frontiers | Frontiers in Computer Science

ORIGINAL RESEARCH
published: 07 April 2022
doi: 10.3389/fcomp.2022.845990

Uniform Polylogarithmic Space Completeness

Flavio Ferrarotti[1]*, *Senén González*[1], *Klaus-Dieter Schewe*[2] and *José María Turull-Torres*[3]

[1] Software Competence Center Hagenberg, Hagenberg, Austria, [2] University of Illinois at Urbana-Champaign Institute (UIUC), Zhejiang University, Haining, China, [3] DIIT, Department of Engineering and Technological Research, Universidad Nacional de La Matanza, Buenos Aires, Argentina

It is well-known that polylogarithmic space (PolyL for short) does not have complete problems under logarithmic space many-one reductions. Thus, we propose an alternative notion of completeness inspired by the concept of uniformity studied in circuit complexity theory. We then prove the existence of a uniformly complete problem for PolyL under this new notion. Moreover, we provide evidence that uniformly complete problems can help us to understand the still unclear relationship between complexity classes such as PolyL and polynomial time.

Keywords: reductions, completeness, polylogarithmic space, PolyL, complexity theory

OPEN ACCESS

Edited by:
Daowen Qiu,
Sun Yat-sen University, China

Reviewed by:
Gokarna Sharma,
Kent State University, United States
Maria Alessandra Ragusa,
University of Catania, Italy

***Correspondence:**
Flavio Ferrarotti
flavio.ferrarotti@scch.at

Specialty section:
This article was submitted to Theoretical Computer Science, a section of the journal Frontiers in Computer Science

Received: *30 December 2021*
Accepted: *16 March 2022*
Published: *07 April 2022*

Citation:
Ferrarotti F, González S, Schewe K-D and Turull-Torres JM (2022) Uniform Polylogarithmic Space Completeness. Front. Comput. Sci. 4:845990. doi: 10.3389/fcomp.2022.845990

1. INTRODUCTION

The class of problems that can be decided by deterministic Turing machines using space bounded by a polynomial in the logarithm of the input size, is known in computational complexity as polylogarithmic space and usually denoted as PolyL (see e.g., Papadimitriou, 1994). Same as we know that every problem in L (logarithmic space) is in P (polynomial time), we have that every problem in PolyL is in QP (quasi-polynomial time). This latter class represents algorithms which run in sub-exponential time, more precisely in time bounded by $2^{O(\log^c n)}$ for inputs of size n and some fixed constant c, and can thus be considered somehow more tractable than exponential time algorithms. Interestingly, the fastest known algorithm for checking graph isomorphism belongs to QP (see Babai, 2016).

It follows from the well-known space hierarchy theorem of Hartmanis et al. (1965) that the subset of problems in PolyL with space bounded above by $\log^c n$ is strictly included in the subset bounded above by $\log^{c+1} n$ for all integers $c > 0$. Therefore, PolyL cannot have a complete problem under logarithmic space many-one reductions. Note that if PolyL had such a complete problem A, then the space complexity of deciding this problem must be bounded above by $\log^c n$ for some constant c, i.e., it must be in DSPACE($\log^c n$) for some fixed c. If we now take a problem B in DSPACE($\log^{c+1} n$) \ DSPACE($\log^c n$), we get (by our assumption on the completeness of A) that there is a Turing machine that decides B in space $O(\log^c n)$, as B could be reduced to A using logarithmic space and then decided in $O(\log^c n)$. This means that $B \in$ DSPACE($\log^c n$), contradicting that by the space hierarchy theorem DSPACE($\log^c n$) $\subset$ DSPACE($\log^{c+1} n$).

Even though PolyL does not have complete problems under logarithmic space many-one reductions, and thus can be considered less robust than L (its logarithmic counterpart), we show in this article that this perception can be challenged. Indeed, considering an alternative notion of completeness, we are able to show that it is still possible to isolate the most difficult problems in PolyL and to draw standard consequences of the kind entailed by the classical notion of completeness.

Our alternative notion of completeness (and hardness) is grounded in the concept of uniformity borrowed from circuit complexity theory (see Balcázar et al., 1990; Immerman, 1999; Murray and Williams, 2017 among others), hence we call it *uniform completeness*. A reviewer suggested that this is also related to Ragusa (2012). The intuitive idea is to consider a countably infinite family of problems instead of a single global problem. Each problem in the family corresponds to a fragment of a same global problem determined by a positive integer parameter. Such problem is uniformly complete for a given complexity class if there is a transducer Turing machine which given a positive integer as input builds a Turing machine in the required complexity class that decides the fragment of the problem corresponding to this parameter.

The article is organized as follows. In section 2, we introduce the necessary background from complexity theory and fix the notation used throughout the article. We dedicate section 3 to examine the quantified Boolean satisfiability problem and to define a restriction of this problem which, as we show in this article, captures the power and complexity of PolyL. We present the notion of uniform completeness (and hardness) in section 4. This constitutes the main novelty of this article. Using this new concept, we then prove in section 5 the existence of a uniform complete problem for PolyL. The problem is defined using the restriction of the quantified Boolean satisfiability problem introduced earlier in section 3. We present our conclusion in section 6.

2. PRELIMINARIES

We assume the reader is familiar with deterministic Turing machines and only include here the formal definitions that are required to fix the notation. We take these formal definitions from Balcázar et al. (1995).

Definition 2.1. A *deterministic Turing machine* with r tapes is a five-tuple $M = \langle Q, \Sigma, \delta, q_0, F\rangle$, where:

- Q is the finite set of *states*.
- Σ is the finite *tape alphabet*.
- $q_0 \in Q$ is the *initial state*.
- $F \subseteq Q$ is the *set of accepting final states*.
- $\delta : Q \times \Sigma^r \to Q \times \Sigma^{r-1} \times \{R, N, L\}^r$ is the (partial) *transition function* of M.

By the previous definition, there is one tape whose content cannot be changed by the transition function. This is the input tape, that we assume is read-only. Consequently, each configuration (a.k.a. instantaneous description or snapshot) does not need to include the contents of the input tape, as the position of the input head is enough to determine the current symbol read from the input.

Definition 2.2. A *configuration* of a Turing machine M with r tapes on a fixed input is a $r+1$ tuple of the form: $(q, i, w_2, \ldots, w_r)$, where q is the current state of M, i is the position of the input tape head and each $w_j \in \Sigma^*\#\Sigma^*$ represents the current content of the j-th work tape. The symbol $\# \notin \Sigma$ marks that the tape head is reading the symbol immediately to its right. All symbols in the infinite work tape j that do not appear in w_j are assumed to be the symbol blank "$\sqcup$". In the *initial configuration* $(q_0, 0, \#, \ldots, \#)$ of M, all work tapes are blank and the input tape head is scanning the leftmost cell. An *accepting configuration* is a configuration whose state is an accepting final state.

The concept of computation can now be defined as a sequence of configurations.

Definition 2.3. A *partial computation* of a Turing machine M with r tapes on an input string $w = a_1, \ldots, a_n$ is a (possibly infinite) sequence of configurations of M, in which each step from a configuration to the next is dictated by the transition function as follows: Assume a configuration *conf* with state q, input tape head in position i and each j-th work tape head scanning a corresponding symbol b_j. If $\delta(q, a_i, b_2, \ldots, b_r) = (q', b'_2, \ldots, b'_k, m_1, \ldots, m_r)$, then the state in the next configuration is q', the position of the input tape head i' equals $i+1$, $i-1$ or i depending on whether m_1 is R, L, or N, respectively, and each string w'_j $(2 \leq j \leq r)$ representing the contents of the j-th work tape equals the corresponding string w_j in *conf* with the possible exceptions of the symbol b_j in the position immediately to the right of # in w_j and the position of # itself. The former is replaced in w'_j by b'_j. The latter is moved one position to the right if $m_j = R$, to the left if $m_j = L$, or not moved if $m_j = N$. If the # in w_j is in the leftmost position and $m_j = L$, then # remains in the same place in w'_j. Likewise, if $i = 0$ and $m_1 = L$, then the head of the input tape remains in place, i.e., $i' = i$. A *computation* is a partial computation that starts with the initial configuration of M, and ends in a configuration where no more steps can be performed.

If a computation of a Turing machine M ends in an accepting configuration, then we call it an *accepting computation* of M. In this case, the word in the input tape is *accepted* by M. The *language* $L(M)$ accepted by M is the set of all words accepted by M.

Non-deterministic Turing machines simply relax the definition of transition function δ with respect to their deterministic counterparts, so that every move is *not* necessarily determined uniquely by the current configuration.

Definition 2.4. A *non-deterministic Turing machine* with r tapes is a five-tuple $M = \langle Q, \Sigma, \delta, q_0, F\rangle$, where Q, Σ, q_0 and F are exactly as in Definition 2.1 and the transition function is defined by

$$\delta : Q \times \Sigma^r \to \mathcal{P}(Q \times \Sigma^{r-1} \times \{R, N, L\}^r)$$

where $\mathcal{P}(A)$ denotes the powerset of A.

The previous definitions of configurations and computations apply in exactly the same way to non-deterministic Turing machines. However, they can have more than one computation for a given input. Thus, a non-deterministic Turing machine M *accepts* an input string w iff there exists a computation of M on w ending in an accepting configuration.

Complexity theory mainly concerns the classification of computational problems in terms of required Turing machine

resources, and the study of the relationship between the resulting classes.

Definition 2.5. Let n denote the size of a Turing machine input, i.e., the number of non-blank cells in the input tape, and let t and s be functions such that $t(n) > n + 1$ and $s(n) \geq 1$. We define $DTIME(t)$, $NTIME(t)$ as the classes of all languages accepted by deterministic and non-deterministic Turing machines, respectively, whose running time is bounded above by $t(n)$. Similarly, We define $DSPACE(s)$, $NSPACE(s)$ as the classes of all languages accepted by deterministic and non-deterministic Turing machines, respectively, whose work space is bounded above by $s(n)$. Running time means the number of transitions in a computation. Work space is the number of different work tape cells (counting all work tapes) used during a computation.

In this work we concentrate in the class of languages accepted by deterministic Turing machines with polylogarithmic bounded space, i.e., in the class known as PolyL. Of course, we also need to reference some related complexity classes. They are formally defined as follows.

Definition 2.6. Let $\log n$ denote the logarithm base 2 of n, i.e., $\log_2 n$, and $\log^k n$ denote $(\lceil \log n \rceil)^k$, where $\lceil \log n \rceil$ is the least integer greater than or equal to $\log n$.

$$\mathrm{L} = \bigcup c \geq 1\ DSPACE(c \cdot \log n);\ \mathrm{NL} = \bigcup c \geq 1\ NSPACE(c \cdot \log n);\ \mathrm{PolyL} = \bigcup k \geq 0\ DSPACE(\log^k n)$$

$$\mathrm{P} = \bigcup k \geq 0\ DTIME(n^k);\ \mathrm{NP} = \bigcup k \geq 0\ NTIME(n^k);\ \mathrm{PSPACE} = \bigcup k \geq 0\ DSPACE(n^k)$$

It is well-known that each deterministic class is closed under complementation and that each deterministic class is included in its non-deterministic counterpart, but it is not known whether this inclusion is strict. It is also well-known that NL is closed under complementation, included in P and strictly included in PSPACE. Also, NP is included in PSPACE. All these results can be found in classical complexity theory books such as in Papadimitriou (1994) and in Balcázar et al. (1995). Obviously, NL is included in PolyL, but it is unclear how PolyL compares with P. We only know that PolyL $\neq$ P, since P has a complete problem under logarithmic space many-one reductions but polyL does not due to the space hierarchy theorem. It is not expected for polyL to be strictly contained in P. Whether the converse is true, it is also unclear.

One of the most important sort of intuitions about complexity classes and their interrelationships is provided by the concepts of reducibility and complete problem. We again borrow the definitions of these well-known concepts from Balcázar et al. (1995).

Definition 2.7. A language A is *polynomial time many-one reducible* to a language B iff there is a function $f : \Sigma^* \to \Sigma^*$, computable in polynomial time by a transducer Turing machine, such that $w \in A$ iff $f(w) \in B$ for all $w \in \Sigma^*$.

As usual, we denote that A is reducible to B by $A \leq_m B$, and if f is the function that defined this reduction, then we say that $A \leq_m B$ *via* f. Polynomial time many-one reductions are sometimes called Karp reductions. For the classes P, L and NL, Karp reductions are considered too strong, since they have complete problems *via* logarithmic space reductions. The concept of completeness provides important insight about the most difficult problems inside a complexity class.

Definition 2.8. Given a complexity class $\mathcal{C}$,

- A language A is $\mathcal{C}$-*hard* iff for any language B in $\mathcal{C}$, we have that $B \leq_m A$.
- A language A is $\mathcal{C}$-*complete* iff it is $\mathcal{C}$-hard and $A \in \mathcal{C}$.

3. A RESTRICTED QUANTIFIED BOOLEAN SATISFIABILITY PROBLEM

Our aim is to define a problem that captures the power and complexity of PolyL. We start from a well-known problem that captures these features in another deterministic space complexity class, namely the PSPACE-complete problem of determining the satisfiability of quantified Boolean sentences (QSAT for short), and explore a restriction of this problem so that it can be uniformly solved in PolyL.

Definition 3.1. Let V be a set of Boolean variables, i.e., a set of symbols that can take the value 0 (false) or 1 (true). The class of *Boolean formulae* over V is defined by the following rules:

- Constants 0 and 1 are Boolean formulae.
- If $x \in V$, then x is a Boolean formulae.
- If φ and ψ are Boolean formulae, then $(\varphi \vee \psi)$, $(\varphi \wedge \psi)$, and $\neg(\varphi)$ are Boolean formulae.
- Nothing else is a Boolean formulae.

The class of *quantified Boolean formulae* is the smallest class defined by the rules:

- Every Boolean formula is a quantified Boolean formula.
- If $x \in V$ and φ is a quantified Boolean formula, then $\exists x\varphi$ and $\forall x\varphi$ are quantified Boolean formulae.

Notice that the previous definition implies that a quantified Boolean formula is always in prenex normal form, i.e., every quantified Boolean formula consists of a (possible empty) prefix of quantifiers followed by a quantifier-free Boolean formula. This is of course not necessary, but it is convenient. We also assume w.l.o.g. that every variable that appears in a formula is quantified in the prefix at most once.

Let $\varphi[x/a]$, where $a \in \{0, 1\}$ and φ is a quantified Boolean formula, denote the formula obtained by substituting a for every occurrence of x in φ. The semantics of quantified Boolean formulae can be formally defined as follows.

Definition 3.2. Let $v : X \to \{0, 1\}$ be a Boolean assignment and φ a quantified Boolean formula, the *truth value of* φ *under* v is defined recursively by the rules:

- $v(\varphi) = 0$ if $\varphi = 0$.
- $v(\varphi) = 1$ if $\varphi = 1$.
- $v(\varphi) = v(x)$ if $\varphi = x$ for some $x \in V$.
- $v(\varphi) = v(\psi) + v(\alpha)$ (Boolean addition) if $\varphi = (\psi \vee \alpha)$.
- $v(\varphi) = v(\psi) \cdot v(\alpha)$ (Boolean multiplication) if $\varphi = (\psi \wedge \alpha)$.
- $v(\varphi) = \overline{v(\psi)}$ (Boolean complement) if $\varphi = \neg(\psi)$.
- $v(\varphi) = v(\psi[x/0]) + v(\psi[x/1])$ (Boolean addition) if $\varphi = \exists x\varphi$.
- $v(\varphi) = v(\psi[x/0]) \cdot v(\psi[x/1])$ (Boolean multiplication) if $\varphi = \forall x\varphi$.

Boolean formulae can be encoded as words over a finite alphabet and thus written down in Turing machine tapes. Here, we encode quantified Boolean formulae as words over the following alphabet:

$$\Sigma_{\text{QBF}} = \{\wedge, \vee, \neg, \forall, \exists, (,), 0, 1, \mathit{true}, \mathit{false}\}$$

where each variable is represented by the binary expression of its subindex, and *true* and *false* denote the Boolean constants.

We can now formally define the QSAT problem as well as its restriction for PolyL.

Definition 3.3. Let QBF denote the set of quantified Boolean formulae encoded as words of a fixed alphabet Σ and let $var(\varphi)$ for $\varphi \in$ QBF denote the set of variables encoded in φ.

- QSAT is the subset of QBF formed by all encodings of quantified Boolean sentences (i.e., formulae without free variables) that evaluate to "true".
- $\text{QSAT}^{pl}_k = \{\varphi \in \text{QSAT} \mid |var(\varphi)|^3 \leq \log^k |\varphi|\}$.

It is well-known that QSAT is complete for PSPACE under Karp reductions. See for instance Theorem 3.29 and its associated lemmata in Balcázar et al. (1995). Using a similar strategy plus a new concept of uniform completeness defined in the next section, we show in this article that the family of problems $\{\text{QSAT}^{pl}_k\}_{k\geq 0}$ captures the essence of the most difficult problems in PolyL.

4. UNIFORM COMPLETENESS

The new notion of completeness (and hardness) that we define in this section is grounded in the concept of uniformity borrowed from circuit complexity theory (see e.g., Balcázar et al., 1990; Immerman, 1999), hence we call it *uniform completeness*.

As a first step we need to define a notion of uniformity for Turing machines.

Definition 4.1. Let $\mathcal{M}$ be a countably infinite class of Turing machines such that for every integer $k > 0$ there is exactly one machine $M_k \in \mathcal{M}$. We say that $\mathcal{M}$ is *uniform* if there is a Turing machine $M_{\mathcal{M}}$ which for every input $k \geq 0$ builds an encoding of the corresponding $M_k \in \mathcal{M}$.

Instead of looking at isolated languages (or problems), we are concerned here with families of languages (or problems).

Definition 4.2. A *language family* $\mathcal{L}$ is a countably infinite class of languages of a same finite vocabulary. We say that $\mathcal{L}$ is *compatible* with a language A if $\bigcup_{L_i \in \mathcal{L}} L_i = A$.

Consequent with the previous definitions, we now consider decidability in the context of families of languages and uniform classes of Turing machines.

Definition 4.3. Let $\mathcal{L}$ be a language family and $\mathcal{M}$ be an uniform class of Turing machines. We say that $\mathcal{M}$ *uniformly decides* $\mathcal{L}$ if for every $L_i \in \mathcal{L}$ there is an $M_j \in \mathcal{M}$ such that M_j decides L_i.

The following definition clarifies when we can say in this context that a language (uniformly) belongs to a complexity class.

Definition 4.4. Let $\mathcal{C}$ be a complexity class and $\mathcal{L}$ be a language family. A language A *is uniformly in* $\mathcal{C}$ via $\mathcal{L}$ if the following holds:

- $\mathcal{L}$ is compatible with A.
- There is a uniform class of Turing machines $\mathcal{M}$ which uniformly decides $\mathcal{L}$.
- Each Turing machine in $\mathcal{M}$ satisfies the same resource restrictions that define $\mathcal{C}$.

For a uniform reduction of a language to a language family, we simply require the existence of a standard polynomial time many-one reduction to a single member of that family.

Definition 4.5. A language A is *uniformly many-one reducible* to a language family $\mathcal{L}$ (denoted $A \leq^u_m \mathcal{L}$) iff there is a language $L \in \mathcal{L}$ such that A is polynomial time many-one reducible to L, i.e., iff $A \leq_m L$.

We have now the necessary tools to define our uniform notion of completeness (and hardness).

Definition 4.6. Given a complexity class $\mathcal{C}$ and a language family $\mathcal{L}$,

- A language A is *uniformly* $\mathcal{C}$*-hard via* $\mathcal{L}$ iff $\mathcal{L}$ is compatible with A and for any language B in $\mathcal{C}$, we have that $B \leq^u_m \mathcal{L}$.
- A language A is *uniformly* $\mathcal{C}$*-complete via* $\mathcal{L}$ iff it is uniformly $\mathcal{C}$-hard *via* $\mathcal{L}$ and uniformly in $\mathcal{C}$ *via* $\mathcal{L}$.

Classical complete problems in complexity theory lead to some interesting consequences such as Corollary 3.19c in Balcázar et al. (1995) which states that if a PSPACE-complete problem under Karp reductions is in P, then PSPACE = P. The following lemma shows that our somehow "weaker" notion of uniform completeness still allows us to derive similar kinds of results.

Lemma 4.1. *Let A be uniformly PolyL-complete via a problem family* $\mathcal{L}$*. If A is uniformly in P via* $\mathcal{L}$*, then PolyL* $\subseteq$ *P.*

Proof: Let $\mathcal{M}$ and $\mathcal{M}'$ be classes of deterministic Turing machines that uniformly decide $\mathcal{L}$ and, respectively, witness that A is uniformly in PolyL and P. Since we assume that A is uniformly complete for PolyL *via* $\mathcal{L}$, it follows by definition that for each language B in PolyL there is an L in $\mathcal{L}$ such that $B \leq_m L$, and thus also a corresponding Turing machine $M_{B,L}$

that computes this reduction in polynomial time. Furthermore, the fact that A is uniformly in P *via* $\mathcal{L}$ implies (again by definition) that there is a deterministic Turing machine $M_L \in \mathcal{M}'$ that decides L in polynomial time. Therefore, we can define a deterministic Turing machine M_B that decides B in polynomial time. This shows that, under the assumptions in this lemma, PolyL $\subseteq$ P. The machine M_B works by simply assembling together $M_{B,L}$ and M_L, redirecting the output of $M_{B,L}$ to a work tape and making M_L read its input from that work tape.

□

It is interesting to note that PolyL is included in the class of problems that have quasi-polynomial time algorithms. This class is defined as $\mathrm{QP} = \bigcup_{k\geq 0} \mathrm{DTIME}(2^{\log^k n})$ (see Babai, 2016 among others).

5. A UNIFORMLY COMPLETE LANGUAGE

In this section we show that the language $\mathrm{QSAT}^{pl} = \bigcup_{k\leq 0} \mathrm{QSAT}^{pl}_k$ captures the essence of the most difficult problems in PolyL. That is, we prove the following result.

Theorem 5.1. *$QSAT^{pl}$ is uniformly PolyL-complete via the language family $\mathcal{L} = \{QSAT^{pl}_k\}_{k\geq 0}$.*

As in classical complexity theory, we essentially need to prove that QSAT^{pl} is PolyL-hard and that it is indeed in this class. The subtle but important difference lies in the fact this is not possible in the traditional sense. We use instead the concept of uniform completeness (and hardness) introduced in Definition 4.6. Theorem 5.1 is thus a direct consequence of the fact that the following two conditions hold *via* the language family $\mathcal{L} = \{\mathrm{QSAT}^{pl}_k\}_{k\geq 0}$:

a. The language QSAT^{pl} is *uniformly* Poly-L-hard.
b. The language QSAT^{pl} is *uniformly* in PolyL.

Lemma 5.2 and 5.3 below prove, respectively, that both conditions are meet.

Lemma 5.2. *$QSAT^{pl}$ is uniformly PolyL-hard via $\mathcal{L}$.*

Proof: Since by definition $\mathcal{L} = \{\mathrm{QSAT}^{pl}_k\}_{k\geq 0}$ and $\mathrm{QSAT}^{pl} = \bigcup_{k\leq 0} \mathrm{QSAT}^{pl}_k$, it is trivial to see that $\mathcal{L}$ is compatible with QSAT^{pl}.

Now we need to show that for any language B in PolyL, $B \leq^u_m \mathcal{L}$. Since this is the case if there is a language QBF^{pl}_k for some k such that $B \leq_m \mathrm{QBF}^{pl}_k$, we only need to show that we can build a quantified Boolean formula φ from the specification of a Turing machine M with polylogarithmic space bound and input w, such that φ evaluates to 1 iff M accepts w. Notice that the k can be as big as necessary and that there will always be a k big enough so that the encoding of φ belongs to QBF^{pl}_k. Thus the formula φ can be built exactly as in the proof that QSAT is PSPACE-hard (see for instance Theorem 3.29 in Balcázar et al., 1995), since PolyL is included in PSPACE. □

Lemma 5.3. *$QSAT^{pl}$ is uniformly in PolyL via $\mathcal{L}$.*

Proof: We have already seen in the proof of the previous lemma that $\mathcal{L}$ is compatible with QSAT^{pl}. Thus, we need to show that there is a uniform class of Turing machines $\mathcal{M}$ which uniformly decides $\mathcal{L}$ and that each Turing machine in $\mathcal{M}$ works in polylogarithmic space.

We start by showing that, for every $k \geq 0$, the language QSAT^{pl}_k is in $\mathrm{DSPACE}(\log^k n)$. Let $\Sigma_{\mathrm{QBF}} = \{\wedge, \vee, \neg, \forall, \exists, (,), 0, 1, \mathit{true}, \mathit{false}\}$. As discussed in section 3, we can encode arbitrary quantified Boolean formulae as words over this finite alphabet. We build a deterministic Turing machine M_k that takes as input a word $w \in \Sigma^*_{\mathrm{QBF}}$ which encodes a (not necessarily well-formed) quantified Boolean formula φ and decides whether $w \in \mathrm{QSAT}^{pl}_k$ working in space bounded above by $\log^k |w|$.

Let *eval* be the recursive procedure described in **Algorithm 1** which computes the value of a quantified Boolean formula in prenex normal form. If the length of w is n, then it is clear that the depth of the recursion defining *eval* cannot exceed this number, since the number of variables must be less than n. Furthermore, since we actually need to decide whether $w \in \mathrm{QSAT}^{pl}_k$, we can stop the recursion and return false if the quantifier free part of the formula has not been reached at a recursion depth of $|var(\varphi)|$ which by definition of QSAT^{pl}_k is less than $\log^k n$.

To implement *eval* we can use a stack, where in each entry we record the quantifier prefix up to that point, using a four-tuple of the form $(Q_i, \bar{b}, v_1, v_2)$ for each quantifier Q_i in the prefix of the formula. The components of this tuple are as follows: Q_i is either $\forall$ or $\exists$, $\bar{b}$ is the index in binary of the quantified variable x_i, v_1 is the truth value 0 or 1 assigned to this variable (initially 0) and v_2 records the truth value of the sub-formulae ψ_i in $Q_i x_i \psi_i$. The value of v_2 is blank if the sub-formula ψ_i has not been evaluated yet. Once ψ_i has been evaluated for first time with $x_i = 0$, the returned truth value 0 or 1 is stored in v_2, v_1 is updated to the value 1 and the subformula ψ_i is evaluated again with $x_i = 1$. At this point, we update v_2 to the truth value obtained by taking the disjunction or conjunction of its current value with the one returned by ψ_i, depending on whether Q_i is $\exists$ or $\forall$, respectively. This value v_2 is then returned as value for the corresponding call $eval(Q_i x_i \psi_i)$.

Given that the described approach needs space $|var(\varphi)| \cdot (8 + \log |var(\varphi)|)$ for each stack entry, and that we have seen that the maximum recursion depth needed in our case is $|var(\varphi)|$, we get

Algorithm 1 Evaluation of a quantified Boolean formula in prenex normal form.

```
1: procedure eval(φ)
2:     if φ is quantifier-free then
3:         return eval_quantifier_free(φ)
4:     if φ has the form ∀xψ then
5:         return eval(ψ[x/0]) ∧ eval(ψ[x/1])
6:     if φ has the form ∃xψ then
7:         return eval(ψ[x/0]) ∨ eval(ψ[x/1])
```

that working space bounded by $|var(\varphi)|^3$ is enough to implement this evaluation strategy for the quantifier prefix of φ.

Regarding the evaluation of the quantifier free part of φ, note that every time that we reach the last quantifier in the prefix, we have a full valuation for the variables in the quantifier free subformula. Thus we can evaluate this quantifier free subformula in space bounded by $\log n$. Note that the algorithm in Buss (1987) for the evaluation of Boolean formulas with variables and a value assignment works in alternating logarithmic time, which is known to be in L (i.e., in deterministic logarithmic space). See Theorem 2.32 in Immerman (1999) among other sources.

Thus, the size of the stack is what determines the upper bound in the space needed by M_k to decide whether w, i.e., the encoding of φ, is in QSAT_k^{pl}. Since this size is $|var(\varphi)|^3$ and by definition of QSAT_k^{pl} we know that $|var(\varphi)|^3 \leq \log^k n$, we get that M_k can decide whether $w \in \mathrm{QSAT}_k^{pl}$ using space bounded above by $\log^k |w|$.

Clearly, the class $\mathcal{M} = \bigcup_{k\geq 0} M_k$, where each M_k is as described above, uniformly decides the language QSAT^{pl}. Furthermore, since we define M_k constructively, there is a Turing machine $M_{\mathcal{M}}$ which for every input k builds an encoding of the corresponding $M_k \in \mathcal{M}$. This concludes our proof. □

6. CONCLUSION

In this article, we explore an alternative notion of completeness for PolyL. This notion is inspired by the concept of uniformity from circuit complexity theory. This results in a new concept of uniform completeness, which shows that we can still isolate the most difficult problems inside PolyL and draw some of the usual interesting conclusions entailed by the classical notion of complete problem (see in particular Lemma 4.1). The result is relevant for it has been well-known since a long time that PolyL has no complete problems in the usual sense. It is plausible that this new concept of uniform completeness can be applied to other interesting complexity classes for which there are no (known) complete problems. The hope is to further our understanding of practically relevant complexity classes. Examples of such classes are deterministic and non-deterministic polylogarithmic time (see Ferrarotti et al., 2020 and Ferrarotti et al., 2021 among others) as well as the well-known quasi-polynomial time.

DATA AVAILABILITY STATEMENT

The original contributions presented in the study are included in the article/supplementary material, further inquiries can be directed to the corresponding author.

AUTHOR CONTRIBUTIONS

All authors listed have made a substantial, direct, and intellectual contribution to the work and approved it for publication.

FUNDING

The research reported in this article has been partially funded by the Federal Ministry for Climate Action, Environment, Energy, Mobility, Innovation and Technology (BMK), the Federal Ministry for Digital and Economic Affairs (BMDW), and the Province of Upper Austria in the frame of the COMET - Competence Centers for Excellent Technologies Programme managed by the Austrian Research Promotion Agency FFG (Österreichische Forschungsförderungsgesellschaft FFGNr. 865891).

REFERENCES

Babai, L. (2016). "Graph isomorphism in quasipolynomial time," in *Proceedings of the Forty-Eighth Annual ACM Symposium on Theory of Computing* (Cambridge, MA), 684–697.

Balcázar, J. L., Díaz, J., and Gabarró, J. (1990). *Structural Complexity II, Vol. 22 of EATCS Monographs on Theoretical Computer Science*. Berlin; Heidelberg; New York, NY; London; Paris; Tokyo; Hong Kong: Springer.

Balcázar, J. L., Díaz, J., and Gabarró, J. (1995). *Structural Complexity I, 2nd Edn*. Berlin; Heidelberg; New York, NY; London; Paris; Tokyo; Hong Kong: Springer.

Buss, S. R. (1987). "The boolean formula value problem is in ALOGTIME," in *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, ed A. V. Aho (New York, NY: ACM), 123–131.

Ferrarotti, F., Gonzales, S., Schewe, K., and Turull Torres, J. M. (2020). A restricted second-order logic for non-deterministic poly-logarithmic time. *Log. J. IGPL* 28, 389–412. doi: 10.1093/jigpal/jzz078

Ferrarotti, F., González, S., Turull Torres, J. M., Van den Bussche, J., and Virtema, J. (2021). Descriptive complexity of deterministic polylogarithmic time and space. *J. Comput. Syst. Sci.* 119, 145–163. doi: 10.1016/j.jcss.2021.02.003

Hartmanis, J., Lewis, P. M., and Stearns, R. E. (1965). "Hierarchies of memory limited computations," in *6th Annual Symposium on Switching Circuit Theory and Logical Design* (Ann Arbor, MI: IEEE), 179–190.

Immerman, N. (1999). *Descriptive Complexity*. New York, NY: Springer.

Murray, C. D. and Williams, R. R. (2017). On the (non) np-hardness of computing circuit complexity. *Theory Comput.* 13, 1–22. doi: 10.4086/toc.2017.v013a004

Papadimitriou, C. H. (1994). *Computational Complexity*. Reading, MA; Menlo Park, CA; New York, NY; Don Mills, ON; Wokingham; Amsterdam; Bonn; Sydney, NSW; Singapore; Tokyo; Madrid; Milan; Paris: Addison-Wesley.

Ragusa, M. A. (2012). Parabolic herz spaces and their applications. *Appl. Math. Lett.* 25, 1270–1273. doi: 10.1016/j.aml.2011.11.022

Conflict of Interest: FF and SG were employed by Software Competence Center Hagenberg.

The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.