non-emptiness, no algorithm is known that runs in time $O(N^{3-\varepsilon})$ for any $\varepsilon > 0$. Unfortunately, standard existing hypotheses in fine-grained complexity theory seem to be insufficient for explaining this hardness aspect. Furthermore, it is known that, unless breakthrought results in circuit complexity appear, perhaps the most well-known hypothesis of fine-grained complexity theory (namely, the strong exponential-time hypothesis, SETH) cannot be used to help explain this hardness [15].

Recently, a new NFA acceptance hypothesis was introduced [8]. Assuming this hypothesis, no algorithm (combinatorial or otherwise) can solve PDA non-emptiness for dense PDA in time $O(n^{3-\varepsilon})$ for any $\varepsilon > 0$ where n is the number of states [8]. However, even this result does not explain the absence of $O(N^{3-\varepsilon})$ time algorithms where N is the number of bits of the input. Also, it is not clear how to extend (or use) that hypothesis to also help explain the absence of faster algorithms for the $\mathrm{PDA} \cap \mathrm{NFA}^{k-1}$ non-emptiness problem.

In this paper we propose a new hypothesis, called the 2NPDA(k) *hypothesis*, on the computational complexity of the 2NPDA(k) language recognition problem (see below). Based on this hypothesis, we prove that there can be no algorithm running in time $O(N^{3k-\varepsilon})$ for the $\mathrm{PDA} \cap \mathrm{NFA}^{k-1}$ non-emptiness problem. To corroborate this hypothesis, we provide reductions between the 2NPDA(k) language recognition problem and other problems in language theory and program analysis. We now move on to describing the 2NPDA(k) language recognition problem and the associated hypothesis.

A. Two-Way Multihead Nondeterministic Pushdown Automata

A *two-way k-head nondeterministic pushdown automaton* (2NPDA(k)) [23], [26] is a machine that consists of a finite set of control states, a read-only input tape, a pushdown store, i.e.,

a stack, and k heads that read the input tape. There is a single initial control state and a subset of states marked as accepting. Based on the control state, the top of the stack, and the letters read by the heads, the machine can nondeterministically pick a transition that updates its control state, replaces the top symbol on the pushdown store with a (possibly empty) string, and moves each head to the left or right. An input word w over the machine's input alphabet is placed on the input tape between designated "end of tape" markers $\triangleleft$ and $\triangleright$.

The machine starts from its initial state with an empty pushdown store, and applies its transitions. A run on an input word is a sequence of transitions starting from the initial state consistent with the word. A run is accepting if it leads to an accepting state; we say the word is *accepted* by the machine. Without loss of generality, we can assume above that a word is accepted in a final state with all heads scanning the right end marker and the pushdown store being empty. The *language* of the machine is the set of accepted words.

A 2DPDA(k) is a 2NPDA(k) machine in which the transition relation is deterministic: there is at most one outcome for any state, top of stack, and letters being read by the heads.

A formal description of 2NPDA(k) machines, following [23] and [26], is given in Appendix C-A. The following is the central decision problem for us:

2NPDA(k) Language Recognition

Fix: 2NPDA(k) M.
Input: A word w.
Decide: Is w is accepted by M?

This actually specifies a family of decision problems, one for each 2NPDA(k) machine M. When referring to a problem from this family, we write "***M*-language recognition**", for any fixed machine M. We are now ready to state the 2NPDA(k) hypothesis, $k \geq 1$.

2NPDA(k) Hypothesis

There is a fixed 2NPDA(k) machine M such that the M-language recognition problem cannot be solved in time $O(|w|^{3k-\varepsilon})$ for any $\varepsilon > 0$.

This is an extension of the 2NPDA(1) hypothesis that was introduced by Neal [30] and Heintze and McAllester [24]. They successfully used the 2NPDA(1) hypothesis to explain the lack of sub-cubic algorithms for many problems in program analysis.

Remark 6. *The class of* 2NPDA(k) *problems has been studied both in language theory and in complexity theory. Ibarra [26] proved that the hierarchy is strict: for each $k \geq 1$,* 2NPDA(k) $\subsetneq$ 2NPDA($k+1$). *Miyano [29] showed that each class has a hardest language. Cook [16, Corollary 1] showed that the union of* 2NPDA(k) *problems for all $k \geq 1$ precisely captures the class* PTIME.

B. Equivalences of 2NPDA(k) *with other problems*

We now show that the 2NPDA(k) language recognition problem is linear-time equivalent to a collection of other problems from program analysis and language theory. Let us define a *linear-time reduction* from problem Π_1 to problem Π_2 as a linear-time algorithm f that takes as input an instance x of Π_1 and outputs an instance $f(x)$ of Π_2 such that x is a yes-instance if and only if $f(x)$ is. Problems Π_1 and Π_2 are linear-time equivalent if there exist linear-time reductions from Π_1 to Π_2 and from Π_2 to Π_1.

We first consider the following decision problem, where L is a context-free language (CFL), i.e., a language L recognized by some PDA.

CFL k-Intersection Reachability

Fix: CFL L.
Input: k NFAs $A_1, \ldots, A_k$ over the same alphabet as L.
Decide: Does the intersection $\mathcal{L}(A_1) \cap \ldots \cap \mathcal{L}(A_k)$ contain a word from L?

Note that $k \geq 1$ is fixed, L is fixed, and the common input alphabet of the k NFAs is fixed.

The above definition actually specifies a family of problems, one for each $k \geq 1$ and each language L. When referring to a problem from this family, we will write "**(L, k)-intersection reachability**". The problem of DCFL k-intersection reachability for DFAs is the special case of CFL k-intersection reachability when the language L is a DCFL, i.e., a context-free language recognized by a deterministic PDA and all of the NFAs $A_1, \ldots, A_k$ are actually DFAs. The general case of the problem will also be referred to as CFL k-intersection reachability for NFAs. For NFAs, note the subfamily with $k = 1$ is exactly the well-known CFL reachability problem [41].

Having introduced this problem, in the rest of this subsection, we show (informally speaking) that for each $k \geq 1$ the following problems are linear-time equivalent:

- 2NPDA(k) language recognition,
- DCFL k-intersection reachability for DFAs,
- CFL k-intersection reachability for NFAs,
- PDA $\cap$ NFA^{k-1} intersection non-emptiness,
- DPDA $\cap$ DFA^{k-1} intersection non-emptiness.

Note that 2NPDA(k) language recognition is parameterised not only by k but also by the automaton. Likewise, CFL k-intersection reachability is also parameterised by the CFL. Thus, we cannot prove that *each* problem from one family is linear-time equivalent to the PDA $\cap$ NFA^{k-1} intersection non-emptiness problem for the same k. However, what we show is that the *hardest* (most difficult) language recognition problem for 2NPDA(k) is linear-time equivalent to the intersection non-emptiness problem, as illustrated below in the formal version of the above-mentioned equivalences.

Theorem 7 (Linear-Time Equivalences)**.** *Let $k \geq 1$ be any fixed number.*

1) *For every 2NPDA(k) $\mathcal{M}$, there is a DCFL L and a linear-time reduction from the $\mathcal{M}$-language recognition*

problem to the (L,k)-intersection reachability problem for DFAs.

2) *For every CFL L, the (L,k)-intersection reachability problem for NFAs (or DFAs) has a linear-time reduction to the $\mathrm{PDA} \cap \mathrm{NFA}^{k-1}$ intersection non-emptiness (or $\mathrm{PDA} \cap \mathrm{DFA}^{k-1}$ intersection non-emptiness, respectively). Moreover, the reduction produces a DPDA if L is a DCFL and the first given NFA is a DFA.*
3) *There is a fixed 2NPDA(k) $\mathcal{M}_k$ such that $\mathrm{PDA} \cap \mathrm{NFA}^{k-1}$ intersection non-emptiness has a linear-time reduction to $\mathcal{L}(\mathcal{M}_k)$.*

Note that by traversing through this sequence of equivalences, it follows that there is a fixed 2NPDA(k) $\mathcal{M}_k$ such that, for every 2NPDA(k) $\mathcal{M}$, the $\mathcal{M}$-language recognition problem is linear-time reducible to the $\mathcal{M}_k$-language recognition problem. Hence, this proves the existence of a "hardest" 2NPDA(k) language, in terms of time complexity.

It is known that there exist "level by level" reductions between the $\mathrm{PDA} \cap \mathrm{NFA}^{k-1}$ intersection non-emptiness problem to the deterministic time hierarchy within PTIME [38], [40], [17]. It follows from Theorem 7 that 2NPDA(k) has similar reductions.

The equivalences of Theorem 7 are with respect to linear-time reductions. For measuring the computational complexity relative to the number of states and transitions in the automata, we can draw the following consequences:

Corollary 8. *Fix $k \geq 1$ and a finite alphabet Σ.*

1) *The 2NPDA(k) hypothesis is false if and only if there exists $\varepsilon > 0$ for which the $\mathrm{PDA} \cap \mathrm{NFA}^{k-1}$ Intersection Non-Emptiness problem has an algorithm with running time $O(m^{3k-\varepsilon})$, where m is the maximum number of transitions in the PDA and NFAs. The same holds for the special case of the problem, DPDA $\cap$ DFA^{k-1}.*
2) *There exists a deterministic context-free language L such that, unless the 2NPDA(k) hypothesis is false, there exists no $\varepsilon > 0$ for which the (L,k)-intersection reachability problem for DFA has an algorithm with running time $O(n^{3k-\varepsilon})$, where n is the maximum number of states in the DFAs.*

We next prove the three claims of Theorem 7.

C. 2NPDA(k) language recognition reduces to DCFL k-intersection reachability for DFAs

Let $\mathcal{M}$ be a 2NPDA(k) machine. For every $w \in \Sigma^*$, we show how to construct a DPDA P and DFAs $A_1, \ldots, A_k$ such that $w \in \mathcal{L}(\mathcal{M})$ if and only if $\mathcal{L}(P) \cap \mathcal{L}(A_1) \cap \ldots \cap \mathcal{L}(A_k) \neq \emptyset$. In our construction, the DPDA P will be independent of the input word w and determined solely by the automaton $\mathcal{M}$; thus, the DCFL L from the theorem statement will be chosen as $L = \mathcal{L}(P)$. The DFAs $A_1, \ldots, A_k$ will be constructed in time linear in the length of w.

The input alphabet of the machines $P, A_1, \ldots, A_k$ is the set δ of transitions of the 2NPDA(k) $\mathcal{M}$. The language $\mathcal{L}(P) \cap \mathcal{L}(A_1) \cap \ldots \cap \mathcal{L}(A_k)$ will consist of all accepting runs of $\mathcal{M}$ on input w. Indeed, a sequence $\rho = t_1 \ldots t_n \in \delta^*$ is an accepting run if and only if the following three conditions are satisfied:

- Transitions of ρ trace a path in the finite graph on the states of $\mathcal{M}$ from the initial state to a final state.
- Stack movements prescribed by the sequence ρ are valid, that is, the sequence of pushes and pops specified by the sequence ρ constitutes a valid computation of the underlying stack.
- For each $i \in [1,k]$, the letters on the input tape that are read by the ith head in the sequence ρ are compatible with the input tape containing the word $\triangleleft w \triangleright$, where $\triangleleft$ and $\triangleright$ are endmarker symbols.

In short, the DPDA P checks the first two conditions, and each of DFAs A_i checks the third condition for i. Further details are given in section C-B in the appendix.

D. CFL k-intersection reachability reduces to $\mathrm{PDA} \cap \mathrm{NFA}^{k-1}$ intersection non-emptiness

Let L be a fixed context-free language, i.e., L is recognized by some fixed PDA P_0. Given k NFAs, the (L,k)-intersection reachability problem asks if there is a word in $\mathcal{L}(A_1) \cap \ldots \cap \mathcal{L}(A_k)$ that belongs to L. We will now reduce the (L,k)-intersection reachability problem to the $\mathrm{PDA} \cap \mathrm{NFA}^{k-1}$ intersection non-emptiness problem.

The reduction first produces a PDA P for the language $L \cap \mathcal{L}(A_1) = \mathcal{L}(P_0) \cap \mathcal{L}(A_1)$ by utilising the standard product construction (see, e.g., Hopcroft, Motwani, and Ullman's textbook [25, Section 7.3.4]). The set of control states of PDA P is the Cartesian product of the sets of control states of P_0 and A_1. Since P_0 is fixed, the description size of P is linear in the description size of A_1.

The reduction then outputs the PDA P and NFAs $A_2, \ldots, A_k$, which together form the input of $\mathrm{PDA} \cap \mathrm{NFA}^{k-1}$ intersection non-emptiness. The correctness and running time analysis of the reduction are immediate.

We remark that, if L is a DCFL and A_1 is a DFA, then the product PDA P is in fact a DPDA [20, Theorem 3.1].

E. $\mathrm{PDA} \cap \mathrm{NFA}^{k-1}$ intersection non-emptiness reduces to 2NPDA(k) language recognition

The input to the $\mathrm{PDA} \cap \mathrm{NFA}^{k-1}$ intersection non-emptiness problem is a concatenation of the string encoding the PDA P and strings encoding the NFAs $A_1, \ldots, A_{k-1}$, with delimiters separating one from another. The encodings use a fixed alphabet, which we denote by Δ; then the input is some $w \in \Delta^*$. In particular, all letters of the input alphabet of $P, A_1, \ldots, A_{k-1}$, denoted by Σ, and of the stack alphabet of P, denoted by Γ, are encoded by words from Δ^*. We describe a fixed 2NPDA(k) $\mathcal{M}_k$ that accepts $w \in \Delta^*$ if and only if w encodes some PDA P and NFAs $A_1, \ldots, A_{k-1}$ that accept some word in common, i.e., if and only if $\mathcal{L}(P) \cap \mathcal{L}(A_1) \cap \ldots \cap \mathcal{L}(A_{k-1}) \neq \emptyset$.

(The description of 2NPDA(k) $\mathcal{M}_k$ will only depend on k but not on PDA P or NFAs $A_1, \ldots, A_{k-1}$. The reduction is linear: in fact, it is just a matter of encoding the list $P, A_1, \ldots, A_{k-1}$ as a word $w \in \Delta^*$.)

```
push (the encoding of) the bottom-of-stack symbol of A_0 = P onto the stack
for i = 1 to k: do position head i to initial state of A_{i-1}
while true do
    while ∗ do          (∗ nondeterministic choice: skip or repeat ∗)
        move head 1 to an outgoing ε-transition t_0 in the encoding of A_0
        execute t_0 in A_0
    for i = 1 to k do move head i to an outgoing transition t_{i-1} in the encoding of A_{i-1}
    for i = 2 to k do check that t_0 and t_{i-1} read the same input letter a ∈ Σ
    for i = 1 to k do execute t_{i-1} in A_{i-1}
    while ∗ do          (∗ nondeterministic choice: skip or repeat ∗)
        move head 1 to an outgoing ε-transition t_0 in the encoding of A_0
        execute t_0 in A_0
    if all of A_0, A_1, ..., A_{k-1} are accepting: then accept
```

Fig. 1: Pseudocode of a 2NPDA(k) $\mathcal{M}_k$ that that accepts $w \in \Delta^*$ if and only if $\mathcal{L}(P) \cap \mathcal{L}(A_1) \cap \ldots \cap \mathcal{L}(A_{k-1}) \neq \emptyset$.

The idea is for $\mathcal{M}_k$ to guess a word in this intersection and to simulate, on the fly, k accepting runs on this word in lock-step: one in the PDA P and $k-1$ in the NFAs $A_1, \ldots, A_{k-1}$. In a nutshell, $\mathcal{M}_k$ uses each one of its heads for keeping track of the states of each machine in these accepting runs, and the stack for storing the content of the stack of the PDA P. The pseudocode in Figure 1 summarises the construction and is meant to be seen as the program of the (fixed) 2NPDA(k) $\mathcal{M}_k$. For the convenience of notation, we denote $A_0 = P$.

Details are provided in section C-D in the appendix.

F. Application: Hardest 2NPDA(k) Languages

We already observed that Theorem 7 proves the existence of a "hardest" 2NPDA(k) language in terms of time complexity. In particular, for each k there exists a fixed 2NPDA(k) $\mathcal{M}_k$ such that $\mathcal{L}(\mathcal{M})$ has a linear-time reduction to $\mathcal{L}(\mathcal{M}_k)$ whenever $\mathcal{M}$ is also a 2NPDA(k). We can strengthen this result by replacing linear-time reductions with homomorphisms, giving a new proof of the result of Miyano [29].

A classical result on the existence of "hardest" context-free language is by Greibach [21], and for 2NPDA(1) such a language was first obtained by Rytter [34]. Our hardest languages, $\mathcal{L}(\mathcal{H}_k)$ below, are different from those of Miyano [29].

Proposition 9. *For each k there exists a fixed 2NPDA(k) $\mathcal{H}_k$ over some alphabet Σ_k with the following property. For every 2NPDA(k) $\mathcal{M}$ over a finite alphabet Σ there is a homomorphism $h\colon \Sigma^* \to \Sigma_k^*$ such that, for every $w \in \Sigma^+$, we have $w \in \mathcal{L}(\mathcal{M})$ if and only if $h(w) \in \mathcal{L}(\mathcal{H}_k)$.*

We do not provide the entire proof of this result as it rests on an application of existing ideas, namely on a similar recent argument for the case $k = 1$ [15, Section 8]. We provide an outline of the proof, sketching the argument.

Conceptually, our hardest language $\mathcal{L}(\mathcal{H}_k)$ is based on the "circular" application of the three reductions of Theorem 7. For a word $w \in \Sigma^*$, the homomorphism h embeds in each morphic image $h(a)$, $a \in \Sigma$, a description of the entire 2NPDA(k) $\mathcal{M}$, encoded using an appropriate but fixed alphabet Σ_k. Roughly speaking, this enables the new fixed 2NPDA(k) $\mathcal{H}_k$ to simulate $\mathcal{M}$, using the same approach as pseudocode from Figure 1. Movements of each head of $\mathcal{H}_k$ between "blocks" $h(a)$, $a \in \Sigma$, will follow the movements of the corresponding head of $\mathcal{M}$ between individual letters a, $a \in \Sigma$, of the input word w. The stack of $\mathcal{H}_k$ will also mimic the stack of $\mathcal{M}$. Auxiliary movements and auxiliary stack operations will be required for the simulation, which are a bit tedious to describe but present no challenge.

A more sophisticated element of the construction is handling of the endmarkers. Intuitively, since the left and right tape delimiters $\lhd$ and $\rhd$ are not given to the morphism h, special treatment of these two letters is required: the automaton $\mathcal{H}_k$ "bounces back" to the main part of the tape upon hitting an endmarker and uses a copy of the description of $\mathcal{M}$ embedded in the first (or last) letter of the tape to continue the simulation. Extra care is necessary to ensure that $\mathcal{H}_k$ can process the additional information, namely that some of the heads of the simulated automaton $\mathcal{M}$ are over the endmarker instead of the first (respectively, last) letter of the input word. The technique of [15, Section 8] can be used to this end. This completes the proof outline, as well as a sketch of the construction of the hardest language $\mathcal{L}(\mathcal{H}_k)$.

ACKNOWLEDGMENTS

We thank Marvin Künnemann, Neha Rino, Henry Sinclair-Banks, and Karol Węgrzycki for useful discussions. ARB and RM were sponsored in part by the Deutsche Forschungsgemeinschaft project 389792660 TRR 248—CPEC. DC is supported by the Engineering and Physical Sciences Research Council [EP/X03027X/1] and by the Centre for Discrete Mathematics and its Applications (DIMAP) and Department of Computer Science, at the University of Warwick.

We hoped that this work would be finished in time for Javier Esparza's Festschrift in 2024. It took us much longer. The authors would still like to offer the result as a belated birthday present for Javier's 60th!

References

[1] A. Abboud, A. Backurs, K. Bringmann, and M. Künnemann, "Fine-grained complexity of analyzing compressed data: Quantifying improvements over decompress-and-solve," in *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*, C. Umans, Ed. IEEE Computer Society, 2017, pp. 192–203. [Online]. Available: https://doi.org/10.1109/FOCS.2017.26

[2] A. Abboud, A. Backurs, and V. Vassilevska Williams, "If the current clique algorithms are optimal, so is valiant's parser," *SIAM J. Comput.*, vol. 47, no. 6, pp. 2527–2555, 2018. [Online]. Available: https://doi.org/10.1137/16M1061771

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, "Time and tape complexity of pushdown automaton languages," *Information and Control*, vol. 13, no. 3, pp. 186–206, 1968. [Online]. Available: https://doi.org/10.1016/S0019-9958(68)91087-5

[4] C. Aiswarya, S. Mal, and P. Saivasan, "Satisfiability of context-free string constraints with subword-ordering and transducers," in *41st International Symposium on Theoretical Aspects of Computer Science, STACS 2024, March 12-14, 2024, Clermont-Ferrand, France*, ser. LIPIcs, O. Beyersdorff, M. M. Kanté, O. Kupferman, and D. Lokshtanov, Eds., vol. 289. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, pp. 5:1–5:20. [Online]. Available: https://doi.org/10.4230/LIPIcs.STACS.2024.5

[5] A. Backurs, N. Dikkala, and C. Tzamos, "Tight hardness results for maximum weight rectangles," in *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, ser. LIPIcs, I. Chatzigiannakis, M. Mitzenmacher, Y. Rabani, and D. Sangiorgi, Eds., vol. 55. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, pp. 81:1–81:13. [Online]. Available: https://doi.org/10.4230/LIPIcs.ICALP.2016.81

[6] A. Bouajjani, J. Esparza, and O. Maler, "Reachability analysis of pushdown automata: Application to model-checking," in *CONCUR '97: Concurrency Theory, 8th International Conference, Warsaw, Poland, July 1-4, 1997, Proceedings*, ser. Lecture Notes in Computer Science, vol. 1243. Springer, 1997, pp. 135–150. [Online]. Available: https://doi.org/10.1007/3-540-63141-0_10

[7] K. Bringmann, "Fine-grained complexity theory (tutorial)," in *36th International Symposium on Theoretical Aspects of Computer Science, STACS 2019, March 13-16, 2019, Berlin, Germany*, ser. LIPIcs, R. Niedermeier and C. Paul, Eds., vol. 126. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 4:1–4:7. [Online]. Available: https://doi.org/10.4230/LIPIcs.STACS.2019.4

[8] K. Bringmann, A. Grønlund, M. Künnemann, and K. G. Larsen, "The NFA acceptance hypothesis: Non-combinatorial and dynamic lower bounds," *TheoretiCS*, vol. 3, 2024. [Online]. Available: https://doi.org/10.46298/theoretics.24.22

[9] K. Bringmann and P. Wellnitz, "Clique-Based Lower Bounds for Parsing Tree-Adjoining Grammars," in *28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), J. Kärkkäinen, J. Radoszewski, and W. Rytter, Eds., vol. 78. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017, pp. 12:1–12:14. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CPM.2017.12

[10] T. M. Chan, "A (slightly) faster algorithm for Klee's measure problem," in *Proceedings of the 24th ACM Symposium on Computational Geometry, College Park, MD, USA, June 9-11, 2008*. ACM, 2008, pp. 94–100. [Online]. Available: https://doi.org/10.1145/1377676.1377693

[11] K. Chatterjee, B. Choudhary, and A. Pavlogiannis, "Optimal Dyck reachability for data-dependence and alias analysis," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, pp. 30:1–30:30, 2018. [Online]. Available: https://doi.org/10.1145/3158118

[12] S. Chaudhuri, "Subcubic algorithms for recursive state machines," in *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, G. C. Necula and P. Wadler, Eds. ACM, 2008, pp. 159–169. [Online]. Available: https://doi.org/10.1145/1328438.1328460

[13] H. Chen, D. Dean, and D. A. Wagner, "Model checking one million lines of C code," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2004, San Diego, California, USA*. The Internet Society, 2004. [Online]. Available: https://www.ndss-symposium.org/ndss2004/model-checking-one-million-lines-c-code/

[14] H. Chen and D. A. Wagner, "MOPS: an infrastructure for examining security properties of software," in *Proceedings of the 9th ACM Conference on Computer and Communications Security, CCS 2002, Washington, DC, USA, November 18-22, 2002*, V. Atluri, Ed. ACM, 2002, pp. 235–244. [Online]. Available: https://doi.org/10.1145/586110.586142

[15] D. Chistikov, R. Majumdar, and P. Schepper, "Subcubic certificates for CFL reachability," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, pp. 1–29, 2022. [Online]. Available: https://doi.org/10.1145/3498702

[16] S. A. Cook, "Characterizations of pushdown machines in terms of time-bounded computers," *J. ACM*, vol. 18, no. 1, pp. 4–18, 1971. [Online]. Available: https://doi.org/10.1145/321623.321625

[17] M. de Oliveira Oliveira and M. Wehar, "On the fine grained complexity of finite automata non-emptiness of intersection," in *Developments in Language Theory - 24th International Conference, DLT 2020, Tampa, FL, USA, May 11-15, 2020, Proceedings*, ser. Lecture Notes in Computer Science, N. Jonoska and D. Savchuk, Eds., vol. 12086. Springer, 2020, pp. 69–82. [Online]. Available: https://doi.org/10.1007/978-3-030-48516-0_6

[18] D. Dolev, S. Even, and R. M. Karp, "On the security of ping-pong protocols," *Inf. Control.*, vol. 55, no. 1-3, pp. 57–68, 1982. [Online]. Available: https://doi.org/10.1016/S0019-9958(82)90401-6

[19] A. Finkel, B. Willems, and P. Wolper, "A direct symbolic approach to model checking pushdown systems," in *Second International Workshop on Verification of Infinite State Systems, Infinity 1997, Bologna, Italy, July 11-12, 1997*, ser. Electronic Notes in Theoretical Computer Science, F. Moller, Ed., vol. 9. Elsevier, 1997, pp. 27–37. [Online]. Available: https://doi.org/10.1016/S1571-0661(05)80426-8

[20] S. Ginsburg and S. A. Greibach, "Deterministic context free languages," *Inf. Control.*, vol. 9, no. 6, pp. 620–648, 1966. [Online]. Available: https://doi.org/10.1016/S0019-9958(66)80019-0

[21] S. A. Greibach, "The hardest context-free language," *SIAM J. Comput.*, vol. 2, no. 4, pp. 304–310, 1973. [Online]. Available: https://doi.org/10.1137/0202025

[22] J. C. Hansen, A. H. Kjelstrøm, and A. Pavlogiannis, "Tight bounds for reachability problems on one-counter and pushdown systems," *Inf. Process. Lett.*, vol. 171, p. 106135, 2021. [Online]. Available: https://doi.org/10.1016/j.ipl.2021.106135

[23] M. A. Harrison and O. H. Ibarra, "Multi-tape and multi-head pushdown automata," *Inf. Control.*, vol. 13, no. 5, pp. 433–470, 1968. [Online]. Available: https://doi.org/10.1016/S0019-9958(68)90901-7

[24] N. Heintze and D. A. McAllester, "On the cubic bottleneck in subtyping and flow analysis," in *Proceedings, 12th Annual IEEE Symposium on Logic in Computer Science (LICS), Warsaw, Poland, June 29 - July 2, 1997*. IEEE Computer Society, 1997, pp. 342–351. [Online]. Available: https://doi.org/10.1109/LICS.1997.614960

[25] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[26] O. H. Ibarra, "On two-way multihead automata," *J. Comput. Syst. Sci.*, vol. 7, no. 1, pp. 28–36, 1973. [Online]. Available: https://doi.org/10.1016/S0022-0000(73)80048-0

[27] J. Kodumal and A. Aiken, "Regularly annotated set constraints," in *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, J. Ferrante and K. S. McKinley, Eds. ACM, 2007, pp. 331–341. [Online]. Available: https://doi.org/10.1145/1250734.1250772

[28] A. A. Mathiasen and A. Pavlogiannis, "The fine-grained and parallel complexity of Andersen's pointer analysis," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, pp. 1–29, 2021. [Online]. Available: https://doi.org/10.1145/3434315

[29] S. Miyano, "Remarks on multihead pushdown automata and multihead stack automata," *Journal of Computer and System Sciences*, vol. 27, no. 1, pp. 116–124, 1983. [Online]. Available: https://doi.org/10.1016/0022-0000(83)90032-6

[30] R. Neal, "The computational complexity of taxonomic inference," 1989, unpublished manuscript. Available at http://www.cs.toronto.edu/~radford/ftp/taxc.pdf.

[31] A. Pavlogiannis, "CFL/Dyck reachability: An algorithmic perspective," *ACM SIGLOG News*, vol. 9, no. 4, pp. 5–25, 2022. [Online]. Available: https://doi.org/10.1145/3583660.3583664

[32] T. W. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, R. K. Cytron and P. Lee, Eds. ACM Press, 1995, pp. 49–61. [Online]. Available: https://doi.org/10.1145/199448.199462

[33] A. Rountev, A. L. Milanova, and B. G. Ryder, "Points-to analysis for java using annotated constraints," in *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2001, Tampa, Florida, USA, October 14-18, 2001*, L. M. Northrop and J. M. Vlissides, Eds. ACM, 2001, pp. 43–55. [Online]. Available: https://doi.org/10.1145/504282.504286

[34] W. Rytter, "A hardest language recognized by two-way nondeterministic pushdown automata," *Inf. Process. Lett.*, vol. 13, no. 4/5, pp. 145–146, 1981. [Online]. Available: https://doi.org/10.1016/0020-0190(81)90045-4

[35] ——, "A note on two-way nondeterministic pushdown automata," *Inf. Process. Lett.*, vol. 15, no. 1, pp. 5–9, 1982. [Online]. Available: https://doi.org/10.1016/0020-0190(82)90075-8

[36] ——, "A simulation result for two-way pushdown automata," *Inf. Process. Lett.*, vol. 16, no. 4, pp. 199–202, 1983. [Online]. Available: https://doi.org/10.1016/0020-0190(83)90124-2

[37] ——, "Fast recognition of pushdown automaton and context-free languages," *Information and Control*, vol. 67, no. 1-3, pp. 12–22, 1985. [Online]. Available: https://doi.org/10.1016/S0019-9958(85)80024-3

[38] J. Swernofsky and M. Wehar, "On the complexity of intersecting regular, context-free, and tree languages," in *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II*, ser. Lecture Notes in Computer Science, M. M. Halldórsson, K. Iwama, N. Kobayashi, and B. Speckmann, Eds., vol. 9135. Springer, 2015, pp. 414–426. [Online]. Available: https://doi.org/10.1007/978-3-662-47666-6_33

[39] V. Vassilevska Williams, "On some fine-grained questions in algorithms and complexity," in *International Congress of Mathematicians (ICM'18)*, 2018, available at https://eta.impa.br/dl/194.pdf and https://people.csail.mit.edu/virgi/eccentri.pdf.

[40] M. Wehar, "On the complexity of intersection non-emptiness problems," Ph.D. dissertation, 2016, SUNY Buffalo. [Online]. Available: http://www.michaelwehar.com/documents/mwehar_dissertation.pdf

[41] M. Yannakakis, "Graph-theoretic methods in database theory," in *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, April 2-4, 1990, Nashville, Tennessee, USA*. ACM Press, 1990, pp. 230–242. [Online]. Available: https://doi.org/10.1145/298514.298576

APPENDIX A
PROOF OF THEOREM 2

Let G be a graph (without self-loops) and $3k$ be a given number. Without loss of generality, let $\{0, 1, \ldots, n-1\}$ be the vertices of G. We will now construct a DOCA $\mathcal{M}_0$ and k many DFAs $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_{k-1}$ over a common alphabet Σ such that the intersection of $\mathcal{M}_0, \mathcal{M}_1, \ldots, \mathcal{M}_{k-1}$ is non-empty if and only if G has a $3k$-clique.

The high-level idea behind the construction of these machines has been described in the main part of the paper and so we concentrate here on the formal aspects. We will construct each machine incrementally in three stages. In each stage, we will introduce some specific machines and prove some properties regarding each of them. We will then compose the machines at each stage to get the desired final construction (i.e., the final machines $\mathcal{M}_0, \ldots, \mathcal{M}_{k-1}$).

Before we describe the three stages, we make a small remark. In each of the three stages, we will actually construct machines such that for each state q and each letter a, there is at most one outgoing transition for the pair (q, a). Strictly speaking, these do not correspond to deterministic machines, because determinism mandates that there be exactly one outgoing transition for the pair (q, a). However, it is easy to see that any such machine can be converted into a language-equivalent deterministic machine by adding a special sink state to which all undefined outgoing transitions are diverted to. The reason we do not add this sink state to all of the machines in our construction is purely for expository purposes as it makes the construction and proofs easier to state and prove. Having made this remark, we now proceed to the first stage of the construction.

First stage: Machines for storing a k-tuple by incrementing the counter: In this stage, we will introduce machines which will allow us to uniquely store a k-tuple of nodes as a number in the counter of the DOCA. As mentioned in the intuition in the main part of the paper, to every k-tuple $C = (v_0, v_1, \ldots, v_{k-1})$ of nodes, we can uniquely assign a number $n^{k-1} \cdot v_{k-1} + n^{k-2} \cdot v_{k-2} + \cdots + v_0$. In this stage, we will construct machines which will first force the counter of the DOCA to reach a value of the above form for some collection of nodes $v_{k-1}, \ldots, v_0$. (In the next stage, we will check that this collection of nodes is indeed a k-clique).

Description of $A_0, \ldots, A_{k-1}$: We now proceed with the formal aspects. We will construct a DOCA A_0 and $k-1$ many DFAs $A_1, \ldots, A_{k-1}$ in this first stage. The common set of input letters for these machines will be $\#_0, \#_1, \ldots, \#_{k-1}$ and $@_0, @_1, \ldots, @_{k-1}$. Each machine $A_i \in \{A_0, \ldots, A_{k-1}\}$ will have the following set of states: For each $s \in \{0, \ldots, n-1\}$ and each $\ell \in \{0, \ldots, k-1\}$, A_i will have a state $(s, \ell)^{+i}$. Furthermore, for each $s \in \{0, \ldots, n-1\}$, A_i will also have a state $(s, done)^{+i}$. The $+$ in the superscript in these states denotes that these are the machines in the first stage responsible for incrementing the counter and the i in the superscript denotes the machine to which these states belong to. The intuition behind these states are that for each $\ell \in \{0, \ldots, k-1\}$, the states $(0, \ell)^{+i}, (1, \ell)^{+i}, \ldots, (n-1, \ell)^{+i}$ correspond to the $(k-\ell)^{th}$ copy of the gadget G_i described in the proof idea in the main part of the paper. The states $(0, done)^{+i}, (1, done)^{+i}, \ldots, (n-1, done)^{+i}$ denote that we have frozen the i^{th} machine with the values $0, 1, \ldots, n-1$ respectively.

Before we describe the transitions of each of these machines, we build some intuition. Each machine A_i will begin at $(0, k-1)^{+i}$. Note that each state of A_i is of the form $(s, p)^{+i}$ for some $s \in \{0, \ldots, n-1\}$ and some $p \in \{0, \ldots, k-1, done\}$. The first part s will be called the *score* of that state and the second part p will be called the *phase* of that state. If the phase is ℓ for some $\ell \in \{0, \ldots, k-1\}$, then we say that the state is active and otherwise, we say that it is done.

The transitions that we will construct will always satisfy the following property: Suppose while reading a word, the machines $A_0, A_1, \ldots, A_{k-1}$ reach states with scores $s_0, \ldots, s_{k-1}$ at some point. Then the value of the counter of A_0 at that point will be *exactly* equal to $\sum_{0 \leq i \leq k-1} n^i s_i$. Furthermore, if some machine A_i reaches a state of the form $(s_i, done)^{+i}$, then this would intuitively mean that we have finished the increments corresponding to the n^i term in our representation, i.e., that we have decided on picking the i^{th} node to be s_i. Intuitively,

then in order to complete our task, we must force each A_i to reach a state of the form $(s_i, done)^{+i}$, i.e., we must force each A_i to reach a done state. This we will do by first forcing A_{k-1} to reach a done state, then A_{k-2}, then A_{k-3} and so on.

We will now formally describe the transitions. The machine A_0 will have the following transitions: For each active state $(s, \ell)^{+0}$,

- If we read $\#_0$ and $s < n-1$, then we increment the counter by 1 and move to $(s+1, \ell)^{+0}$.
- If we read any one of $\#_1, \#_2, \ldots, \#_{k-1}$, and $s = n-1$, then we increment the counter by 1 and move to $(0, \ell)^{+0}$.
- If we read $@_\ell$ and $\ell > 0$ and $s = 0$, then we move to $(0, \ell-1)^{+0}$.
- If we read $@_\ell$ and $\ell = 0$, then we move to $(s, done)^{+0}$.
- In all the other cases, the transition is not defined.

Each machine A_i for $i \geq 1$ will have the following transitions: For each active state $(s, \ell)^{+i}$,

- If we read any one of $\#_0, \ldots, \#_{i-1}$, then we stay at $(s, \ell)^{+i}$.
- If we read $\#_i$, and $s < n-1$, then we move to $(s+1, \ell)^{+i}$.
- If we read any one of $\#_{i+1}, \ldots, \#_{k-1}$, and $s = n-1$, then we move to $(0, \ell)^{+i}$.
- If we read $@_\ell$ and $\ell > i$ and $s = 0$, then we move to $(0, \ell-1)^{+i}$.
- If we read $@_\ell$ and $\ell = i$, then we move to $(s, done)^{+i}$.
- In all the other cases, the transition is not defined.

Furthermore, for each done state $(s, done)^{+i}$, if we read any one of $\#_0, \ldots, \#_{i-1}, @_0, \ldots, @_{i-1}$, we stay there and in any other case, the transition is not defined.

This completes the description of all the machines. Recall that the initial state of each machine A_i is $(0, k-1)^{+i}$. Note that in every machine, for each state q and each letter a, there is at most one state q' to which the machine can move to while reading the letter a from q. Hence, for every machine A_i and every word w, there can be at most one run of machine A_i.

Properties of $A_0, \ldots, A_{k-1}$: We now prove some properties of these machines and show that they conform with the intuitions given above. To this end, let us set up some notation. Suppose there exist runs of the machines $A_0, \ldots, A_{k-1}$ on some word w. We say that the runs are active with phase $k-1-i$ for some $0 \leq i \leq k-1$ if at the end of the runs,

- The states of the machines $A_0, \ldots, A_{k-1-i}$ are all active with phase $k-1-i$,
- The states of the machines $A_{k-i}, \ldots, A_{k-1}$ are all done.

Finally, we say that this collection of runs is perfect if at the end of each run for each machine, the phase of the state of that machine is done.

Having stated these definitions, we state our first result. It proves that the collection of runs that we get for any word w is either active or perfect.

Lemma 10 (Incrementing Counter Soundness Lemma)**.** *Let $\rho = (\rho_0, \ldots, \rho_{k-1})$ be a collection of runs of the machines $A_0, \ldots, A_{k-1}$ along some word w. Then ρ is either active or perfect. Furthermore, if $s_0, \ldots, s_{k-1}$ are the scores of the states of $A_0, \ldots, A_{k-1}$ at the end of ρ, then the value of the counter of A_0 at the end of ρ is $\sum_{0 \leq i \leq k-1} n^i s_i$.*

Proof. This follows by an induction on the length of w. Note that the collection for the empty word is active. Suppose we have already shown this property for some word w and we would like to show this for the word wa where a is some letter. Let ρ be a collection of runs for w.

Suppose ρ is perfect. Then, since there are no outgoing transitions from any done state of A_0, it follows that reading any letter at this point will not produce a run in A_0 and so there is nothing left to prove for the word wa.

Suppose ρ is active. Hence there is some $i \geq 0$ such that at the end of the runs in ρ,

- The states of the machines $A_0, \ldots, A_{k-1-i}$ are all active with phase $k-1-i$,
- The states of the machines $A_{k-i}, \ldots, A_{k-1}$ are all done.

Let s_ℓ be the score of the state of each A_ℓ at the end of reading w. By induction hypothesis, the value of the counter of A_0 after reading w is $\sum_{0 \leq \ell \leq k-1} n^\ell s_\ell$.

Suppose there is a first index e in $\{0, \ldots, k-1-i\}$ such that $s_e < n-1$. Now let us consider all possible values that the letter a can have.

- If $a \notin \{\#_0, \#_1, \ldots, \#_e, @_{k-1-i}\}$, then no transition at A_e is possible, and in this case, there is nothing left to prove for the word wa.
- Suppose $a \in \{\#_0, \ldots, \#_{e-1}\}$. Let $a = \#_x$ for some $x \in \{0, \ldots, e-1\}$. By assumption on e, the machine A_x is at the state $(n-1, k-1-i)^{+x}$ at the end of reading w. However, from this state there is no outgoing transition labelled by $\#_x$. Hence, in this case as well, there is nothing left to prove for the word wa.
- Suppose $a = \#_e$. Then notice that the machines $A_0, \ldots, A_{e-1}$ will move to $(0, k-1-i)^{+0}, \ldots, (0, k-1-i)^{+(e-1)}$, A_e will move to $(s_e+1, k-1-i)^{+e}$ and all the other machines will remain where they are and the counter will be increased by 1. Hence, in this case, we get an active run with phase $k-1-i$.
  Notice that the scores of the machines before reading a were $\underbrace{n-1 \cdots n-1}_{e-1 \text{ times}}, s_e, s_{e+1}, \ldots, s_{k-1}$ and the scores of the machines after reading a are $\underbrace{0 \cdots 0}_{e-1 \text{ times}}, s_e + 1, s_{e+1}, \ldots, s_{k-1}$. Since the counter was incremented by 1 upon reading a, using the induction hypothesis we can now conclude that the induction also carries over at this step.
- Suppose $a = @_{k-1-i}$. Note that in this case the only way that transitions are defined for the machines $A_0, \ldots, A_{k-2-i}$ is if all of the scores $j_0, \ldots, j_{k-2-i}$ are 0. In this case, the machines $A_0, \ldots, A_{k-2-i}$ will move to the states $(0, k-2-i)^{+0}, (0, k-2-i)^{+1}, \ldots, (0, k-2-i)^{+(k-2-i)}$, A_{k-1-i} will move to the state $(s_{k-1-i}, done)^{+(k-1-i)}$ and the states of all the other machines are unchanged. Hence, in this case, we get an active run with phase $k-2-i$ if $k-1-i > 0$

or a perfect run if $k-1-i=0$. Note that the scores of none of the states have changed and the counter value was undisturbed. Hence, we can conclude that the induction step also carries over at this step.

Suppose there is no index e in $\{0,\dots,k-1-i\}$ such that $s_e < n-1$. Hence, we have that $s_0 = s_1 = \cdots = s_{k-1-i} = n-1$. Note that if $a \in \{\#_{k-i},\cdots,\#_{k-1},@_{k-i}\dots,,@_{k-1}\}$, A_{k-i} does not have a transition defined for the letter a at its current state, because its current state is a done state. Further suppose $a = \#_e$ for some $e \in \{0,\dots,k-1-i\}$. Since the score of the current state of A_e is $n-1$, it follows that no transition for a is defined at the current state of A_e. Finally, suppose $a \in \{@_0,\dots,@_{k-1-i}\}$. Since the phase of the current state of A_{k-1-i} is $k-1-i$, it must be the case that $a = @_{k-1-i}$, since otherwise no transition for a is defined from the current state of A_{k-1-i}. However, if $k-1-i>0$, then the current state of the machine A_{k-2-i} has score $n-1$ from which no outgoing transition is defined for a. Hence $k-1-i$ must be 0, which means that the current state of A_0 is still active with phase 0 and the states of all the other machines are done. In this case, reading a will make A_0 move to $(n-1,done)^{+0}$ and all the other machines will remain where they are. Since the counter value was undisturbed, the scores of none of the states have changed and we get a perfect run, the induction carries over in this case as well. □

We now prove a lemma which acts as a sort of converse to the above lemma. It shows that for any number $0 \le N = \sum_{0\le i\le k-1} n^i s_i < n^k$, there is a word w with which we can force the counter value of A_0 to reach exactly N whilst simultaneously guiding all the machines to states with scores $s_0, s_1, \dots, s_{k-1}$.

Lemma 11 (Incrementing Counter Completeness Lemma)**.** *Let $0 \le N = \sum_{0\le i\le k-1} n^i s_i < n^k$ with $0 \le s_i \le n-1$ for each s_i. Then, there is a word w satisfying the following property: There is a collection of perfect runs $\rho = (\rho_0,\dots,\rho_{k-1})$ for the machines $A_0,\dots,A_{k-1}$ along the word w, with the counter of A_0 reaching the value N and each A_i reaching the state $(s_i, done)^{+i}$.*

Proof. We prove this by induction on N. Note that for the base case of $N=0$, it can be easily verified that setting $w := @_{k-1}, @_{k-2},\dots,@_0$ satisfies the claim.

Suppose we have proved the claim for some $N < n^k - 1$ and we want to prove it for $N+1$. Let w be the word that we obtain for N and let $\rho = (\rho_0,\dots,\rho_{k-1})$ be the perfect collection of runs that we get out of reading w from the machines $A_0,\dots,A_{k-1}$. For each i, let Σ_i denote the set $\{\#_0,\dots,\#_i\}$.

Since ρ is perfect, we claim that w has to be a word of the form $\Sigma_{k-1}^* @_{k-1} \Sigma_{k-2}^* @_{k-2} \dots \Sigma_0^* @_0$. To see this, note the following facts regarding our construction:

- For every i, the only transitions that take an active state of A_i to a done state of A_i are the ones labelled by $@_i$. Hence, the word w must contain at least one occurrence of $@_i$ for each i.
- For every i, once the machine A_i reaches a done state, upon reading any letter a, it either stays at the same state or has no transition for a (if $a \in \{\#_i,\dots,\#_{k-1},@_i,\dots,@_{k-1}\}$). Hence, after the first occurrence of $@_i$, no more occurrences of $\#_i$ or $@_i$ can happen.
- For every i, if the machine A_i is at an active state and reads $@_j$ for $j<i$, then transitions are undefined. This combined with the fact that the only transitions that take an active state of A_i to a done state of A_i are the ones labelled by $@_i$ leads us to conclude that the occurrence order of the letters not in Σ_{k-1} must be of the form $@_{k-1}, @_{k-2},\dots,@_0$.

These three points combined together imply that w must be a word of the form $\Sigma_{k-1}^* @_{k-1} \Sigma_{k-2}^* @_{k-2} \dots \Sigma_0^* @_0$. Hence, we let $w = w_{k-1} @_{k-1} w_{k-2} @_{k-2} \dots w_0 @_0$ with each $w_i \in \Sigma_i^*$. Now note the following fact.

> Fact: For every i and every $j<i$, the machine A_j is at an active state with score 0 before and after reading the prefixes of w which stop before and after the letter $@_i$ in w respectively.

Indeed to see the truth of this fact, note that the only way for a machine A_j to have a transition upon reading $@_i$ for some $i>j$, is if A_j is at an active state with score 0. Furthermore when A_j is at an active state with score 0 and it reads $@_i$ for $i>j$ then A_j continues to be at an active state with score 0. This then immediately implies the above fact.

Now, let $N = \sum_{0\le i\le k-1} n^i s_i$ with each s_i between 0 and $n-1$. Since $N < n^k - 1$, there must be a smallest i such that $s_i < n-1$. We now construct a new word w' as $w' = w_{k-1} @_{k-1} w_{k-2} @_{k-2} \dots w_{i+1} @_{i+1} w_i w_{i-1} \dots w_0 \#_i @_i @_{i-1} \dots @_0$, i.e., we push the letters $@_i, @_{i-1},\dots,@_0$ to the very end and insert a $\#_i$ in between w_0 and $@_i$. We claim that w' is the required word for $N+1$.

First, we have to show that each machine A_j has a run upon reading w', i.e., at no point while reading w' does A_j reach a state where a transition is undefined. Assuming the contrary, suppose there is some machine A_j which upon reading w', reaches a point where no transition is defined. Since the prefix of w' up till the end of w_i is exactly the same as w, it follows that A_j has an undefined transition only after reading w_i in w'. Note that before w_i, the letters $@_k,\dots,@_{i+1}$ have already appeared and by construction of $A_{k-1},\dots,A_{i+1}$, it follows that all these machines have already reached a done state by the time w_i is read. Furthermore, note that all the letters that appear after w_i in w' belong to $\Sigma_i \cup \{@_i, @_{i-1},\dots,@_0\}$. Since all done states in $A_{k-1},\dots,A_{i+1}$ have a self-loop upon reading any such letter it follows that $j \le i$.

By the above given fact, the machine A_j is always at an active state with score 0 before reading the subwords $w_{i-1}, w_{i-2},\dots,w_j$ in w. By construction of A_j, the transitions for reading any letter from Σ_{k-1} is independent of the phase of the active state, i.e., if an active state q with score s moves to an active state q' with score s' upon reading any letter from Σ_{k-1}, then *any* active state with score s will move

to an active state with score s' upon reading that same letter. It follows then that A_j always has an outgoing transition at each point whilst reading the subwords $w_{i-1}, \ldots, w_j$ in w'.

Now, let s be the score of the state of A_j obtained after reading the prefix of w' ending with w_j. Note that this must be the same as the score of the state of A_j obtained after reading the prefix of w ending with w_j. Now, notice that in w, after reading w_j, A_j only reads letters from $\Sigma_{j-1} \cup \{@_j, @_{j-1}, \ldots, @_0\}$. Since none of these letters can change the score of A_j it follows that s is the final score of A_j after reading the complete word w. Note that by definition of j, if $j = i$, then $s < n-1$ and if $j < i$, then $s = n-1$.

Now, notice that all active states of A_j do not change their state after reading $\#_\ell$ for any $\ell < j$. Since A_j was at a state with score s after reading the prefix of w' ending with w_j, it continues to be at a state with score s after reading the subwords $w_{j-1}, \ldots, w_0$ in w'. Now, if $j = i$, then $s < n-1$ and so A_j will move to an active state with score $s+1$ after reading $\#_i$. On the other hand if $j < i$, then $s = n-1$ and so A_j will move to an active state with score 0 after reading $\#_i$.

Now, suppose $j = i$. Hence after reading $@_i$ in w', A_j will move to a done state and continue to stay there while reading the letters $@_{i-1}, \ldots, @_0$. On the other hand, suppose $j < i$. Hence after reading $@_i, @_{i-1}, \ldots, @_{j+1}$, it will continue to remain at an active state with score 0. After reading $@_j$, it will move to a done state and continue to stay there while reading the letters $@_{j-1}, \ldots, @_0$. It follows that in either case, A_j has a run upon reading w', which leads to a contradiction.

It then follows that we have a collection of runs ρ' corresponding to the machines $A_0, \ldots, A_{k-1}$ reading the word w'. Note that w' contains exactly one occurrence of each $@_i$. Since reading the letter $@_i$ from any state in the machine A_i either has no outgoing transition or moves it to a done state and since every outgoing transition from a done state is to itself, it follows that the collection ρ' cannot be active. By the Incrementing Counter Soundness Lemma (Lemma 10), ρ' has to be perfect.

Note that the number of letters seen from Σ_{k-1} in w' is one more than the number of letters seen from Σ_{k-1} in w. Note that whenever A_0 reads some letter from Σ_{k-1}, it either increments its counter by 1 or has no transition corresponding to that letter. Since the latter cannot happen with w', it follows that the value of the counter of A_0 after reading w' is $N+1$. This completes the proof. □

The proof of these two lemmas also finishes the first stage of the reduction.

Second stage: Gadgets for finding $2k$-cliques: In this stage, we will construct gadgets which will help us find $2k$ cliques. More precisely, we will construct a DOCA B_0 and $k-1$ DFAs $B_1, \ldots, B_{k-1}$ whose states taken together initially, we will verify to be a k-clique. From there on, we will "hop" from one k-clique to another, such that whenever we hop from one k-clique S to another k-clique S', we will ensure that $S \cup S'$ is by itself a $2k$-clique.

We now look at the formal aspects. The common set of input letters for these machines will be as follows: For each k-clique S of the given graph G, we will have a letter which we will also denote by S. Further each machine $B_i \in \{B_0, \ldots, B_{k-1}\}$ will have the following set of states: For each $s \in \{0, \ldots, n-1\}$ and $p \in \{check, \alpha, \beta, \gamma, \delta\}$, we will have a state $(s,p)^i$. As before, the part s will be called the score of the state and the part p will be called the phase of the state.

We will now describe the transitions of each machine B_i. For each state $(s, check)^i$, if we read some letter $S = (v_0, v_1 \ldots, v_{k-1})$ corresponding to some k-clique where $v_i = s$, then we move to $(s, \alpha)^i$. Further for any state $(s, \ell)^i$ with $\ell \in \{\alpha, \beta, \gamma\}$, if we read some letter $S = (v_0, v_1, \ldots, v_{k-1})$ corresponding to some k-clique such that s is adjacent to every node in S, then we move to $(v_i, \ell')^i$ where

- $\ell' = \beta$ if $\ell = \alpha$
- $\ell' = \gamma$ if $\ell = \beta$
- $\ell' = \delta$ if $\ell = \gamma$

As before, in all the other cases, no transition is defined.

The following proposition immediately follows from the construction of each machine and from the fact that there are no self-loops in the given graph G.

Proposition 12. *For any i, the machine B_i, starting from some state of the form $(s, check)^i$ and reading a word w, can reach a state of the form $(s', \delta)^i$ if and only if $w = S_1, S_2, S_3, S_4$ for some k-cliques S_1, S_2, S_3, S_4 such that s is the i^{th} node of S_1, s' is the i^{th} node of S_4 and the i^{th} node of S_1, S_2, S_3 is adjacent to every node in S_2, S_3, S_4 respectively.*

As mentioned before, this gadget simply "hops" from one k-clique to another. This intuition is made concrete by the following lemma, which easily follows from the above proposition.

Lemma 13 (Clique Finding Lemma)**.** *The machines $B_0, \ldots, B_{k-1}$, starting from states of the form $(s_0, check)^0, \ldots, (s_{k-1}, check)^{k-1}$ and reading a word w can reach states of the form $(s'_0, \delta)^0, \ldots, (s'_{k-1}, \delta)^{k-1}$ if and only if $w = S_1, S_2, S_3, S_4$, for some k-cliques S_1, S_2, S_3, S_4 such that $S_1 = (s_0, \ldots, s_{k-1}), S_4 = (s'_0, \ldots, s'_{k-1})$ and $S_1 \cup S_2, S_2 \cup S_3, S_3 \cup S_4$ are all $2k$-cliques.*

This completes the second stage of the reduction.

Third stage: Gadgets for retrieving a k-tuple by decrementing the counter: In this stage, we will introduce gadgets which will allow us to retrieve a k-tuple which is stored in the counter and check that that tuple is the same as the one that is currently stored in the states of the machines. These gadgets will simply be the "reverse" of the gadgets introduced in the first stage.

Formally, we construct machines $C_0, \ldots, C_{k-1}$ as follows: Let $A_0, \ldots, A_{k-1}$ be the machines introduced in the first stage. Note that every state of every machine A_i is of the form q^{+i} for some $q \in \{0, \ldots, n-1\} \times \{0, \ldots, k-1, done\}$. With this observation, we now construct each C_i.

C_0 is an OCA which has the same alphabet as A_0. For every state of the form q^{+0} in A_0, C_0 will have a

state of the form q^{-0}. Further, C_0 has the following transitions: If (p^{+0}, a, u, q^{+0}) is a transition in A_0 where $u \in \{-1, 0, +1\}$, then C_0 has a corresponding "reverse" transition $(q^{-0}, a, -u, p^{-0})$.

We can now construct each C_i for every $i > 0$ similarly. C_i is a finite automaton which has the same alphabet as A_i. For every state of the form q^{+i} in A_i, C_i will have a state of the form q^{-i}. Further, C_i has the following transitions: If (p^{+i}, a, q^{+i}) is a transition in A_i , then C_i has a corresponding "reverse" transition (q^{-i}, a, p^{-i}).

We note that each machine C_i is deterministic. This is because in each machine A_i, for each state q^{+i} and each letter a, we had at most one *incoming transition* to the state q^{+i} upon reading a. Hence, the machine C_0 is a DOCA and the machines $C_1, \ldots, C_{k-1}$ are all DFAs.

Note that any run in any machine C_i is the reverse of some run in the machine A_i and vice versa. Hence, we can define notions of reverse-perfect and reverse-active collections of runs in C_i, as a collection of runs obtained by reversing perfect and active collections of runs in A_i. This observation combined with the Incrementing Counter lemmas (Lemma 10 and Lemma 11) that we proved in the first stage immediately implies the following two lemmas.

Lemma 14 (Decrementing Counter Soundness Lemma)**.** *Let $\rho = (\rho_0, \ldots, \rho_{k-1})$ be a collection of reverse-perfect runs of the machines $C_0, \ldots, C_{k-1}$ along some word w, when starting at some configurations of the form $((s_0, done)^{-0}, N), (s_1, done)^{-1}, \ldots, (s_{k-1}, done)^{-(k-1)}$. Then $N = \sum_{0 \leq i \leq k-1} n^i s_i$.*

Lemma 15 (Decrementing Counter Completeness Lemma)**.** *Let $0 \leq N = \sum_{0 \leq i \leq k-1} n^i s_i < n^k$ with $0 \leq s_i \leq n-1$ for each s_i. Then, there is a word w satisfying the following property: There is a collection of reverse-perfect runs $\rho = (\rho_0, \ldots, \rho_{k-1})$ for the machines $C_0, \ldots, C_{k-1}$ along the word w, when starting at the configurations $((s_0, done)^{-0}, N), (s_1, done)^{-1}, \ldots, (s_{k-1}, done)^{-(k-1)}$.*

This finishes the third stage of the reduction.

Putting the three stages together: Now we put the three stages together and complete the reduction.

For each i, we have constructed three different machines A_i, B_i, C_i. Let us now combine them together into one machine $\mathcal{M}_i$ in the following manner: $\mathcal{M}_i$ will have all the states, letters and transitions of A_i, B_i and C_i. In addition, it will have two fresh letters $!, ?$ and the following transitions:

- From each state of A_i whose phase is done, i.e., each state of the form $(s, done)^{+i}$, upon reading the letter !, we move to the state $(s, check)^i$ of B_i.
- From each state of B_i whose phase is δ, i.e., each state of the form $(s, \delta)^i$, upon reading the letter ?, we move to the state $(s, done)^{-i}$ of C_i.

Let the initial state of each $\mathcal{M}_i$ be $(0, k-1)^{+i}$ and let the final state of each $\mathcal{M}_i$ be $(0, k-1)^{-i}$. We now have the following lemma, which is a result of the lemmas that we proved in the previous stages.

Lemma 16. *There is a word w such that w is accepted by each $\mathcal{M}_i$ if and only if there is a $3k$-clique in the graph G.*

Proof. Suppose there is a $3k$-clique S in the graph G. Hence, there are three k-cliques S_1, S_2, S_3 such that $S_1 \cup S_2, S_2 \cup S_3, S_3 \cup S_1$ are each $2k$-cliques. Let $S_1 = (s_0, \ldots, s_{k-1})$.

By the Incrementing Counter Completeness lemma (Lemma 11), there is a word w_1 such that each machine $\mathcal{M}_i$, starting from $(0, k-1)^{+i}$ can read the word w_1 and reach the state $(s_i, done)^{+i}$. Furthermore, at the end of reading w_1, the counter value of $\mathcal{M}_0$ will be $N = \sum_{0 \leq i \leq k-1} n^i s_i$.

Afterwards by reading !, each machine $\mathcal{M}_i$ will move from $(s_i, done)^{+i}$ to $(s_i, check)^i$. After that, by the Clique Finding lemma (Lemma 13), upon reading the word $w_2 = S_1 S_2 S_3 S_1$, each $\mathcal{M}_i$ will move to $(s_i, \delta)^i$. Then, by reading ?, each $\mathcal{M}_i$ will move to $(s_i, done)^{-i}$. Finally, by the Decrementing Counter Completeness lemma (Lemma 15), there is a word w_3 such that, after reading w_3, each $\mathcal{M}_i$ will move to $(0, k-1)^{-i}$, which is the final state of $\mathcal{M}_i$. Moreover, at the end of reading w_3, the counter value of $\mathcal{M}_0$ will be 0. Hence, the word w is accepted by each $\mathcal{M}_i$.

Now, suppose there is a word w that is accepted by each $\mathcal{M}_i$. By construction of $\mathcal{M}_i$, it follows that w has to be of the form $w_1!w_2?w_3$ for some w_1, w_2, w_3. Furthermore, for each i, the words w_1, w_2 and w_3 are read entirely in the parts of $\mathcal{M}_i$ corresponding to A_i, B_i and C_i respectively. Now, we note the following.

- For each i, the letter ! can only be read from a done state of the machine A_i. This means that the collection of runs of $A_0, \ldots, A_{k-1}$ on the word w_1 must be a perfect collection. Let $(s_i, done)^{+i}$ be the state visited by A_i after reading w_1. By the Incrementing Counter Soundness lemma (Lemma 10), the value of the counter of A_0 at the end of w_1 is $N = \sum_{0 \leq i \leq k-1} n^i s_i$.
- For each i, reading the letter ! from $(s_i, done)^{+i}$ leads to the state $(s_i, check)^i$ of B_i. Also, the letter ? could be read only from states whose phase is δ. By the Clique Finding lemma (Lemma 13), it follows that w_2 must be of the form $S_1 S_2 S_3 S_4$ for some k-cliques S_1, S_2, S_3, S_4 such that the i^{th} node in S_1 is s_i, $S_1 \cup S_2, S_2 \cup S_3, S_3 \cup S_4$ are all $2k$-cliques and B_i, upon reading w_2 has a run from $(s_i, check)^i$ to $(s'_i, \delta)^i$ where s'_i is the i^{th} node in S_4.
- For each i, reading the letter ? from $(s'_i, \delta)^i$ leads to the state $(s'_i, done)^{-i}$ of C_i. Note that since B_i does not modify the counter values, the counter value after reading ? is still N.

  By assumption, w is accepted by each $\mathcal{M}_i$ and so this means that each C_i, starting at $(s'_i, done)^{-i}$, upon reading w_3, has a run which ends at $(0, k-1)^{-i}$. Moreover, the run for C_0 upon reading w_3 must lead to the counter value 0. By the Decrementing Counter Soundness lemma (Lemma 14), it follows that $N = \sum_{0 \leq i \leq k-1} n^i s'_i$. Since $N = \sum_{0 \leq i \leq k-1} n^i s_i$, it follows that $s_i = s'_i$ for each i and so the clique S_4 is actually S_1.

This then implies that $S_1 \cup S_2 \cup S_3$ is a $3k$-clique and so we are done. □

Running time of the reduction: Let us now analyse the running time taken by the reduction. Note that each machine in the first and third stages can be constructed in time $O(nk^2)$ where n is the number of nodes of G. Each machine in the second stage can be constructed in time $O(n^{k+1})$. Finally, putting A_i, B_i and C_i together to form $\mathcal{M}_i$ can be done in $O(nk)$ time. Since k is a constant, it then follows that the total time taken to construct each $\mathcal{M}_i$ is $O(n^{k+1})$.

Note that the number of states of each machine is bounded by $O(nk)$ and our alphabet size is bounded by $O(n^k k)$. Since k is a constant, it follows that if we can solve the $\text{PDA} \cap \text{NFA}^{k-1}$ problem (even when the given machines are only DOCA and DFAs) in time $O((E^{(\omega-1)k}|\Sigma| + E^{\omega k})^{1-\varepsilon})$, where E is the maximum number of states among all the given machines and ε is any number strictly bigger than 0, then we can solve the $3k$-clique problem in time $O(n^{\omega k-\varepsilon})$, which would contradict the $3k$-clique hypothesis. The same argument proves that combinatorial algorithms cannot solve the $\text{PDA} \cap \text{NFA}^{k-1}$ problem in time $O(\max(E^{2k}|\Sigma|, E^{3k}))^{1-\varepsilon})$ for any $\varepsilon > 0$, unless the $3k$-clique problem can be solved in time $O(n^{3k-\varepsilon})$ by a combinatorial algorithm. Hence, Theorem 2 follows.

APPENDIX B
PROOF OF THEOREM 5

Let $3(k-1)$ be a fixed number and G be a graph over the nodes $\{0, \ldots, n-1\}$ without self-loops. The high-level construction of the machines $\mathcal{M}_0, \ldots, \mathcal{M}_{k-1}$ has already been discussed in the proof idea in the main part of the paper and here we concentrate on the formal aspects. We will construct the machine incrementally by using gadgets, each of which will correspond to one specific sub-part or part mentioned in the proof idea. Then we will finally put together all the sub-parts and parts to get the final machines.

As mentioned in the proof idea, first we will use a linear-sized alphabet and then describe how to replace it with one that is of constant size. To this end, the input alphabet will be $\{0, \ldots, n-1\} \cup \{\overline{0}, \overline{1}, \ldots, \overline{n-1}\} \cup \{\#, @\}$ and the stack alphabet will be $\{0, \ldots, n-1\}$ for all of the gadgets that we will construct.

As mentioned before, each machine $\mathcal{M}_i$ will be constructed by first constructing gadgets and then composing them together in a specific manner. We will be doing this quite often (corresponding to each sub-part as well as the prologue and the epilogue) and hence it can become quite repetitive. However, this act of composing together gadgets is uniform throughout and hence we define it formally here, so that it can be reused (repeatedly) in the construction.

Gadgets and their composition: For the purposes of this construction, a gadget to us will simply be any machine (PDA or NFA) A whose states are of the form (x, c, s) where $x \in \{0, \ldots, n-1\} \cup \{\Diamond\}, 0 \leq c \leq \max(5, k)$ and $s \in S_A$ where S_A is some set called the *auxiliary set*. We will often denote an element (x, c, s) with $s \in S_A$ as $(x, c)^s$, which will be called the superscript notation.

For a gadget A, its value is the largest value c such that a state of the form $(x, c)^s$ appears in A. The value of A will be denoted by $val(A)$. A copy of A is another gadget B which is exactly the same as the gadget A obtained by renaming the set S_A to some fresh set S_B.

Initial states of a gadget A can only be states of the form $(x, 0)^s$ for some x and s. Final states of a gadget A can only be states of the form $(x, val(A))^s$ for some x and s. We will always have the constraint that for any x, there is **exactly one** initial and final state whose first entry is x. Hence, given x, we can abuse notation, and, for example, say that we consider the initial (resp. final) state x of a gadget A to mean the unique initial (resp. final) state of A that has x as its first entry.

We say that we compose a finite sequence of gadgets $A_1, A_2, \ldots, A_\ell$ to get another gadget B if B is constructed from $A_1, \ldots, A_\ell$ by taking all of their states and transitions and adding the following new transitions: For each $i < \ell$, from each final state x of A_i we add a transition to the initial state x of A_{i+1} which reads the input letter @. The initial (resp. final) states of B will be the initial (resp. final) states of A_1 (resp. A_ℓ). The composition B intuitively corresponds to first executing A_1, then A_2 and so on all the way till A_ℓ.

Having stated all the necessary definitions regarding gadgets, we now move on to the prologue construction.

1) Prologue: We now describe the gadgets for the prologue, i.e., we will construct a PDA P_0 and $k-1$ many NFAs $P_1, \ldots, P_{k-1}$ corresponding to the construction mentioned in the prologue.

Each gadget P_i will have as its states (x, c, p_i) where $x \in \{0, \ldots, n-1\} \cup \{\Diamond\}$ and $c \in \{0, \ldots, k\}$. Note that in this case the auxiliary set of P_i is simply $\{p_i\}$ and hence each state (in the superscript notation) is of the form $(x, c)^{p_i}$.

The transitions of P_0 are as follows: Upon reading some node z from a state $(\Diamond, c)^{p_0}$ with $c < k-1$, it will push z into the stack and move to $(\Diamond, c+1)^{p_0}$. Intuitively, this gadget simply pushes $k-1$ nodes into the stack.

The transitions of each P_i with $i > 0$ are as follows: Upon reading some node z from a state $(x, c)^{p_i}$ with $c < k-1$, it will move to $(x, c+1)^{p_i}$ if $c \neq i-1$ or if $c = i-1$ and $x = z$. Intuitively, the machine P_i will not really do anything until the i^{th} letter is read (which must be a node) and when that happens, it will check that that node is exactly the node stored in its state.

Recall our convention that for each element in $x \in \{0, \ldots, n-1\} \cup \{\Diamond\}$, there will be exactly one initial and final state whose first entry is x. (In the case of P_i, the initial state by convention is $(x, 0)^{p_i}$ and the final state by convention is $(x, k-1)^{p_i}$). Now, from the construction of the gadgets, we can deduce the following theorem.

Theorem 17 (The Prologue Theorem)**.** *Let w be some word. Then, for all i, there is a run of P_i on w between some initial state x_i and some final state x_i' if and only if $w = x_1' x_2' \ldots x_{k-1}'$, $x_0 = x_0' = \Diamond$ and for each $i > 0$, $x_i = x_i'$ is a node. Moreover, in any such collection of runs, the only*

change in the stack of P_0 is the pushing of $x'_1, \ldots, x'_{k-1}$ (in that order).

Proof. Let us prove the right-to-left implication by constructing a run for each P_i. Indeed, the machine P_0 upon reading x'_c moves from $(\Diamond, c-1)^{p_0}$ to $(\Diamond, c)^{p_0}$ and pushes x'_c onto the stack. The machine P_i upon reading x_c works as follows: If $c < i$, it moves from $(x_i, c-1)^{p_i}$ to $(x_i, c)^{p_i}$. If $c = i$, since $x_i = x'_i$, it can move from $(x_i, c-1)^{p_i}$ to $(x'_i, c)^{p_i}$. If $c > i$, it moves from $(x'_i, c-1)^{p_i}$ to $(x'_i, c)^{p_i}$. This completes the desired construction.

Let us now prove the other direction. Suppose for each i, there is a run of P_i on w as promised. Note that any transition from any state of the form $(x, c)^{p_i}$ takes it to a state of the form $(x', c+1)^{p_i}$ for some x' (Further in the case of P_0, if x is $\Diamond$, then x' is $\Diamond$ as well). It then follows that $|w| = k-1$ and $x_0 = x'_0 = \Diamond$.

Note that there are no transitions reading $\#, @$ and hence each letter of w is a node. Now, the observation in the previous paragraph means that, for each $1 \leq i, c \leq k$, the machine P_i, before and after reading the c^{th} letter of w, will be in states of the form $(x, c-1)^{p_i}$ and $(x', c)^{p_i}$ for some x, x'. By definition of the transitions, if $c \neq i$, then $x = x'$ and if $c = i$ then $x = x' = x_i$, where x_i is the i^{th} letter of w. It then follows that if $(x', k-1)^{p_i}$ is the state with which P_i completes its run, then $x' = x = x_i$.

Note that while reading the word w, P_0 simply pushes each letter onto the stack. It then follows that the only change in the stack during the entire run of P_0 is the pushing of $x'_1, x'_2, \ldots, x'_{k-1}$, in that order. This completes the proof. □

Finally, we also note the following observation, which follows immediately from the construction given above.

Proposition 18 (Size of P_i)**.** *The number of states in each P_i is $O(nk)$ and each P_i can be constructed in time $O(nk)$.*

2) Part 1: Now we will describe the gadgets for the first part by first designing gadgets for each of the sub-parts of the first part.

Sub-Part I: The Setup: Here, we will construct the gadgets necessary to push copies of nodes into the stack. These gadgets are similar to the gadgets from the Prologue with a minor difference.

For each $i \in \{0, \ldots, k-1\}, j \in \{2, \ldots, k-1\}$ we will construct a machine A_i^j, which will have as its states $(x, c)^{a_i}$ for $x \in \{0, \ldots, n-1\} \cup \{\Diamond\}$ and $c \in \{0, \ldots, j-1\}$. Now the transitions of these machines are as follows.

For each $j \in \{2, \ldots, k-1\}$, the machine A_0^j is a PDA, which upon reading some node z from a state of the form $(\Diamond, c)^{a_i}$ with $c < j-1$, pushes z into the stack and moves to $(\Diamond, c+1)$. Intuitively, this machine simply pushes $j-1$ nodes into the stack.

For each $i \in \{1, \ldots, k-1\}, j \in \{2, \ldots, k-1\}$, the machine A_i^j is an NFA, which upon reading some node z from a state of the form $(x, c)^{a_i}$ with $c < j-1$, moves to $(x, c+1)$ if $i \neq j$ or $i = j$ and $x = z$. Intuitively, the machine A_i^j does not really do anything unless $i = j$. If $i = j$, it will simply check that the input consists of exactly $j-1$ letters all of which are exactly the same node that it had stored in its state at the beginning.

The following lemma is easy to see from the construction of the machines. Its proof is similar to the proof of the Prologue theorem.

Lemma 19. *Let w be some word and let $j \in \{2, \ldots, k-1\}$. Then, for all i, there is a run of A_i^j on w starting from some initial state x_i and ending at some final state x'_i if and only if $w = x_j^{j-1}$, $x_0 = x'_0 = \Diamond$ and for each $i > 0$, $x_i = x'_i$ is a node. Moreover, in any such collection of runs, the only change in the stack of P_0 is the pushing of $j-1$ copies of x_j.*

Now, for each i, let us compose the gadgets $A_i^2, A_i^3, \ldots, A_i^k$ to get a new gadget A_i. The following theorem now follows by using the definition of composition and repeatedly applying the above lemma.

Theorem 20 (The Setup Theorem)**.** *Let w be some word. Then, for all i, there is a run of A_i on w starting from some initial state x_i and ending at some final state x'_i if and only if $w = x_2@x_3^2@\ldots@x_{k-1}^{k-2}$, $x_0 = x'_0 = \Diamond$ and for each $i > 0$, $x_i = x'_i$ is a node. Moreover, in any such collection of runs, the only change in the stack of P_0 is the pushing of $x_2 x_3^2 \ldots x_{k-1}^{k-2}$ (in that order).*

Proof. Let us first prove the right-to-left implication. Suppose w is of the form $x_2@x_3^2@\ldots@x_k^{k-1}$, $x_0 = x'_0 = \Diamond$ and for each $i > 0$, $x_i = x'_i$ is a node. For each i, we can now construct a run of A_i on w starting from the initial state x_i and ending at the final state x'_i as follows. Let $w_i = x_i^{i-1}$. By the previous lemma, for each A_i^j with $j \in \{2, \ldots, k\}$, there is a run of w_i starting from the initial state x_i of A_i^j and ending with the final state x_i of A_i^j. By using the definition of composition of the gadgets, it follows that we have a run of the desired form for A_i.

Let us now prove the other direction. Suppose, for all i, there is a run of A_i on w starting from some initial state x_i and ending at some final state x'_i. By definition of composition of gadgets, w must be of the form $w_2@w_3@\ldots@w_k$ such that for each $j \in \{2, \ldots, k\}$, A_i^j has an accepting run on w_i. Now, using the definition of composition and the previous lemma, we can conclude this side of the implication.

□

Finally, we observe that

Proposition 21 (Size of A_i)**.** *The number of states in each A_i is $O(nk^2)$ and each A_i can be constructed in time $O(nk^2)$.*

Sub-Part II: The Check: Here, we will construct the gadgets necessary to check the neighborhood relation between nodes. Given a node stored at the top of the stack and another node which is stored in some NFA, the purpose of these gadgets is to check whether these two nodes are neighbors. The intuition behind these gadgets has been presented in the proof idea section in the main part of the paper and so we concentrate only on the formal aspects here.

For each $i \in \{0, \ldots, k-1\}, j \in \{1, \ldots, k-1\}$, we will construct a machine B_i^j which will have as its states $(x, c)^{b_i^j}$ for $x \in \{0, \ldots, n-1\} \cup \{\Diamond\}$ and $c \in \{0, 1, 2, 3, 4, 5\}$. Now the transitions of these machines are as follows.

For each $j \in \{1, \ldots, k-1\}$, the machine B_0^j is a PDA which upon reading some letter ℓ from a state of the form $(x, c)^{b_i^j}$ does the following:

- If $x = \Diamond$, $c = 0$ and ℓ is a node, then it moves to $(\ell, 1)^{b_i^j}$.
- If $c = 1$ and $\ell = \#$, then it moves to $(x, 2)^{b_i^j}$.
- If $c = 2$ and $\ell = \overline{y}$ for some node y, then it moves to $(x, 3)^{b_i^j}$ only if it can pop ℓ from the stack.
- If $c = 3$ and $\ell = \#$, then it moves to $(x, 4)^{b_i^j}$.
- If $c = 4$ and $\ell = x$ is a node, then it moves to $(\Diamond, 5)^{b_i^j}$.

For each $i \in \{1, \ldots, k-1\}, j \in \{1, \ldots, k-1\}$, the machine B_i^j is an NFA which upon reading some letter ℓ from a state of the form $(x, c)^{b_j^i}$ with x a node does the following: If $i \neq j$ and $c < 5$, then it moves to $(x, c+1)^{b_i^j}$. If $i = j$, then

- If $c = 0$ and $\ell = x$ is a node, then it moves to $(x, 1)^{b_i^j}$.
- If $c = 1$ and $\ell = \#$, then it moves to $(x', 2)^{b_i^j}$ where x' is some neighbor of x in G.
- If $c = 2$ and $\ell = \overline{x}$, then it moves to $(x, 3)^{b_i^j}$.
- If $c = 3$ and $\ell = \#$, then it moves to $(x', 4)^{b_i^j}$ where x' is some neighbor of x in G.
- If $c = 4$ and $\ell = x$ is a node, then it moves to $(x, 5)^{b_i^j}$.

Now, for any two nodes x, y let $N(x, y)$ be the word $x\#\overline{y}\#x$. The following lemma follows from an analysis of the constructed gadgets.

Lemma 22. *Let w be some word and let $j \in \{1, \ldots, k-1\}$. Then, for all i, there is a run of B_i^j on w starting from some initial state x_i and ending at some final state x_i' if and only if x_j is a node and there is a neighbor z of x_j such that $w := N(x_j, z)$, z is at the top of the stack of P_0 at the beginning, $x_0 = x_0' = \Diamond$ and for each $i > 0$, $x_i = x_i'$ is a node. Moreover, in any collection of runs, the only change in the stack of P_0 is the popping of z.*

Proof. Let us first prove the right-to-left implication. In this case, first notice that all the machines B_i^j except for B_0^j and B_j^j simply have the run $(x_i, 0)^{b_i^j}, (x_i, 1)^{b_i^j}, (x_i, 2)^{b_i^j}, (x_i, 3)^{b_i^j}, (x_i, 4)^{b_i^j}, (x_i, 5)^{b_i^j}$ on the word w. The machine B_0^j upon reading the subword $x_j\#$ has the run $(\Diamond, 0)^{b_0^j}, (x_j, 1)^{b_0^j}, (x_j, 2)^{b_0^j}$. From there it reads $\overline{z}$ and since z is at the top of the stack, B_0^j pops it and moves to $(x_j, 3)^{b_0^j}$. From there it reads $\#x_j$ and moves to $(x_j, 4)^{b_0^j}, (\Diamond, 5)^{b_0^j}$. Similarly, the machine B_j^j upon reading w has the run $(x_j, 0)^{b_j^j}, (x_j, 1)^{b_j^j}, (z, 2)^{b_j^j}, (z, 3)^{b_j^j}, (x_j, 4)^{b_j^j}, (x_j, 5)^{b_j^j}$. Hence, this direction of the claim is true.

Let us now prove the other direction. Note that for any $i \notin \{0, j\}$, any run of B_i^j must be of the form $(x_i, 0)^{b_i^j}, (x_i, 1)^{b_i^j}, \ldots, (x_i, 5)^{b_i^j}$ with x_i being a node. Hence $x_i = x_i'$ is true for $i \notin \{0, j\}$.

Now let us analyze the runs of B_0^j on B_j^j on the word w. By construction, it is easy to see that $|w| = 5$. Hence $w = a_1a_2a_3a_4a_5$ for some letters a_1, a_2, a_3, a_4, a_5. Note that if $a_1 \neq x_j$ or x_j is not a node, then there is no transition from $(x_j, 0)^{b_j^j}$ labelled by a_1. Hence, $a_1 = x_j$, x_j is a node and the machines B_0^j and B_j^j move to the states $(x_j, 1)^{b_0^j}$ and $(x_j, 1)^{b_j^j}$ respectively. Now, if $a_2 \neq \#$, then there are no transitions from either of these states. Hence, $a_2 = \#$ and B_0^j and B_j^j move to the states $(x_j, 2)^{b_0^j}$ and $(z', 2)^{b_j^j}$ for some z' which is a neighbor of x_j. For the same reasons as above, a_3 must be $\overline{z'}$. Furthermore, B_0^j attempts to pop the input letter from the stack and so a_3 must also be equal to $\overline{z}$. Hence, $z = z'$ and so z and x_j are neighbors. The machines now move to the states $(x_j, 3)^{b_0^j}$ and $(z, 3)^{b_j^j}$. Now a_4 must be $\#$ and the machines now move to the states $(x_j, 4)^{b_0^j}$ and $(x', 4)^{b_j^j}$ for some neighbor x' of z. Finally, a_5 must be x_j and so x' must also be equal to x_j and then the machines move to $(\Diamond, 5)^{b_0^j}$ and $(x_j, 5)^{b_j^j}$, thereby completing the proof. □

Now, for each j, first create $k-j-1$ many copies of B_i^j and call them $B_i^{j,k-1}, \ldots, B_i^{j,j+1}$. Then, for each i, we compose all of these gadgets in the following order to get the gadget B_i: $B_i^{1,k-1}, B_i^{2,k-1}, \ldots, B_i^{k-2,k-1}, B_i^{1,k-2}, B_i^{2,k-2}, \ldots, B_i^{k-3,k-2}, B_i^{1,k-3}, \ldots, B_i^{1,2}$.

The intuition behind the composition is as follows: Assume that before we start executing the gadgets $B_0, \ldots, B_k$, the stack of B_0, from the top, contains the word $x_{k-1}^{k-2}x_{k-2}^{k-3}\ldots x_3^2x_2$ for some nodes $x_2, \ldots, x_{k-1}$ and each NFA B_i stores the node x_i (Note that this is guaranteed to us by sub-part I). Then, we execute each B_i, which first executes $B_i^{1,k-1}$ which ensures that x_1 and x_{k-1} are neighbors and pops the topmost x_{k-1}. Then, B_i executes $B_i^{2,k-1}$ which ensures that x_2 and x_{k-1} are neighbors and pops the next x_{k-1} and so on till $B_i^{k-2,k-1}$, which will ensure that x_{k-1} is a neighbor of every other node. At this point, the topmost part of the stack contains $k-3$ copies of x_{k-2}. Now, B_i will start executing $B_i^{1,k-2}$ which will ensure that x_1 and x_{k-2} are neighbors, then $B_i^{2,k-2}$, which will ensure that x_2 and x_{k-2} are neighbors and so on all the way up till $B_i^{1,2}$. Hence, in this way we would have ensured that $x_1, x_2, \ldots, x_{k-1}$ is a $(k-1)$-clique. This intuition is made more precise by the next paragraph.

For any $k-1$ nodes $x_1, \ldots, x_{k-1}$, let $N(x_1, \ldots, x_{k-1})$ be the word

$$N(x_1, x_{k-1})@N(x_2, x_{k-1})@\ldots N(x_{k-2}, x_{k-1})@N(x_1, x_{k-2})$$
$$@N(x_2, x_{k-2})\ldots N(x_{k-3}, x_{k-2})\ldots N(x_1, x_2)$$

The following theorem follows by applying the definition of composition to the machines B_i along with the previous lemma. Its proof is similar to the proof of the Setup Theorem.

Theorem 23 (The Check Theorem)**.** *Let w be some word. Then, for each i, there is a run of B_i on w starting from some initial state x_i and ending at some final state x_i' if and only if for each $i > 0$, $x_i = x_i'$ is a node, $w = N(x_1, \ldots, x_{k-1})$, the word $x_{k-1}^{k-2}, x_{k-2}^{k-3}\ldots, x_2$ is at the top of the stack of P_0 at the beginning, $x_0 = x_0' = \Diamond$ and $x_1, \ldots, x_{k-1}$ is a $(k-1)$-clique.*

Moreover, in any such collection of runs, the only change in the stack of B_0 is the popping of $x_{k-1}^{k-2}, x_{k-2}^{k-3}, \ldots, x_2$.

We conclude with a discussion on the number of states and time taken to construct each B_i. Each B_i is obtained by composing $O(k^2)$ many gadgets, each of which have $O(n)$ states and each of which can be constructed in $O(|G|)$ time. Hence, it follows that

Proposition 24 (Size of B_i)**.** *The number of states in each B_i is $O(nk^2)$ and each B_i can be constructed in $O(|G|k^2)$ time.*

Sub-Part III: The Exploration: Here, we will construct the gadgets necessary to find $k-1$ more nodes $y_1, \ldots, y_{k-1}$ such that each node in the collection $x_1, \ldots, x_{k-1}$ (which was checked to be a $(k-1)$-clique in sub-part II), is a neighbor of each node in the collection $y_1, \ldots, y_{k-1}$.

We will create four different types of gadgets for this sub-part and then compose them all together. All these gadgets will be similar to the gadgets that we have seen in the previous sub-parts (and the prologue), but with slight modifications. We will now describe the first type of gadgets, whose intuitive purpose will be to simply push some node into the stack $k+1$ times.

For each $i \in \{0, \ldots, k-1\}$, we will create a machine D_i, which will have as its states $(x, c)^{d_i}$ for $x \in \{0, \ldots, n-1\} \cup \{\Diamond\}$ and $c \in \{0, \ldots, k\}$.

The machine D_0 is a PDA which upon reading some node z from a state of the form $(x, c)^{d_0}$ with $c < k$, pushes z into the stack and

- Moves to $(z, c+1)^{d_0}$ if $x = \Diamond$ and $c = 0$ or if $x = z$ and $c > 0$.
- Moves to $(\Diamond, c+1)^{d_0}$ if $x = z$ and $c = k-1$.

Intuitively, this gadget remembers the first node that is read and then ensures that the next $k-1$ input letters are the same as the first node. After ensuring this, it forgets the node that it remembered. Further, each time it reads a node it simply pushes it into the stack. Now, we move on to the description of the NFAs.

For each $i \in \{1, \ldots, k-1\}$, the machine D_i is an NFA which upon reading some node z from a state of the form $(x, c)^{d_i}$ with $c < k-1$ moves to $(x, c+1)$. Intuitively, these gadgets do nothing except continue to remember the node that they were remembering before.

The following lemma follows immediately from the construction of these gadgets. Its proof is similar to the proof of the Prologue theorem, but even easier.

Lemma 25. *Let w be some word. Then, for each i, there is a run of D_i on w starting from some initial state x_i and ending at some final state x_i' if and only if $w = y^k$ for some node y, $x_0 = x_0' = \Diamond$ and for each $i > 0$, $x_i = x_i'$ is a node. Moreover, in any such collection of runs, the only change in the stack of D_0 is the pushing of k copies of y.*

The second type of gadgets that we will construct will use the D_i's that we just constructed and the B_i^j's from sub-part II. To this end, for each i, we will construct a gadget E_i by composing the gadgets $D_i, B_i^1, B_i^2, \ldots, B_i^{k-1}$. Intuitively, the idea is that each NFA E_i will initially begin with some node stored in its state. Then the gadget D_i will serve the purpose of pushing k many copies of some node y onto the stack. Then, B_i^1 will ensure that this node y is a neighbor of the node stored in the state of E_1, B_i^2 will ensure that y is a neighbor of the node stored in the state of E_2 and so on. Let us now state this formally.

Recall that for any two nodes x, y we had defined the word $N(x, y) = x\#\overline{y}\#x$. Given any node y and $k-1$ many nodes $x_1, \ldots, x_{k-1}$ let $E(y, x_1, \ldots, x_{k-1})$ be the word $y^k @ N(x_1, y) @ N(x_2, y) @ \ldots @ N(x_{k-1}, y)$. From the definition of composition of gadgets and from the lemmas that we have proved regarding the gadgets D_i and B_i^j the following lemma follows.

Lemma 26. *Let w be some word. Then, for each i, there is a run of E_i on w starting from some initial state x_i and ending at some final state x_i' if and only if for each $i > 0$, $x_i = x_i'$ is a node, there is a node y that is a neighbor of each x_i such that $w = E(y, x_1, \ldots, x_{k-1})$ and $x_0 = x_0' = \Diamond$. Moreover, in any such collection of runs, the stack content at the end is exactly the same as the beginning, except for an additional single copy of y at the top.*

Before we move on to the third type of gadgets, we note the size and time taken to construct each E_i. Each E_i is obtained by composing $D_i, B_i^1, \ldots, B_i^{k-1}$. D_i has $O(nk)$ states and can be constructed in $O(nk)$ time. Each B_i^j has $O(n)$ states and can be constructed in $O(|G|)$ time. It follows that

Proposition 27 (Size of E_i)**.** *Each E_i has $O(nk)$ states and can be constructed in $O(nk + |G|)$ time.*

The third type of gadgets is obtained by simply repeating the second type for $k-1$ times. Formally, we create a gadget F_i, by first creating $k-1$ copies of E_i and then composing them all together. Note that we immediately have the following theorem.

Theorem 28. *Let w be some word. Then, for each i, there is a run of F_i on w starting from some initial state x_i and ending at some final state x_i' if and only if for each $i > 0$, $x_i = x_i'$ is a node, there are nodes $y_1, \ldots, y_{k-1}$ that are all neighbors of each x_i, $w = E(y_1, x_1, \ldots, x_{k-1}) @ E(y_2, x_1, \ldots, x_{k-1}) @ \ldots @ E(y_{k-1}, x_1, \ldots, x_{k-1})$ and $x_0 = x_0' = \Diamond$. Moreover, in any such collection of runs, the stack content at the end is exactly the same as the beginning, except for an additional single copy of the word $y_{k-1} \ldots y_1$ at the top.*

We will now describe the fourth type of gadgets. Their job is to pop the topmost $k-1$ letters on the stack and store them in the NFAs. To this end, for each $i \in \{0, \ldots, k-1\}$, we will create a machine H_i, which will have as its states $(x, c)^{h_i}$ for $x \in \{0, \ldots, n-1\} \cup \{\Diamond\}$ and $c \in \{0, \ldots, k-1\}$.

The machine H_0 is a PDA which upon reading some $\overline{z}$ from a state of the form $(\Diamond, c)^{h_0}$ with $c < k$, pops z from the stack and moves to $(\Diamond, c+1)^{h_0}$. Intuitively, this gadget forces the $k-1$ input letters to be the topmost $k-1$ letters in the stack.

For each $i \in \{1, \ldots, k-1\}$, the machine H_i is an NFA which upon reading some $\overline{z}$ from a state of the form $(x, c)^{h_i}$ with $c < k$ moves to $(x, c+1)^{h_i}$ if $c \neq k-1-i$ and otherwise moves to $(z, c+1)^{h_i}$. Intuitively, this gadget remembers the $(k-i+1)^{th}$ input letter that is read in a sequence of k input letters.

From the construction, we can now prove the following lemma. The proof is similar to the proof of the Prologue theorem.

Lemma 29. *Let w be some word. Then, for each i, there is a run of H_i on w starting from some initial state x_i and ending at some final state x'_i if and only if $w = \overline{y_{k-1}} \ldots \overline{y_1}$ for some nodes $y_{k-1}, \ldots, y_1$, the word w is at the top of the stack of H_0 at the beginning, $x_0 = x'_0 = \Diamond$ and for each $i > 0$, $x'_i = y_i$. Moreover, in any such collection of runs, the only change in the stack is the popping of w.*

Now, let us create the final gadget for this sub-part. For each i, we compose the gadget F_i and H_i to get a new gadget L_i. From the statements for the gadgets F_i and H_i, we get the following theorem.

Theorem 30 (The Exploration Theorem)**.** *Let w be some word. Then, for each i, there is a run of L_i on w starting from some initial state x_i and ending at some final state y_i if and only if for each $i > 0$, x_i and y_i are nodes, $w = E(y_1, x_1, \ldots, x_{k-1}) @ E(y_2, x_1, \ldots, x_{k-1}) @ \ldots @ E(y_{k-1}, x_1, \ldots, x_{k-1}) @ \overline{y_{k-1}} \ldots \overline{y_1}$, each node in $\{y_1, \ldots, y_{k-1}\}$ is a neighbor of each node in $\{x_1, \ldots, x_{k-1}\}$ and $x_0 = y_0 = \Diamond$. Moreover, in any such collection of runs, the stack content at the end is exactly the same as the beginning.*

We conclude with a discussion on the number of states and time taken to construct each L_i. Each L_i is obtained by composing F_i and H_i. Each F_i has $O(k)$ copies of E_i and so by Proposition 27, it follows that each F_i has $O(nk^2)$ states. and that F_i can be constructed in $O(nk^2 + |G|k)$ time. From the construction of H_i, note that each H_i has $O(nk)$ states and can be constructed in $O(nk)$ time. Hence, it follows that

Proposition 31 (Size of H_i)**.** *Each L_i has $O(nk^2)$ states and can be constructed in $O(nk^2 + |G|k)$ time.*

This completes the third sub-part.

Now, we wrap up by combining the gadgets from the three sub-parts into one big gadget. To this end, for each i, we will create a gadget X_i by composing A_i, B_i and L_i. For any collections of nodes $x_1, \ldots, x_{k-1}$ and $y_1, \ldots, y_{k-1}$ we will define three words given by $A(x_1, \ldots, x_{k-1}) = x_1 x_2 \ldots x_{k-1}, B(x_1, \ldots, x_{k-1}) = x_2 @ x_3^2 @ x_4^3 @ \ldots @ x_{k-1}^{k-2}$ and $L(x_1, \ldots, x_{k-1}, y_1, \ldots, y_{k-1}) = E(y_1, x_1, \ldots, x_{k-1}) @ E(y_2, x_1, \ldots, x_{k-1}) @ \ldots @ E(y_{k-1}, x_1, \ldots, x_{k-1}) @ y_{k-1} \ldots y_1$. Note that by the theorems that we have proved in this section, any word that is accepted by all of the A_i's (resp. all of the B_i's or all of the L_i's) must be of the form $A(x_1, \ldots, x_{k-1})$ (resp. $B(x_1, \ldots, x_{k-1})$ or $L(x_1, \ldots, x_{k-1}, y_1, \ldots, y_{k-1})$). With this intuition, we then define $W(x_1, \ldots, x_{k-1}, y_1, \ldots, y_{k-1}) = A(x_1, \cdot, x_{k-1}) @ B(x_1, \cdot, x_{k-1}) @ L(x_1, \cdot, x_{k-1}, y_1, \cdot, y_{k-1})$.

Intuitively, the gadget X_i first executes A_i which begins with a collection of $k-1$ nodes $x_1, \ldots, x_{k-1}$ and then sets up the stack in a specific way. Then it executes B_i which verifies that these nodes $x_1, \ldots, x_{k-1}$ form a $(k-1)$-clique. Finally, it executes L_i which finds another collection of nodes $y_1, \ldots, y_{k-1}$ such that every node in this new collection is a neighbor of every node in $\{x_1, \ldots, x_{k-1}\}$.

Now, from the Setup Theorem, the Check Theorem and the Exploration Theorem, we have the following main result.

Theorem 32 (End of Part Theorem)**.** *Let w be some word. Then, for each i, there is a run of X_i on w starting from some initial state x_i and ending at some final state y_i if and only if for each $i > 0$, x_i and y_i are nodes, $w = W(x_1, \ldots, x_{k-1}, y_1, \ldots, y_{k-1})$, $x_1, \ldots, x_{k-1}$ is a $(k-1)$-clique, each node in $\{y_1, \ldots, y_{k-1}\}$ is a neighbor of each node in $\{x_1, \ldots, x_{k-1}\}$ and $x_0 = y_0 = \Diamond$. Moreover, in any such collection of runs, the stack content at the end is exactly the same as the beginning.*

We conclude this part by a discussion on the size of X_i. By propositions 21,24 and 31, it follows that

Proposition 33 (Size of X_i)**.** *Each X_i has $O(nk^2)$ states and can be constructed in $O(nk^2 + |G|k)$ time.*

Second and Third Parts: The second and third parts are exactly like the first part. More precisely, for each i, we create two more copies of X_i and call them Y_i and Z_i. Y_i and Z_i are the gadgets for the second and the third part.

Intuitively, the first part X_i begins with a collection of nodes $x_1, \ldots, x_{k-1}$, verifies it to be a $(k-1)$-clique and then finds another collection $y_1, \ldots, y_{k-1}$ such that each node in this new collection is a neighbor of each node in $\{x_1, \ldots, x_{k-1}\}$, then it ends with the nodes y_i stored in the state of each of the NFAs X_i. Now, if we compose Y_i with X_i, Y_i will verify that $y_1, \ldots, y_{k-1}$ is a $(k-1)$-clique and then find another collection of nodes $z_1, \ldots, z_{k-1}$ such that each node in this new collection is a neighbor of each node in $\{y_1, \ldots, y_{k-1}\}$. Furthermore, each NFA Y_i will end with the node z_i stored in its state. Hence, if we now compose Z_i with it, we can verify that $\{z_1, \ldots, z_{k-1}\}$ is a $(k-1)$-clique and also find another collection of nodes $\{x'_1, \ldots, x'_{k-1}\}$ such that each node in this new collection is a neighbor of each node in $\{z_1, \ldots, z_{k-1}\}$. If we then check that each $x'_i = x_i$ (which will be done by the Epilogue gadget) then we are guaranteed that the x_i's, y_i's and z_i's together form a $3(k-1)$-clique. Hence, we will first compose the X_i',s Y_i's and Z_i's together and then finally compose the Prologue and Epilogue gadgets.

For each i, let M_i be the gadget obtained by composing X_i, Y_i and Z_i. By the End of Part theorem applied thrice we get the following.

Theorem 34 (Main sub-parts Theorem)**.** *Let w be some word. Then, for each i, there is a run of M_i on w starting from some initial state x_i and ending at some final state x'_i if and only if*

- *For each $i > 0$, x_i and x'_i are nodes and there exists nodes $y_1, \ldots, y_{k-1}, z_1, \ldots, z_{k-1}$ such that $w = W(x_1, \ldots, x_{k-1}, y_1, \ldots, y_{k-1})@W(y_1, \ldots, y_{k-1}, z_1, \ldots, z_{k-1})@W(z_1, \ldots, z_{k-1}, x'_1, \ldots, x'_{k-1})$.*
- *The sets $X := \{x_1, \ldots, x_{k-1}\}, Y := \{y_1, \ldots, y_{k-1}\}$ and $Z := \{z_1, \ldots, z_{k-1}\}$ are all $(k-1)$-cliques.*
- *Every node in X is connected to every node in Y, every node in Y is connected to every node in Z and every node in Z is connected to every node in $X' = \{x'_1, \ldots, x'_{k-1}\}$.*
- *$x_0 = x'_0 = \Diamond$.*

Moreover, in any such collection of runs, the stack content at the end is exactly the same as the beginning.

This completes the main three parts of the reduction. Now we move on to the Epilogue.

Epilogue: Recall that the Prologue gadget pushes in $k-1$ nodes $x_1, \ldots, x_{k-1}$ into the stack. If we start the gadget M_i after the Prologue, then by the Main sub-parts theorem, it will end with some node x'_i in each of its NFAs and the stack content will remain the same at the end as it was in the beginning, i.e., it will have the $k-1$ nodes $x_{k-1}, \ldots, x_1$. Hence, to check if each $x'_i = x_i$, we only need to pop the stack one element at a time and check that the i^{th} element popped is the node stored in the $(k-i)^{th}$ NFA. Equivalently, it suffices to check that the $(k-i)^{th}$ element popped is the node stored in the i^{th} NFA. This is what the Epilogue gadget will accomplish now.

Formally, for each $i \in \{0, \ldots, k-1\}$, we will construct a gadget P_i^r, which will have as its states $(x, c)^{p_i^r}$ for $x \in \{0, \ldots, n-1\} \cup \{\Diamond\}$ and $c \in \{0, \ldots, k-1\}$.

The transitions of P_0^r are as follows: Upon reading some $\overline{z}$ from a state $(\Diamond, c)^{p_0^r}$ with $c < k-1$, it will pop z from the stack and move to $(\Diamond, c+1)^{p_0^r}$. Intuitively, this gadget simply pops $k-1$ nodes from the stack.

The transitions of each P_i^r with $i > 0$ are as follows: Upon reading some $\overline{z}$ from a state $(x, c)^{p_i^r}$ with $c < k-1$, it will move to $(x, c+1)^{p_i^r}$ if $c \neq k-i-1$ or $c = k-i-1$ and $x = z$. Intuitively, the gadget P_i^r ensures that the $(k-i)^{th}$ letter read is the same as the one stored in its state.

From the construction of the gadgets, the following theorem immediately follows.

Theorem 35 (Epilogue Theorem)**.** *Let w be some word. Then, for each i, there is a run of P_i^r on w between some initial state x'_i and some final state x_i if and only if $w = x'_{k-1} \ldots x'_1$, $x'_0 = x_0 = \Diamond$, for each $i > 0$, $x'_i = x_i$ is a node and the stack of P_0^r at the beginning of the run contains the word w at the top. Moreover, in any such collection of runs, the only change in the stack of P_0^r is the popping of the word w.*

Note that we immediately get the following bound on the size of P_i^r.

Proposition 36 (Size of P_i^r)**.** *Each P_i^r has $O(nk)$ states and can be constructed in $O(nk)$ time.*

Now, it is time to construct the final machines $\mathcal{M}_i$, which we can obtain by composing P_i, M_i and P_i^r. From the Prologue, Main sub-parts and the Epilogue Theorems, we get the following theorem, which establishes the correctness of the reduction.

Theorem 37 (Correctness of the Reduction)**.** *Let w be some word. Then, for each i, there is an accepting run of $\mathcal{M}_i$ on w if and only if there exists a $3(k-1)$-clique in the graph G.*

By propositions 18, 33 and 36, it follows that the number of states of each $\mathcal{M}_i$ is $O(nk^2)$ and each $\mathcal{M}_i$ can be constructed in $O(nk^2 + |G|k)$ time. Since k was a fixed constant to begin with, it follows that

Proposition 38. *Each $\mathcal{M}_i$ has $O(n)$ states and can be constructed in $O(|G|)$ time.*

We also note that the PDA $\mathcal{M}_0$ is deterministic, in the sense that, for every state q and every letter a it has at most one outgoing transition from q labelled by a. Note that this can be naturally converted into a complete machine that has exactly one outgoing transition for each letter, by adding sink states. However, for the sake of brevity, we do not do it here.

A. Reducing the alphabet size

Now, we will show how to reduce the input and stack alphabet size in the machines $\mathcal{M}_i$ so that it becomes one of constant size. The intuitive idea has already been sketched before in the proof idea section in the main part of the paper and so we focus on the formal aspects here.

Note that in both the input and the stack alphabet, if we remove the set of nodes $\{0, \ldots, n-1\} \cup \{\overline{0}, \ldots, \overline{n-1}\}$, then we are left with a constant-sized alphabet. (In the case of the stack alphabet, it is in fact exactly equal to $\{0, \ldots, n-1\}$ and hence the PDA only pushes/pops nodes into the stack). Hence, we only need to "compress" these nodes. To this end, for any node ℓ, let msbf(ℓ) (resp. lsbf(ℓ)) be the most significant bit first encoding of ℓ (resp. least significant bit first encoding of ℓ) over $\{0, 1\}$. In this way, we can represent each node ℓ by $\log n$ sized words over $\{0, 1\}$.

Now, we modify the machines $\mathcal{M}_0, \ldots, \mathcal{M}_k$ so that instead of reading nodes, they read $\log n$ sized words over the alphabet $\{0, 1\}$ and interpret them as nodes. A naive way of doing this would be to introduce a $\log n$ sized gadget for each transition of each machine which replaces reading a node with reading its corresponding $\log n$ sized msbf encoding. Furthermore, whenever it wants to push some node into the stack, it pushes the input as it is read, i.e., it pushes the msbf encoding and whenever it wants to pop some node from the stack, it pops the lsbf encoding of that node.

However, doing this the naive way would increase the number of states by $O(m \log n)$ (with m being the number of edges of G) which might be quadratic in n. It turns out that by modifying this naive idea a bit more, we can arrive at the desired machines with just an extra $O(n \log n)$ states. The modification is simply to combine all the "naive gadgets" going between any two pair of states into one "smart gadget", which saves a lot of states and allows us to reuse the gadgets.

We now describe this formally by first making a series of observations regarding the gadgets that we have made.

Let p be some state in some machine $\mathcal{M}_i$. Recall that p is a 3-tuple, the first of which is either some node of the graph G or the symbol $\Diamond$.

Now, by examining all the gadgets that we have constructed, we have the following first observation.

> Suppose there is some outgoing transition from p that reads a for some node a. Then, all the outgoing transitions only read letters from the set $\{y : 0 \leq y \leq n-1\}$, i.e., they only read nodes of the graph G. Further, one of the following always applies:
>
> - Either, for each node y of the graph, there is exactly one outgoing transition from p reading y. Furthermore, all such outgoing transitions lead to the same state.
> - Or p stores some node x and there is exactly one outgoing transition from p reading some node a. Furthermore, a must be x.
> - Or p stores $\Diamond$ and for each node y of the graph, there is exactly one outgoing transition from p reading y.
>
> Moreover, if the underlying machine is the PDA $\mathcal{M}_0$, then
>
> - Either all the outgoing transitions from p do not change the stack.
> - Or all the outgoing transitions from p push the input letter onto the stack.

Our second observation is a dual to the above observation for letters of the form $\overline{a}$.

> Suppose there is some outgoing transition from p that reads $\overline{a}$ for some node a. Then, all the outgoing transitions only read letters from the set $\{\overline{y} : 0 \leq y \leq n-1\}$. Further, one of the following always applies:
>
> - Either, for each node y of the graph, there is exactly one outgoing transition from p reading $\overline{y}$. Furthermore, all such outgoing transitions lead to the same state.
> - Or p stores some node x and there is exactly one outgoing transition from p reading some node $\overline{a}$. Furthermore, a must be x.
> - Or p stores $\Diamond$ and for each node y of the graph, there is exactly one outgoing transition from p reading $\overline{y}$.
>
> Moreover, if the underlying machine is the PDA $\mathcal{M}_0$, then every outgoing transition from p reading a letter of the form $\overline{y}$, pops the node y from the stack.

Intuitively, these observations mean that whenever p stores a node of the graph G, then we do not have to introduce a separate gadget for every outgoing transition of p reading a or $\overline{a}$ for some node a. Instead, we can club together all these gadgets into one gadget that goes to the same state q. Hence, for "most" states, we only need a $\log n$ sized gadget. Regarding the other states, they all store $\Diamond$ and there are only constantly many of them (where we use the fact that k is a constant) and so we can afford to introduce a separate gadget for each such state, which will only lead to an increase of $O(n \log n)$ states overall. We now move on to the formal aspects.

Let p be some state of some machine $\mathcal{M}_i$. Suppose there is at least one outgoing transition of p that reads some node a. We now consider each of the three cases given by the first observation.

Case 1: In this case, for each node y, we have exactly one outgoing transition reading y of the graph and all of these outgoing transitions move to the same state q (and we have no other outgoing transitions). Intuitively, this means that it does not matter which node is being read as long as we are sure that the input being read is indeed a node (and not # or @). With this in mind, we can replace all such outgoing transitions with the following gadget between p and q. First, we add $\log n$ many states $p := (p,0), (p,1), \ldots, (p, \log n) = q$ and then from state (p,i) we move to $(p, i+1)$ upon reading either 0 or 1. Furthermore, if the machine $\mathcal{M}_i$ is $\mathcal{M}_0$, i.e., the PDA, then by the first observation

- Either all the outgoing transitions from p in $\mathcal{M}_0$ do not alter the stack, in which case in the new gadget as well no stack operations are performed.
- Or all outgoing transitions from p in $\mathcal{M}_0$ push the input node that is read into the stack, in which case, in the new gadget as well we push the input letter that is read, i.e., while moving between (p,i) and $(p,i+1)$ we push either 0 or 1 if the input letter that is read is either 0 or 1 respectively. This ensures that every action of pushing a node into the stack is replaced by the action of pushing its msbf representation into the stack.

Case 2: In this case, p stores some node x and there is exactly one outgoing transition of p and this transition reads the node x. Let q be the state to which this transition goes to. Intuitively, this means that we can only read the node x from this state and so exactly one gadget between p and q suffices here. With this in mind, we can replace this outgoing transition with the following gadget between p and q. First, we add $\log n$ many states $p := (p,0), (p,1), \ldots, (p, \log n) = q$ and then from state (p,i) we move to $(p, i+1)$ upon reading the $(i+1)^{th}$ bit in the msbf encoding of x. This ensures that any way of using this gadget would be forced to read the msbf encoding of x.

Furthermore, if the machine $\mathcal{M}_i$ is $\mathcal{M}_0$, i.e., the PDA, then

- Either the outgoing transition from p in $\mathcal{M}_0$ does not alter the stack, in which case in the new gadget as well no stack operations are performed.
- Or the outgoing transition from p in $\mathcal{M}_0$ pushes the input node x into the stack, in which case, in the new gadget as well we push the input letter that is read.

Case 3: In this case, p stores $\Diamond$. By the observation above, for each node y, there is exactly one outgoing transition from p reading y. Let q_y be state to which this transition goes to.

We now attach a gadget between p and q_y by adding $\log n$ many states $p := (p,0)^y, (p,1)^y, \ldots, (p,\log n)^y := q_y$ and then from state $(p,i)^y$ we move to $(p,i+1)^y$ upon reading the $(i+1)^{th}$ bit in the msbf encoding of y.

Furthermore, if the machine $\mathcal{M}_i$ is $\mathcal{M}_0$, i.e., the PDA, then

- Either the outgoing transition reading y from p in $\mathcal{M}_0$ does not alter the stack, in which case in the new gadget as well no stack operations are performed.
- Or the outgoing transition reading y from p in $\mathcal{M}_0$ pushes y into the stack, in which case, in the new gadget as well we push the input letter that is read.

Now, suppose there is some outgoing transition from p that reads some letter of the form $\overline{a}$ for some node a. Intuitively, the gadgets here are exactly the same as the ones before, except they will read the lsbf encoding of the node, rather than the msbf encoding.

Formally, we now consider each of the three cases given by the second observation.

Case 1: In this case, let q be the unique state to which all outgoing transitions of p go to. We add $\log n$ many states $p := (p,0), (p,1), \ldots, (p,\log n) = q$ and then from state (p,i) we move to $(p,i+1)$ upon reading either 0 or 1. Furthermore, if the machine $\mathcal{M}_i$ is $\mathcal{M}_0$, i.e., the PDA, then by the second observation, all outgoing transitions from p in $\mathcal{M}_0$ pop the input node that is read into the stack, in which case, in the new gadget as well we pop the input letter that is read. This ensures that every action of popping a node into the stack is replaced by the action of popping its lsbf representation into the stack.

Case 2: In this case, let p store some node x and let q be the unique state to which the (only) outgoing transition of p reading $\overline{x}$ moves to. We add $\log n$ many states $p := (p,0), (p,1), \ldots, (p,\log n) = q$ and then from state (p,i) we move to $(p,i+1)$ upon reading the $(i+1)^{th}$ bit in the lsbf encoding of x. Furthermore, if the machine $\mathcal{M}_i$ is $\mathcal{M}_0$, i.e., the PDA, then by the second observation, the outgoing transition from p in $\mathcal{M}_0$ pops x from the stack, in which case, in the new gadget as well we pop the input letter that is read.

Case 3: In this case, for each node y, let q_y be the state that p moves to upon reading $\overline{y}$. Between p and q_y we introduce $\log n$ many states $p := (p,0)^y, (p,1)^y, \ldots, (p,\log n)^y = q_y$ and then from state $(p,i)^y$ we move to $(p,i+1)^y$ upon reading the $(i+1)^{th}$ bit in the lsbf encoding of y. Furthermore, if the machine $\mathcal{M}_i$ is $\mathcal{M}_0$, i.e., the PDA, then by the second observation, this outgoing transition from p in $\mathcal{M}_0$ pops y from the stack, in which case, in the new gadget as well we pop the input letter that is read.

This completes our transformation. Call the new machines as $\mathcal{M}'_0, \mathcal{M}'_1, \ldots, \mathcal{M}'_{k-1}$. Given some word γ over the nodes, let $mbin(\gamma)$ (resp. $lbin(\gamma)$) denote the word obtained by replacing each node in γ with its msbf (resp. lsbf) representation. The following proposition is immediate from construction.

Proposition 39. *Let a be some node.*

- *There is a step of the form $(p,\gamma) \xrightarrow{a} (q,\eta)$ (resp. $(p,\gamma) \xrightarrow{\overline{a}} (q,\eta)$) in $\mathcal{M}_0$ for some states p,q and some stack content γ,η if and only if there is a run of the form $(p, lbin(\gamma)) \xrightarrow{mbin(a)} (q, lbin(\eta))$ (resp. $(p, lbin(\gamma)) \xrightarrow{lbin(a)} (q, lbin(\eta))$) in $\mathcal{M}'_0$.*
- *There is a step of the form $p \xrightarrow{a} q$ (resp. $p \xrightarrow{\overline{a}} q$) in $\mathcal{M}_i$ for some $i > 0$ and some states p,q if and only if there is a run of the form $p \xrightarrow{mbin(a)} q$ (resp. $p \xrightarrow{lbin(a)} q$) in $\mathcal{M}'_i$.*

Using this proposition it is then easy to see that

Theorem 40. *There is a word accepted by all of the machines $\mathcal{M}_0, \mathcal{M}_1, \ldots, \mathcal{M}_{k-1}$ if and only if there is a word accepted by all of the machines $\mathcal{M}'_0, \mathcal{M}'_1, \ldots, \mathcal{M}'_{k-1}$.*

Now, note that for each state p that does not store $\Diamond$, we have added only $\log n$ more states. As for the states that do store $\Diamond$, we have added $n \log n$ more states. However, note that the number of states that store $\Diamond$ in each of the gadgets presented in each of the parts is a constant (depending only on k). By Proposition 38, it follows then that the total number of states in each $\mathcal{M}'_i$ is $O(n \log n)$. Further, it also follows that each machine $\mathcal{M}'_i$ can be constructed in time $O(|G| + n \log n)$. It can also be verified that the PDA $\mathcal{M}'_0$ is deterministic, once again, in the sense that for every state q and every letter a, there is at most one outgoing transition from q labelled by a.

Now, suppose we can solve the $\mathrm{PDA} \cap \mathrm{NFA}^{k-1}$ problem in time $O(n^{\omega(k-1)-\varepsilon})$ for some $\varepsilon > 0$. Then, by doing the above construction in $O(|G| + n \log n)$ time, we can reduce the $3(k-1)$-clique problem to the $\mathrm{PDA} \cap \mathrm{NFA}^{k-1}$ problem such that all the machines in the instance have $O(n \log n)$ states. It then follows that we could solve the original $3(k-1)$-clique instance in time $O(n \log n + |G| + n^{\omega(k-1)-\varepsilon} (\log n)^{\omega(k-1)-\varepsilon})$. Since $(\log n)^{\omega(k-1)-\varepsilon}$ grows asymptotically slower than $n^{\varepsilon/2}$ for any $\varepsilon > 0$, it follows that we can solve the given $3(k-1)$-clique instance in time $O(n^{\omega(k-1)-\varepsilon+\varepsilon/2}) = O(n^{\omega(k-1)-\varepsilon/2})$, which is a contradiction to the $3(k-1)$-clique hypothesis. Similarly, we can argue for an $O(n^{3(k-1)})$ lower bound for combinatorial algorithms. Theorem 5 now follows.

APPENDIX C
2NPDA(k) AND LINEAR TIME EQUIVALENCES

A. Formal Description of 2NPDA(k) Machines

A 2NPDA(k) machine [23], [26] has the form $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q is a finite set of states, Σ are Γ are finite alphabets of input and stack symbols, respectively, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and a transition relation $\delta \subseteq Q \times \Sigma^k \times \Gamma \times Q \times \Gamma^* \times \{-1, 0, +1\}^k$. We assume Σ contains two designated "end of tape" symbols $\lhd$ and $\rhd$, and that a head cannot move left if it reads $\lhd$ and cannot move right it is reads $\rhd$. We assume that Γ contains a designated "end of stack" symbol Z_0 such that any transition $(q, \vec{a}, Z_0, q', \gamma, d) \in \delta$ satisfies $\gamma = Z_0$. Thus, no transition of $\mathcal{A}$ replaces Z_0 on the stack with a different symbol; also, no transition pushes Z_0 on the stack when the top of the stack is not Z_0.

Informally, the 2NPDA(k) machine $\mathcal{A}$ has a finite control (states from Q) and k reading heads. In a single transition, the machine simultaneously reads k symbols (elements of Σ) from the input tape and the top symbol (an element of Γ) from the pushdown store. Based on the transition relation δ, 2NPDA moves by changing the control state, replacing the top symbol of the pushdown store by a finite string of symbols (possibly the empty string), and moving each of its input heads at most one symbol left or right (some heads can remain at the same position).

Initially, the machine is in state q_0, and its pushdown store consists of the single symbol Z_0. The input tape consists of a word $w \in (\Sigma \setminus \{\triangleleft, \triangleright\})^*$ surrounded by a left marker $\triangleleft$ and a right marker $\triangleright$, and every head of the machine scans the left marker $\triangleleft$.

A configuration of the 2NPDA $\mathcal{A}$ is a triple $(q, w_1\sigma_1 w_2 \dots \sigma_k w_{k+1}, \gamma)$, where $q \in Q$, $w_i \in \Sigma^*$ for each $i \in \{1, \dots, k+1\}$, $\sigma_1 \dots \sigma_k$ is a permutation of $\{1, \dots, k\}$, and $\gamma \in \Gamma^+$. (In this description, we assume that $\{1, \dots, k\} \cap \Sigma = \emptyset$.) The configuration represents the situation where the state is q, the word $w_1 \dots w_{k+1}$ is on the input tape, the k heads are at positions on the input tape that are preceded in the configuration by σ_1 to σ_k. In other words, if $\sigma_i = \ell$, then the ℓth head of $\mathcal{A}$ observes the first letter of the word $w_{i+1} \dots w_{k+1}$. Intuitively, w_{i+1} is the word on the input tape between the cell observed by the ith leftmost head and the cell observed by the $i+1$st leftmost head (excluding the latter). Notice that $w_1 = \varepsilon$ if and only if one if the heads observes the leftmost cell; w_{k+1} is never empty.

We write $c = (q', s, \gamma Z) \to (q', s', \gamma\gamma') = c'$ whenever there is a transition $(q, \vec{a}, Z, q', \gamma', \vec{d}) \in \delta$ with the following properties. In c each head, i, observes some letter a_i such that $\vec{a} = (a_1, \dots, a_k)$. In c' each head moves by d_i positions, where $\vec{d} = (d_1, \dots, d_k)$, compared to c. We require that the scan position does not "fall off" the input word. Note that the input tape is not changed, only the scan position may change. We write $\to^*$ for the reflexive and transitive closure of $\to$.

The initial configuration of $\mathcal{A}$ on a word $w \in (\Sigma \setminus \{\triangleleft, \triangleright\})^*$ is the configuration $(q_0, \sigma_1\sigma_2 \dots \sigma_k \triangleleft w \triangleright, Z_0)$ where each $\sigma_i = i$. An accepting configuration of $\mathcal{A}$ on a word $w \in (\Sigma \setminus \{\triangleleft, \triangleright\})^*$ is a configuration of the form $(q, \triangleleft w \sigma_1 \sigma_2 \dots \sigma_k \triangleright, Z_0)$ where $q \in F$ and each $\sigma_i = i$. A *run* of the 2NPDA $\mathcal{A}$ on a word $w \in (\Sigma \setminus \{\triangleleft, \triangleright\})^*$ is a sequence of configurations $C_0, C_1, \dots, C_k$ such that C_0 is the initial configuration of $\mathcal{M}$ on w and each $C_i \to C_{i+1}$. A run is said to be accepting if the last configuration of that run is an accepting configuration. A word $w \in (\Sigma \setminus \{\triangleleft, \triangleright\})^*$ is *accepted* by $\mathcal{A}$ if there is an accepting run of $\mathcal{A}$ on the word w. The language $L(\mathcal{A})$ of $\mathcal{A}$ is the set of all accepted words (in $(\Sigma \setminus \{\triangleleft, \triangleright\})^*$).

B. 2NPDA(k) language recognition reduces to DCFL k-intersection reachability for DFA

Let $\mathcal{M}$ be a 2NPDA(k). For every $w \in \Sigma^*$, we show how to construct a DPDA P and DFA $A_1, \dots, A_k$ such that

$$w \in \mathcal{L}(\mathcal{M}) \qquad \text{iff} \qquad \mathcal{L}(P) \cap \mathcal{L}(A_1) \cap \dots \cap \mathcal{L}(A_k) \neq \emptyset.$$

In our construction, the DPDA P will be independent of the input word w and determined solely by the automaton $\mathcal{M}$; thus, the DCFL L from the theorem statement is chosen as $L = \mathcal{L}(P)$. The DFA $A_1, \dots, A_k$ will be constructed in time linear in w.

The tape alphabet of the machines P, $A_1, \dots, A_k$ is the set δ of transitions of the 2NPDA(k) $\mathcal{M}$. The language $\mathcal{L}(P) \cap \mathcal{L}(A_1) \cap \dots \cap \mathcal{L}(A_k)$ will consist of all accepting runs of $\mathcal{M}$ on input w. Indeed, a sequence $\rho = t_1 \dots t_n \in \delta^*$ is an accepting run if and only if the following three conditions are satisfied:

- Transitions of ρ trace a path in the finite graph on the states of $\mathcal{M}$ from the initial state to a final state.
- Stack movements prescribed by the sequence ρ are valid, that is, the sequence of pushes and pops belongs to the Dyck language over the alphabet $\Gamma \cup \Gamma^{-1}$, where Γ is the stack alphabet of $\mathcal{M}$.
- For each $i \in [1, k]$, the letters on the input tape that are read by the ith head in the sequence ρ are compatible with the input tape containing word $\triangleleft w \triangleright$, where $\triangleleft$ and $\triangleright$ are endmarker symbols.

In short, the DPDA P checks the first two conditions, and each of the DFA $A_1, \dots, A_k$ checks the third condition for some i. To complete the proof, it remains to describe the construction of the machines $P, A_1, \dots, A_k$ and to analyse the running time of the reduction.

Construction of DPDA P and DCFL L: The deterministic pushdown automaton P is determined by the 2NPDA(k) $\mathcal{M}$. The set of its control states is equal to that of $\mathcal{M}$, call it Q. Every state $q_1 \in Q$ has outgoing transitions labelled with all input letters of the form $(q_1, a, Z, q_2, \gamma', d) \in \delta$. This way, the first condition in the list above is checked by P. On letter $(q_1, a, Z, q_2, \gamma', d)$, the DPDA P pops $Z \in \Gamma$ from the top of its stack, replacing it with $\gamma' \in \Gamma^*$. This way, the second condition in the list above is also checked by P. In short, for each $t = (q_1, a, Z, q_2, \gamma', d) \in \delta$ the DPDA P has a transition $(q_1, t, Z, q_2, \gamma')$. The DPDA P ignores the symbol $a \in \Sigma$ and the head movements d prescribed by t.

To complete the description of P, we choose the initial state and the set of final states the same as they are in $\mathcal{M}$. As already announced, $L = \mathcal{L}(P)$ is the sought DCFL.

Construction of DFA $A_1, \dots, A_k$: Let $i \in [1, k]$. The deterministic finite automaton A_i verifies the third condition in the list above for i. The set of control states of A_i is $\{0, 1, \dots, |w|+1\}$, which we think of as possible positions of the ith head of $\mathcal{M}$ over the input tape. The initial state is 0, and the only final state is $|w|+1$, in line with the semantics of 2NPDA(k).

Reading $t = (q_1, a, Z, q_2, \gamma', d) \in \delta$ from its input tape, the DFA A_i ignores all components except $a \in \Sigma$ and $d \in \{-1, 0, +1\}^k$. In fact, in the vector $d = (d_1, \dots, d_k)$ only d_i is relevant, and all other components are ignored too. From state j, where $1 \leq j \leq |w|$, only transitions where a is equal to the ith letter of the word w can depart. Similarly, for $j = 0$ and $j = |w|+1$, the requirement is that $a = \triangleleft$ and $a = \triangleright$, respectively. Upon reading t as above, the automaton A_i moves to the state $j + d_i$ as long as this state exists.

Running time of the reduction: The reader will readily see that P and L are determined solely by $\mathcal{M}$. The DFA $A_1, \ldots, A_k$ depend on w and have $|w| + 2$ states each. Their input alphabet is δ, which is again independent of w, and so all of these machines can be constructed in linear time given w.

C. CFL k-intersection reachability reduces to $\mathrm{PDA} \cap \mathrm{NFA}^{k-1}$ intersection non-emptiness

Let the CFL L be fixed. Given k NFA, the (L, k) intersection reachability problem asks if there is a word in $\mathcal{L}(A_1) \cap \ldots \cap \mathcal{L}(A_k)$ that belongs to L.

The reduction takes some fixed PDA P_0 for the CFL L and produces a PDA P for the language $L \cap \mathcal{L}(A_1) = \mathcal{L}(P_0) \cap \mathcal{L}(A_1)$ by utilising the standard product construction (see, e.g., Hopcroft, Motwani, and Ullman's textbook [25, Section 7.3.4]). The set of control states of PDA P is the Cartesian product of the sets of control states of P_0 and A_0. Since P_0 is fixed, the description size of P is linear in the description size of A_1.

The reduction outputs the PDA P and NFA $A_2, \ldots, A_k$, which together form the input of $\mathrm{PDA} \cap \mathrm{NFA}^{k-1}$ intersection non-emptiness. The correctness and running time analysis of the reduction are immediate.

We remark that, if L is a DCFL and A_1 is a DFA, then the product PDA P is in fact a DPDA [20, Theorem 3.1].

D. $\mathrm{PDA} \cap \mathrm{NFA}^{k-1}$ intersection non-emptiness reduces to 2NPDA(k) language recognition

The input to the $\mathrm{PDA} \cap \mathrm{NFA}^{k-1}$ intersection non-emptiness problem is a concatenation of the string encoding the PDA P and strings encoding the NFA $A_1, \ldots, A_{k-1}$, with delimiters separating one from another. The encodings use a fixed alphabet, which we denote by Δ; then the input is some $w \in \Delta^*$. In particular, all letters of the input alphabet of $P, A_1, \ldots, A_{k-1}$, denoted by Σ, and of the stack alphabet of P, denoted by Γ, are encoded by words from Δ^*. We describe a fixed 2NPDA(k) $\mathcal{M}$ that accepts $w \in \Delta^*$ if and only if w encodes some PDA P and NFA $A_1, \ldots, A_{k-1}$ that accept some word in common: $\mathcal{L}(P) \cap \mathcal{L}(A_1) \cap \ldots \cap \mathcal{L}(A_{k-1}) \neq \emptyset$.

It will be clear that the description of 2NPDA(k) $\mathcal{M}$ only depends on k but not on PDA P or NFA $A_1, \ldots, A_{k-1}$. The reduction is linear: in fact, it is just a matter of encoding the list $P, A_1, \ldots, A_{k-1}$ as a word $w \in \Delta^*$.

The idea is for $\mathcal{M}$ to guess a word in this intersection on the fly and to simulate k accepting runs on this word in lockstep: one in the PDA P and $k-1$ in the NFA $A_1, \ldots, A_{k-1}$. In a nutshell, $\mathcal{M}$ uses one of its heads for each of these accepting runs, and the stack for storing the content of the stack of the PDA P. The pseudocode in Figure 1 (page 12) summarises the construction and is meant to be seen as the program of the (fixed) 2NPDA(k) $\mathcal{M}$. For the convenience of notation, we denote $A_0 = P$. We now detail each step of the simulation.

Initialising the stack: The description of the PDA A_0 contains the encoding of its bottom-of-stack symbol as a string over Δ, which is separated from other parts of the input word w of $\mathcal{M}$ by delimiters (also coming from Δ). At the beginning of the simulation, $\mathcal{M}$ locates this encoding using one of its heads and pushes it onto the stack. The head returns to the left endmarker.

Positioning head i to the initial state of A_i: As already mentioned, $\mathcal{M}$ uses one head per each of the machines $A_0, A_1, \ldots, A_{k-1}$. To keep track of the current control state of A_{i-1}, the ith head of $\mathcal{M}$ is used. We can assume that the string encoding the automaton A_{i-1} — be that PDA P or NFA A_{i-1} — includes a list of the states of A_{i-1}, in which the initial state comes first. The program of 2NPDA(k) $\mathcal{M}$ moves the ith head right from the left endmarker, skipping encodings of $i-1$ automata completely, and stopping over the first element in the list of control states of the ith automaton, A_{i-1}.

Guessing and executing transitions: This phase of the simulation consists of the three **for** loops in the pseudocode above, as well as two nondeterministic **while** loops handling ε-transitions in the PDA A_0. We discuss the **for** loops first.

In the first loop, moving each head of $\mathcal{M}$ to an outgoing transition within the encoding of A_{i-1} is non-deterministic: t_{i-1} is guessed. The implementation is self-explanatory, except for the following detail. As an invariant of the simulation, we require that, in between iterations of the main **while** loop in the pseudocode, head i of $\mathcal{M}$ is positioned over (the encoding of) the current control state of A_{i-1}, call it q_{i-1}, within the list of all states of A_{i-1}. When the next transition t_{i-1} of A_{i-1} is guessed, the 2NPDA(k) $\mathcal{M}$ must check that t_{i-1} departs from q_{i-1}. To this end, $\mathcal{M}$ first pushes the encoding of q_{i-1} on the stack. It then moves head i to the encoding of t_{i-1}, thus guessing t_{i-1}. At this point $\mathcal{M}$ pops from the stack to check the match of the control state. If the check fails, the nondeterministic branch rejects.

In the second loop, the goal is to ensure that the guessed transitions $t_0, t_1, \ldots, t_{k-1}$ all read the same input letter $a \in \Sigma$ from the input. Recall that letters of the alphabet Σ are encoded by words over Δ. (In fact, this is why it is not necessarily possible to guess a upfront and store it in the control state of $\mathcal{M}$.) To perform the check, each head of $\mathcal{M}$ locates the encoding of the input letter within the description of the corresponding transition t_{i-1}. The heads then move in synchrony to check equality of the letters. As above, if the check fails, the nondeterministic branch of the computation rejects.

In the third loop, the transitions $t_0, t_1, \ldots, t_{k-1}$ are executed:

- For NFA $A_1, \ldots, A_{k-1}$, it suffices, using head i, to push the encoding of the destination of the transition t_{i-1} onto the stack, then locate the list of control states of A_{i-1} and guess the position of the destination in that list. After that, the stack is popped to compare the destination as recorded on the stack (which is popped) with the state in the list, ensuring the invariant of the simulation.
- For PDA A_0, we also need to simulate the operations on the stack. Recall that the semantics of a PDA transition dictates that a stack symbol $Z \in \Gamma$ be popped from the

top of the stack and replaced by a word $\gamma' \in \Gamma^*$. Again as previously, letters of the stack alphabet Γ are encoded using words from Δ^*. To perform the stack operations, the 2NPDA(k) $\mathcal{M}$ locates the encoding of Z and starts popping the stack, checking that the symbols match the encoding of Z. If the check fails, the nondeterministic branch of computation rejects (because the guessed transition is not available from the current configuration). Otherwise $\mathcal{M}$ proceeds to push the encoding of γ'. After these stack operations, $\mathcal{M}$ goes on to update the current control state, as in the case of NFA.

The **for** loops discussed above ensure that all of $A_0, A_1, \ldots, A_{k-1}$ synchronise on the input letters, i.e., in effect $\mathcal{M}$ guesses k sequences of transitions that form k accepting runs. However, unlike the NFA $A_1, \ldots, A_{k-1}$, the PDA A_0 may have ε-transitions. These are taken care of by the two nondeterministic **while** loops: with the help of head 0, $\mathcal{M}$ can simulate an arbitrary sequence of ε-transitions taken by A_0 before and after Σ-transitions.

Checking acceptance: When $\mathcal{M}$ guesses the end of the word in $\mathcal{L}(A_0) \cap \mathcal{L}(A_1) \cap \ldots \cap \mathcal{L}(A_{k-1})$, it pushes the encoding of the current control states of $A_0, A_1, \ldots, A_{k-1}$ onto the stack and then moves the heads to locate these states in the corresponding lists of final states in the input word $w \in \Delta^*$. The stack is popped to verify that all these states are indeed final. After that, one of the heads locates the encoding of the bottom-of-stack symbol of A_0 within w. By popping the stack, $\mathcal{M}$ verifies that the simulated stack of A_0 contains this symbol only and, therefore, that A_0 has reached an accepting configuration. If all checks succeed, $\mathcal{M}$ accepts.