Computing With a Fixed Number of Pointers

Martin Hofmann and Ramyaa Ramyaa

Ludwig Maximilian University
Munich, Germany
{hofmann,ramyaa}@ifi.lmu.de

Abstract

Consider the P-complete problem HORN which asks whether a given set of Horn clauses is (un)satisfiable. To solve it one keeps a dynamic set of atoms that are forced to be true. Using the clauses one then adds atoms to this set until saturation is reached. It is easy to see that this dynamic set will in general more than constant size even if we allow to discard already proved atoms. Given that we need logarithmic space to store a single atom on a Turing machine tape this seems like a strong intuitive argument for the hypothesis that logarithmic space is different from polynomial time. We thus tried to find formal models of computation in which this intuitive argument can be made rigorous. Thus, we study computational models that can be simulated in logarithmic space and encompass logspace algorithms which manipulate a constant size of objects that require logarithmic space individually such as pointers or graph nodes. The hope is then to be able to show that such models are provably unable to solve P-complete problems. We report in this survey article on our partial results towards this goal as well as the state-of-the-art in general.

1998 ACM Subject Classification F.4.3 Formal Languages

Keywords and phrases Logarithmic space, Jumping graph automata (JAGs), st-connectivity, co-st-connectivity, Cayley graphs

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2013.3

Category Invited Talk

1 Introduction

Consider the problem HORN defined as follows. We are given a finite set of letters Σ and a finite set of clauses $\mathcal{C}$ each of the form $U \to V$ where U, V are subsets of Σ with $|U| \leq 2$ and $|V| \leq 1$. A *valuation* $\eta : \Sigma \to \{0, 1\}$ *satisfies* a clause $U \to V$ if $\eta(x) = 0$ for some $x \in U$ or $\eta(y) = 1$ for some $y \in V$. E.g. η satisfies $x, y \to z$ if whenever $\eta(x) = \eta(y) = 1$ then $\eta(z) = 1$, too. It satisfies $\to x$ (here $U = \emptyset$) if $\eta(x) = 1$; it satisfies $x \to$ (here $V = \emptyset$) if $\eta(x) = 0$.

For example, the instance $\mathcal{C} = \{\to x;\ x \to y;\ x,y \to z;\ z \to\}$ (where $\Sigma = \{x, y, z\}$) is unsatisfiable. We call a letter x with $\to x$ in $\mathcal{C}$ an *axiom* and we call a letter y with $y \to$ in $\mathcal{C}$ a *goal*. In this terminology an instance is unsatisfiable if one of the goals may be deduced from the axioms with the understanding that if there is a clause $x, y \to z$ and x, y have both been deduced then z may be deduced, too.

The obvious dynamic programming algorithm (grow a set of letters that can be deduced by repeatedly going over the clauses until you reach a goal) places HORN into the class PTIME (polynomial time) and it is well-known and easy to see that HORN is PTIME-complete w.r.t. LOGSPACE-reductions, e.g. by a straightforward reduction to Cook's solvable paths [1].

On the other hand, considering that storing a single letter requires logarithmic space (in the number of letters $|\Sigma|$), we see that this algorithm does not run in logarithmic space on a

33rd Int'l Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2013).
Editors: Anil Seth and Nisheeth K. Vishnoi; pp. 3–18

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Turing machine unless we can bound the size of the dynamic set in which we keep already deduced letters by a constant. We might try to improve the algorithm by removing entries from the dynamic set according to some strategy. However, even if we allow nondeterministic selection of the letters to be "forgotten", we will not be able to achieve a constant bound as can be seen from the following pebble argument:

▶ Proposition 0.1. For each $d \in \mathbb{N}$ let $\Sigma_d = \{0,1\}^{\leq d}$ and

$$\mathcal{C}_d = \{\rightarrow w \mid \ |w| = d\} \cup \{w0, w1 \rightarrow w \mid \ |w| < d\} \cup \{\epsilon \rightarrow\}$$

The instance $(\Sigma_d, \mathcal{C}_d)$ is unsatisfiable. Let $A_1, \ldots, A_N$ be a sequence of subsets of Σ such that $A_1 = \emptyset$ and $A_{i+1} \subseteq A_i \cup \{x \mid \ y, z \rightarrow y \in \mathcal{C}_n,\ \{y,z\} \subseteq A_n\}$. If $\epsilon \in A_N$ then $\max_i |A_i| \geq d+1$. The easy proof is by induction on d.

Since $|\Sigma_d| = 2^{d+1} - 1$ this furnishes a logarithmic lower bound on the size of the dynamic set in any algorithm for HORN that uses the dynamic programming strategy and is allowed to forget already established facts according to any strategy however clever it might be. Thus, no such algorithm can be implemented in logarithmic space (to be precise, space $\log(n)^2$ is needed at least). We also remark that even a $\sqrt{n}$ lower bound on the size of the dynamic set can be achieved using a more complicated instance based on Cook's "pyramids" [3].

A simpler yet related problem is st-connectivity in directed or undirected graphs. It can be considered as the special case of HORN where one has only one axiom (the source), one goal (the target), and for every other clause $U \rightarrow V$ has $|U| = |V| = 1$. Those then represent the edges of a graph. In this case, the pebbling argument no longer works and indeed, we can solve st-connectivity with a nondeterministic algorithm that keeps only a constant number of letters (graph nodes) in memory. In fact just one suffices. On the other hand, it is not known whether a deterministic algorithm with the same space bounds exists for this problem.

Indeed, Cook and Rackoff have shown that no deterministic algorithm exists for st-connectivity even on undirected graphs (USTCON) which uses graph nodes only abstractly rather than having access to their machine representation and can only hold a fixed number of graph nodes simultaneously in memory. Of course, one must make precise what is meant by "use graph nodes only abstractly". There are several formalisations which attempt to do so. We describe below in more detail: Cook and Rackoff's Jumping Automata on Graphs (JAGs) for which their original result holds; Deterministic Transitive Closure Logic, an extension of first order logic by a transitive closure operator providing unlimited iteration; and, finally, the PURPLE programming language introduced by the first author and U. Schöpp which extends JAGs with a Java like iterator and subsumes DTC-logic. In [14] we could extend Cook and Rackoff's result to PURPLE and thus also show that DTC-logic cannot define USTCON which was an open question at the time.

Interestingly, however, Reingold's result [19] shows that USTCON *can* be solved in logarithmic space on a Turing machine. However, this algorithm, in addition to storing a constant number of graph nodes stores a logarithmic number of booleans.

At the time, a motivation for Cook and Rackoff's work could have been the attempt to separate PTIME from LOGSPACE (deterministic logarithmic space) or indeed LOGSPACE from NLOGSPACE (nondeterministic logarithmic space). Indeed, before Reingold's result one might be tempted to try to show the non-existence of any LOGSPACE-algorithm by devising an ingenuous abstraction that associates with any run of a deterministic algorithm for USTCON a run of an accompanying JAG. Now, Reingold's result shows that without further restrictions on the nature of such hypothetical algorithm such an argument is bound to fail.

On the other hand, Reingold's result shows that looking at graph nodes or propositional letters as abstract objects provably makes a difference which raises the hope that while

an unconditional separation of, say, LOGSPACE and PTIME is not attainable with current methods one might still obtain a "relativized" separation, for instance, in the form of a rigorous proof that no algorithm for HORN exists that only uses a fixed number of abstract pointers (referring to graph nodes, letters, clauses, or similar).

Of course, neither JAGs nor DTC-logic and even PURPLE are able to solve USTCON, and thus are unable to solve HORN. Thus, for a meaningful result of this kind, one would need an extension of any of these formalisms which does encompass USTCON. In this paper we summarise our efforts at finding such an extension and try to explain why this turned out to be unexpectedly difficult.

The rest of this paper is structured as follows.

The next section describes the basic framework we consider - systems which build upon minimal abstract pointers.

Section 3 describes Cook and Rackoff's jumping automata on graphs, the first, and most basic model for computation with a fixed number of pointers. We also report therein on our current study of the computational strength of nondeterministic jumping automata.

Section 4 describes the programming formalism PURPLE [15] which extends JAGs with an iteration construct that provides access to all nodes of the input structure not only those that are accessible from originally known ones. We present our results on the expressivity of PURPLE and its extensions.

Section 5 is about (deterministic) transitive closure logic known from finite model theory. We show that its strength lies strictly between JAGs and PURPLE.

Section 6 concludes and gives some directions for future work.

2 Minimal Abstract Pointers

Our aim is to define a system contained in LOGSPACE that is provably weaker than PTIME, but powerful enough to make the separation non trivial. Following the intuition outlined above, the systems we consider aim to capture the subclass of "abstract pointer algorithms" within LOGSPACE.

The minimal requirements of such a system is the ability to refer to a node of a graph (or any structured input) using pointer (or pebble) variables, comparing two such variables for equality, assigning the value of one to another, and traversing along an edge. Thus, the concrete representation of the pointers is hidden. The main difference between these systems and Turing machines is that Turing machines have access to a binary encoding of the input graph which embodies a total ordering on the nodes.

In fact, it is easy to see that once a total ordering on the input nodes is available even the most basic systems we consider capture all of LOGSPACE. Such a total ordering can be provided in a number of ways, for instance by assuming that one of the directions $\{1, \ldots, d\}$ threads all the nodes ($\lambda x.x.i$ is a permutation for some i). So, we want to focus on graphs for which such total ordering is neither directly available nor definable.

On the other hand, we typically assume the presence of a local order, in particular an ordering of the edges adjacent to or emanating from a node. A *1-locally ordered graph*, (1LO graph) is a one such that for each node, the edges emanating from it are ordered. A *2-locally ordered graph* (2LO graph) is one such that for each node, the edges coming out of it or leading to it are ordered. For undirected graphs these two coincide.

The encoding of graphs using abstract pointers and the presentation of graphs in the context of JAGs both induce at least a 1-local ordering. It is, however, possible to represent graphs without local ordering as pointer structures, e.g. by using special edge objects having

pointers to their source and target. In the context of first-order logic and extensions thereof, the unordered case may seem more natural but leads to artificial weaknesses.

An interesting intermediary between absence and presence of a total ordering is the ability to count or more generally perform arithmetic till the size of the input, called *counting*. A total order can be utilized to simulate counting. However, this is not the case with local orderings. In the extreme case of discrete graphs, local ordering provides no information at all, and it is not possible to count over these graphs.

The systems we study are: jumping automata on graphs (JAGs), "pure pointer language" (PURPLE language), and Deterministic transitive closure Logic (DTC). It has been shown that without counting, these systems cannot solve connectivity. We describe these systems and several extensions.

3 Jumping automata on graphs

Cook and Rackoff ([2]) introduced Jumping Automata on Graphs (JAGs) in order to study space lower bounds for reachability problems. A JAG is a finite automaton which can move a fixed number of pebbles along labelled edges of input graphs of fixed degree. Thus, JAGs are a nonuniform machine model.

A labelled degree d graph for $d > 1$ comprises a set V of vertices and a function $\rho : V \times \{1, \ldots, d\} \to V$. If $\rho(v, i) = v'$ then we say that (v, v') is an edge labelled i from v to v'. All graphs considered here are labelled degree d graphs for some d. The important difference to the more standard graphs of the form $G = (V, E)$ where $E \subseteq V \times V$ is that the out degree of each vertex is exactly d and, more importantly, that the edges emanating from any one node are linearly ordered. One extends ρ naturally to sequences of labels (from $\{1, \ldots, d\}^*$) and writes $v' = v.w$ if $\rho(v, w) = v'$ for $v, v' \in V$ and $w \in \{1, \ldots, d\}^*$. The induced sequence of intermediate vertices (including v, v') is called the path labelled w from v to v'. Such a graph is undirected if for each edge there is one in the opposite direction $\rho(v, i) = v' \Rightarrow \exists j.\rho(v', j) = i$. It is technically useful to slightly generalise this and also regard such graphs as undirected if each edge can be reversed by a path of a fixed maximum length.

▶ **Definition 1.** A d-Jumping Automaton for Graphs (d-JAG), J, consists of

- a finite set Q of states with distinguished start state q_0 and accept state q_a
- a finite set P of objects called pebbles (numbered 1 through p)
- a transition function δ which assigns to each state q and each equivalence relation π on P (representing incidence of pebbles) a set of pairs $(q', \vec{c})$ where $q' \in Q$ is the successor state and where $\vec{c} = (c_1, \ldots, c_p)$ is a sequence of moves, one for each pebble. Such a move can either be of the form move(i) where $i \in \{1, \ldots, d\}$ (move the pebble along edge i) or jump(j) where $j \in \{1, \ldots, p\}$ (jump the pebble to the (old) position of pebble j).
- The automaton is deterministic if $\delta(q, \pi)$ is a singleton set for each q, π.

The input to a JAG is a labelled degree d graph. An *instantaneous description* (id) of a JAG J on an input graph G is specified by a state q and a function, *node*, from the P to the nodes of G where for any pebble p, *node*(p) gives the node on which the pebble p is placed.

Given an id $(q, node)$ a legal move of J is an element $(q', \vec{c}) \in \delta(q, \pi)$ where π is the equivalence relation given by $p\ \pi\ p' \iff node(p) = node(p')$. The action of a JAG, or the *next move* is given by its transition function and consists of the control passing to a new state after moving each pebble i according to c_i: (a) if $c_i =$ move(j) move i along edge j; (b) if $c_i =$ jump(j) move (jump) it to *node*(j). Any sequence (finite or infinite) of id's of a JAG J on an input G which form consecutive legal moves of J is called a *computation* of J

on G. We assume that input graphs G have distinguished nodes *startnode* and *targetnode*, and that JAGs have dedicated pebbles s and t. The initial id of J on input G has state q_0, $node(t) = targetnode$ and $\forall q \neq t.\ node(q) = startnode$. J accepts G if the computation of J on G starting with the initial id ends in an id with state q_a.

One criticism of JAGs is that they are artificially weak on directed graphs. Since edges can only be traversed in the forward direction, there is no way for a JAG to reach a node without incoming edges, for example. One solution to this is to work with graphs having a local ordering both on the outgoing and on the incoming edges of each node, so that edges can be traversed in both directions [7].

Cook and Rackoff's result [2] shows that even with this modifications, JAGs can only compute local properties:

▶ **Theorem 2** ([2]). *(u)st-connectivity cannot be solved by* JAG*s.*

It is instructive to deduce this result from a generalization due to Schöpp [20] who gave a formal proof in Coq. For any group G define its exponent $\exp(G)$ as the maximum element order, i.e., the least m so that $g^m = e$ holds for all $g \in G$. Note that $\exp((\mathbb{Z}/m\mathbb{Z})^d) = m$.

▶ **Theorem 3.** *Let G be a group. A* JAG *with p pebbles and q states can visit at most $(q \cdot \exp(G))^{d^p}$ nodes in the course of any computation on $CG(G)$.*

This implies that JAGs trivially cannot solve HORN since USTCON is obviously a special case of HORN. It is thus natural to investigate strengthenings of JAGs.

3.1 Counters

The RAMJAG [18] consists of a finite state control together with p pebbles and a fixed number of $O(\log(n))$-bit registers which in total require $O(log(q))$ bits of storage. Its storage is defined as $(plog(n) + log(q))$ bits. on which it can perform the usual RAM operations on the registers and also three special instructions: walk, jump and compare. The instructions $walk(P, j)$ and $jump(P, P')$ are the same as that in a JAG. The instruction $compare(P, P', R)$ checks whether pebbles P and P' are on the same node and stores the result (T or F) in a register R. where n is the size of the input graph. Obviously, using more registers this bound extends to any polynomial in n.

▶ **Theorem 4** ([18]). *Reingold's algorithm for* USTCON *can be implemented with* RAMJAG*s.*

We remark that Beame et al. citeDBLP:journals/siamcomp/BeameBRRT99 showed that JAGs with arithmetic registers ($O(\log n)$ space bounded JAGs in their terminology) are equivalent to LOGSPACE Turing machines without using and in fact prior to Reingold's theorem.

▶ **Corollary 5.** RAMJAG*s can define a total order on connected graphs*

Proof. Choose an arbitrary node as the start node and enumerate all logarithmic length paths from it. The order in which the nodes are visited gives a total ordering on the graph. ◀

▶ **Corollary 6.** JAG*s cannot count*

Proof. JAGs cannot solve USTCON, but RAMJAGs can. This shows that counting cannot be simulated in JAGs. ◀

3.2 Nondeterminism

Regarding nondeterministic JAGs (ND-JAGs) the situation is less clear. For our purpose, ND-JAGs are relevant, for they are stronger than deterministic JAGs and PURPLE since they can solve STCON, so assuming that they are not equivalent to NLOGSPACE it would then constitute and interesting and perhaps accessible open question whether nondeterministic JAGs can solve HORN.

Before, however, attempting that question one should first try to see whether ND-JAGs can solve co-STCON or even just co-USTCON. i.e., whether there exists a nondeterministic JAG with the property that if *startnode* and *targetnode* are not connected then there exists an accepting run whereas in the case where they are connected, all runs reject or abort. It has been left as an open problem in [5] whether or not ND-JAGs are able to decide co-USTCON or indeed whether their computational power equals all of NLOGSPACE.

In a recent as yet unpublished paper [12] the authors have tried to make some progress towards the special case of this question where the graphs under consideration are *Cayley graphs*. Recall that the Cayley graph of a group G with generators $\vec{m}$, written as $CG(G, \vec{m})$ is the labelled degree $|\vec{m}|$ graph whose nodes are the elements of G and where the edge labelled i from node v leads to $m_i v$, formally $\rho(v, i) = m_i v$. Note that since every generator has an inverse the Cayley graphs are undirected in the above relaxed sense.

Cayley graphs are interesting in this contexts because they furnish the hard examples in [2] and [14]. In the former case the underlying group is $(\mathbb{Z}/m\mathbb{Z})^m$ whose Cayley graph resembles an m-dimensional torus of circumference d. E.g. for $m = 480$ and $d = 2$ one obtains a 480 x 480 "screen" with opposite borders identified as is common in some video games. It is hard for a JAG to find its way through such a graph because the close neighbourhoods of all nodes are the same and because repeated moves quickly lead to a repetition. E.g. the order of each cyclic subgroup of $(\mathbb{Z}/m\mathbb{Z})^d$ is $\leq m$ whereas the order of the group itself is m^d. Of course, a JAG does not necessarily stupidly repeat a fixed move and it required Cook and Rackoff an ingenious pumping argument to turn this intuition into a rigorous proof. The result in [14] generalises this using an iterated wreath product of a cyclic group $\mathbb{Z}/m\mathbb{Z}$.

▶ **Definition 7.** A nondeterministic Jumping Automaton for Graphs (ND-JAG) J is a JAG whose transition function is nondeterministic. It accepts an input if there is *some* finite computation starting at the initial configuration that reaches q_a, and rejects an input if *no* such computation does. A $d-$ND-JAG operates on graphs of degree d. Again, we assume that appropriate degree reduction is applied before inputting a graph to an ND-JAG.

It is easy to see that the argument used in [2] for deterministic JAGs which is similar to a pumping argument cannot be adapted easily to ND-JAGs, which can solve reachability (guess a path from *startnode* to *targetnode*). However, it is unclear whether ND-JAGs can solve co-st-connectivity. Since JAGs cannot count, it is reasonable to believe that ND-JAGs cannot implement Immerman-Szelepcsenyi's algorithm, and more generally, cannot solve co-st-connectivity.

▶ **Theorem 8** ([12]). ND-JAG*s are equivalent to* NLOGSPACE *and thus can in particular solve co-*STCON *if the input consists of disjoint copies of one of Cayley graphs where the underlying group is one of the following*

- *an abelian group*
- *a finite simple group*
- *groups obtained from these by direct, semidirect, or wreath product*
- *iterations of the above product constructions*

To us this power of ND-JAGs came as quite a surprise; initially, we believed that even on the (abelian) group $(\mathbb{Z}/m\mathbb{Z})^d$ that were used by Rackoff the ND-JAGs would be strictly weaker than NLOGSPACE.

To give a taste of these results we sketch here a special case:

▶ **Theorem 9.** *There exists an* ND-JAG *that can visit all nodes of* $CG(\vec{m}, (\mathbb{Z}/m\mathbb{Z})^d)$ *in a fixed order where* $\vec{m}$ *comprises the unit vectors* $e_1, \ldots e_d$ *of dimension* d*.*

This shows in particular that co-STCON can be solved if the input consists of several disjoint copies of this Cayley graph.

Proof sketch. Indeed, each element of the group can be uniquely written in the form $\lambda_1 e_1 + \ldots \lambda_d e_d$ where $\lambda_i \in \{0, \ldots, m-1\}$. Moreover, we can order the elements of the group lexicographically using this representation. Now, we can design an ND-JAG that places a "cursor pebble" on all nodes of the Cayley graph in this lexicographic order. Suppose that the cursor pebble is on $\lambda_1 e_1 + \ldots \lambda_d e_d$. Using another pebble we nondeterministically trace a path from *startnode* to the cursor pebble making sure that it has the form "some e_1, then some e_2, etc". By uniqueness of representation this path will repeat the coefficients λ_i or fail to reach the cursor pebble in which case we abort. As we trace this path we can on-the-fly move a third pebble to the lexicographically next position by incrementing the first coefficient that is different from $m-1$ and resetting all the previous ones. ◀

Given these results our current working hypothesis is that ND-JAGs can solve co-STCON on all graphs (in fact, we even believe now that they capture NLOGSPACE on all graphs); the natural next step will be to demonstrate this for arbitrary Cayley graphs. We also note that our results considerably narrow the search for counterexamples to the conjecture and in particular rule out all the known ones.

We also note that any counterexample to our working hypothesis would in particular have to be such that deterministic JAGs cannot solve STCON on them (otherwise we would trivially get co-STCON) so that we would need a new proof of Cook-Rackoff's result with substantially different example graphs for the known ones are thwarted by our current results.

4 Purple language

Rather than as an automaton, we may understand a JAG as a while-program whose variables are partitioned into two types: boolean variables and graph pointer variables. Boolean variables are used to represent the finite state of the JAG and the usual boolean operations are available for boolean expressions. Pointer variables are used to reference graph nodes in the same way that pebbles are used in the automata-theoretic formulation of JAGs. For pointer variables one only has an equality test and, for each constant number i, a successor operation to move a pointer variable along the i-th edge from the graph node it points to. Over unconnected graphs, JAGs may not be able to visit all nodes. For example, the property whether a graph contains a node with a self-loop is not decidable with JAGs for the simple reason that such a witnessing node might be in an unreachable part of the graph.

To address this, the formalism PURPLE (for "pure pointer language"), was introduced by the first author and U. Schöpp [15]. Essentially, it consists of augmenting this programming language-theoretic version of JAGs with a special loop construct (`forall` x `do` P) whose meaning is to set the pointer variable x successively to all graph nodes in some arbitrary order and to evaluate the loop body P after each such setting. The important point is that the order is arbitrary and will in general be different each time a `forall`-loop is evaluated.

A program computes a function or predicate only if it gives the same (and correct) result for all such orderings. The `forall`-loop in PURPLE can be used to evaluate first-order quantifiers and thus to encode DTC- logic (see below) on locally ordered graphs. Moreover, PURPLE is strictly more expressive than that logic. For instance, determining whether the input graph has an even number of nodes is not possible in locally-ordered DTC logic [10], but the following PURPLE-program does this: ($b := \mathit{true}$; `forall` x `do` $b := \mathit{not}(b)$.

PURPLE program are parametrised by a finite set L of labels and a finite set S of predicate symbols. Each predicate symbol p is assumed to have a finite arity $ar(p) \in \mathbb{N}$.

The input of a programs is a pointer structure, which interprets the labels and predicates: A *pointer structure* on L and S ((L,S)-model, for short) specifies a finite set U as a universe, a function $[\![l]\!] \colon U \to U$ for each label $l \in L$ and a set $[\![p]\!] \subseteq U^{ar(p)}$ for each predicate symbol p.

The special case of d-labelled graphs arises when $L = \{1, \dots, d\}$ and $S = \emptyset$.

A program with labels L and predicate symbols S can access its input structure through the following terms for pointers to elements of the universe and for booleans.

$$\begin{aligned} t^U &::= x^U \mid t^U.l \text{ for any label } l \in L \\ t^{\texttt{bool}} &::= x^{\texttt{bool}} \mid \neg t^{\texttt{bool}} \mid t_1^{\texttt{bool}} \wedge t_2^{\texttt{bool}} \mid p(x_1^U, \dots, x_{ar(p)}^U) \text{ for any predicate } p \in S \end{aligned}$$

We call x^U pointer variables. The intention is that $t^U.l$ is interpreted by $[\![l]\!](t^U)$.

The programs themselves are given by the grammar.

$$\begin{aligned} \mathit{Prg} ::= \quad & \texttt{skip} \mid \mathit{Prg}_1; \mathit{Prg}_2 \mid x^U := t^U \mid x^{\texttt{bool}} := t^{\texttt{bool}} \\ & \mid \texttt{if } t^{\texttt{bool}} \texttt{ then } \mathit{Prg}_1 \texttt{ else } \mathit{Prg}_2 \mid \texttt{forall } x^U \texttt{ do } \mathit{Prg} \end{aligned}$$

We write `if` $t^{\texttt{bool}}$ `then` Prg for `if` $t^{\texttt{bool}}$ `then` Prg `else skip`.

A *configuration* $\langle \rho, q \rangle$ consists of a *pebbling* ρ and a *state* q. The pebbling ρ maps pointer variables (which we also call pebbles) to elements of the universe U. The state q is a function mapping boolean variables to booleans. Given a configuration I, we can define an interpretation of the terms $[\![t^{\texttt{bool}}]\!]_I \in \{\texttt{true}, \texttt{false}\}$ and $[\![t^U]\!]_I \in U$ in the usual way.

A big-step reduction relation $\mathit{Prg} \vdash_M I \longrightarrow O$ between configurations I and O on some (L,S)-model M and a program Prg is defined inductively by the following clauses:

- $\texttt{skip} \vdash_M I \longrightarrow I$.
- $\mathit{Prg}_1; \mathit{Prg}_2 \vdash_M I \longrightarrow O$ if $\mathit{Prg}_1 \vdash_M I \longrightarrow R$ and $\mathit{Prg}_2 \vdash_M R \longrightarrow O$ for some R.
- $x^U := t^U \vdash_M \langle \rho, q \rangle \longrightarrow \langle \rho[x \mapsto [\![t]\!]_{\langle \rho, q \rangle}], q \rangle$.
- $x^{\texttt{bool}} := t^{\texttt{bool}} \vdash_M \langle \rho, q \rangle \longrightarrow \langle \rho, q[x \mapsto [\![t]\!]_{\langle \rho, q \rangle}] \rangle$.
- `if` t `then` Prg_1 `else` $\mathit{Prg}_2 \vdash_M I \longrightarrow O$ if $[\![t]\!]_I = \texttt{true}$ and $\mathit{Prg}_1 \vdash_M I \longrightarrow O$.
- `if` t `then` Prg_1 `else` $\mathit{Prg}_2 \vdash_M I \longrightarrow O$ if $[\![t]\!]_I = \texttt{false}$ and $\mathit{Prg}_2 \vdash_M I \longrightarrow O$.
- `forall` x^U `do` $\mathit{Prg}_1 \vdash_M I \longrightarrow O$ if there exists an enumeration $u_1, u_2, \dots, u_n$ of $[\![U]\!]$ and configurations $I = \langle \rho_1, q_1 \rangle, \langle \rho_2, q_2 \rangle, \dots, \langle \rho_{n+1}, q_{n+1} \rangle = O$, such that $\mathit{Prg}_1 \vdash_M \langle \rho_k[x \mapsto u_k], q_k \rangle \longrightarrow \langle \rho_{k+1}, q_{k+1} \rangle$ holds for all $k \in \{1, \dots, n\}$.

When the model M is clear from the context we may omit the subscript.

In order for a PURPLE program to accept (resp. reject) an input, it must do so no matter what enumerations are chosen for its `forall`-loops. This is defined formally in the next definition, in which we use a boolean variable *result* to indicate acceptance.

▶ **Definition 10.** A program Prg *accepts* (resp. rejects) an (L,S)-model M if $\mathit{Prg} \vdash_M \langle \rho, q \rangle \longrightarrow \langle \rho', q' \rangle$ implies $q'(\mathit{result}) = \texttt{true}$ (resp. $q'(\mathit{result}) = \texttt{false}$) for all ρ, ρ', q and q'.

A program Prg *recognises* a set X of (L,S)-models if it accepts any model in X and rejects all others. Note that a program may neither accept nor reject its input, namely if for some

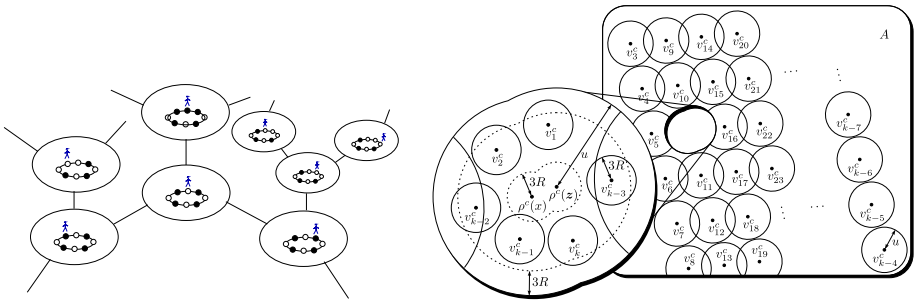

Figure 1 The lamplighter graph $CG(\Lambda(\mathbb{Z}/8\mathbb{Z}))$ and the traversal sequence from [14]

runs it returns `true` and for others it returns `false`. To put it simply, a PURPLE program for some problem X should give the correct answer, be it `true` or `false` for any given input and independent of the run, i.e. the traversal sequences chosen.

▶ **Theorem 11.** *It is undecidable whether a given* PURPLE *program is such that for every input, it either rejects or accepts.*

We also note that predicate symbols, which were not part of the original definition of PURPLE [15], are there just for notational convenience and do not add expressive power. Unary predicates can be modelled with an extra pointer that points to designated nodes for "true" and "false". A binary relation can be modelled by introducing an extra node for each pair of related nodes with pointers *fst* and *snd* pointing to the latter two nodes. One uses a unary predicate to differentiate between the actual nodes and these helper nodes.

4.1 Power of purple

While PURPLE subsumes JAGs and also deterministic transitive closure logic (see below) it is strictly weaker than LOGSPACE. Notice that PURPLE programs can be evaluated on a LOGSPACE-bounded Turing machine.

▶ **Theorem 12** ([15]). *Checking whether the input is a discrete graph with n nodes, where n is a power of two, is possible in* LOGSPACE *but cannot be programmed in* PURPLE.

▶ **Theorem 13** ([14]). USTCON *cannot be decided in* PURPLE.

Proof idea. For any group G one defines the lamplighter group $\Lambda(G)$ by $\Lambda(G) = \{(f, g) \mid f \in 2^G, g \in G\}$ and $(f, g)(f', g') = (\lambda h.f(h) + f'(gh), gg')$ where $+$ refers to addition modulo 2. Given a set of generators $(m_1, \ldots, m_k)$ for G one can generate $\Lambda(G)$ by $(0, m_i)$ for $i = 1 \ldots k$ and the *toggle move* (t, e) where $t(e) = 1$ and $t(g) = 0$ for $g \neq e$. The nodes of the Cayley graph $CG(\Lambda(G))$ can be thought of as states of a system involving one streetlight at each node of $CG(G)$ and a lamplighter situation at one such node. Fig. 1 shows 8 of the 2048 nodes of $CG(\Lambda(\mathbb{Z}/8\mathbb{Z}))$. If G has order n, number of generators m and exponent e and n', m', e' denote those measures for $\Lambda(G)$ then we have $n' = n2^n$, $m' = m + 1$, $e' = 2e$. Thus, repeated application of the lamplighter construction furnishes groups with a very high order yet moderate number of generators and exponent. Nothing beyond these numerical properties is required from the lamplighter construction for our purpose.

Now, for any PURPLE-program P we can find t, m depending on size and nesting depth of loops in P so that when run on $CG(\Lambda^t(\mathbb{Z}/m\mathbb{Z}))$ the effect of P can be simulated (for appropriately chosen traversal sequences of the `forall`-loops) by a very large JAG. Theorem 3 applied to this JAG then shows that some nodes remain unvisited.

To obtain the desired simulation by a JAG assume that all `forall`-loops except for the outermost one have already been eliminated. One thus essentially has a JAG with one special iteration pebble that is supposed to be placed on every node in some arbitrary order. In between such placements the JAG is to be run until it reaches a dedicated state. Now, in an intuitive sense that can be made precise, if the traversal sequence is chosen in such a way that temporally close nodes, in particular successive ones, are sufficiently far apart spatially, then the only nodes of the sequence that the JAG will be able to remember will be those from the beginning and the end of the traversal sequence. Thus, if the traversal sequence is chosen in this fashion then at the end of the traversal all pebbles will be close to the original nodes and it is possible to hardwire the total effect of the `forall`-loop into a very large JAG. The second half of Figure 1 illustrates this traversal sequence $v_1^c, v_2^c, \ldots$. Herein R and u are appropriately chosen large numbers. The radius R is related to the number of nods the JAG representing the body of the loop is able to visit according to Theorem 3, quantity u on the other hand stems from the combinatorial possibility of designing an appropriate sequence. Finally, $\rho^c(\_)$ indicates the initial pebble positions. ◀

This result provides the strongest possible evidence so far that USTCON cannot be solved with a constant number of abstract pointers and that the use of arithmetic in Reingold's algorithm is intrinsic to the problem solved. The result also answers an open question about transitive closure logic, see Section 5 below.

Similar to the case of JAGs, this implies that PURPLE trivially cannot solve HORN since USTCON is obviously a special case of HORN. It is thus natural to investigate strengthenings of PURPLE by various computational devices that stay within LOGSPACE or NLOGSPACE. We describe our efforts in this direction in the following subsections.

4.2 Counters

So, one obvious addition to PURPLE is counting. Though PURPLE subsumes JAGs, we explored whether PURPLE with counting is strictly continued within LOGSPACE. Here we extend PURPLE by counting variables ("counters"), each of which can hold a number from 0 to the size of the input structure's universe, and we extend the terms with arithmetic operations:

$$t^{\texttt{bool}} ::= \cdots \mid \mathit{iszero}(t^{\texttt{count}}) \qquad\qquad t^{\texttt{count}} ::= x^{\texttt{count}} \mid \mathit{max} \mid \mathit{pred}(t^{\texttt{count}})$$

We extend the operational semantics such that the state q now not only maps boolean variables to booleans, but also counting variables to numbers.

PURPLE with counters (PURPLE$_c$) can do arithmetic: the complement of a counter can be computed using *max* and repeated decrement; the increment can be implemented using double complementation and decrement; The rest of the operations follow by repeated applications of these. PURPLE$_c$ can count the number of tuples of nodes satisfying any PURPLE-definable property, and so can simulate counting quantifiers used in DTC or TC logics.

▶ **Lemma 14.** PURPLE$_c$ *captures* LOGSPACE *on graphs with a two-way local ordering, represented as pointer structures as described above.*

Outline. PURPLE captures all of LOGSPACE on ordered graphs. So, it suffices to show that a total ordering can be defined on any input graph.

To this end we note that given any graph node n, PURPLE$_c$ can define a total ordering of the weakly connected component containing n. We use Reingold's algorithm for undirected s-t-connectivity, which checks for connectivity by enumerating all nodes in the weakly

connected component of s and checking if t appears therein. This algorithm be implemented by RAMJAGs [18], and so, by an easy translation, also in PURPLE$_c$. In this way, PURPLE$_c$ can order the nodes of the weakly connected component according to their order of their first appearance in the enumeration.

In their proof that TC-logic with counting captures NLOGSPACE [8], Etessami and Immerman have shown how a total ordering can be defined using counting from such orderings of the weakly connected components. This argument can be adapted to PURPLE$_c$ to complete the proof. ◀

4.2.1 Iterators

Another possibility for strengthening PURPLE that we investigated consists of replacing `forall`-loops by *iterators* that are available e.g. in the Java library for the representation of sets as trees or hash maps. Thus, in addition to Boolean and pointer variables, this extension of PURPLE then has *iterator* variables which store a subset of the nodes of the input graph. The operations allowed on an iterator variable are:

- *Initialize*: The variable is assigned the set of all nodes of the input graph
- *Next*: If the variable refers to a non-empty set, an arbitrary node is removed from it (so the variable now refers to a set without this node), and the node is returned.
- *isNull*: Returns `true` if the variable is empty and `false` otherwise

With iterators in place, the `forall`-loop can be replaced by a plain while loop. Further, we can determine whether the number of nodes with a self loop is equal to the number of nodes without one. Surprisingly, iterators permit the definition of counting and thus render PURPLE with iterators (on locally ordered input) equivalent to full LOGSPACE.

▶ **Theorem 15.** PURPLE *with iterators can count.*

Proof. A counter variable is represented by an iterator variable which is assigned a subset of nodes with the required cardinality. It is direct that the operations of checking for zero and decrement can be performed. To increment a variable x, we use another iterator variable y as follows: Initialize y and keep decrementing both x and y till x is zero; increment y once more and initialize x; finally, decrement both variables till y is zero. Similarly, variable can be copied (assigned to another variable). ◀

4.3 Nondeterminism with counting but without local order

By the asymmetric acceptance and rejection conditions that are used in any nondeterministic definition, it is not clear whether nondeterministic PURPLE is closed under complementation. Since the answer is not known even in the case of JAGs, we add counting as well, allowing us to implement Immerman-Szelepcsenyi's algorithm for complementation in NLOGSPACE to get PURPLE $_{c,nd}$. However, adding counting boosts the power of PURPLE to full NLOGSPACE in the presence of local ordering. So, we consider input graphs without local ordering in this case. These can be presented via primitive predicates or special edge objects.

Counters are added as described above. Adding nondeterminism is not completely straightforward, as we must separate nondeterministic choices from the choices made in the evaluation of `forall`-loops. We would like to allow programs to make nondeterministic choices, while still maintaining that their acceptance behaviour is independent of the enumerations chosen in the `forall`-loops.

PURPLE with nondeterminism (PURPLE$_{nd}$) has a command for nondeterministic choice:

$Prg ::= \ldots \mid$ `choose` Prg_1 `or` Prg_2

To define the semantics of PURPLE with nondeterminism, we amend the notion of configuration so that it now consists of a triple $\langle \rho, q, \sigma \rangle$, where ρ and q are a pebbling and a state as before and σ is an infinite list enumerations of the universe U. This new component σ specifies the runs of all future `forall` loops. Therefore, in the definition of $(\texttt{forall } x^U \texttt{ do } Prg) \vdash \langle \rho, q, \sigma \rangle \longrightarrow \langle \rho', q', \sigma' \rangle$ we do not use an arbitrary enumeration of U, but we take the first one $u_1, \ldots, u_n$ from σ. That is, we require there to be configurations $\langle \rho, q, \mathit{tail}(\sigma) \rangle = \langle \rho_1, q_1, \sigma_1 \rangle, \ldots, \langle \rho_{n+1}, q_{n+1}, \sigma_{n+1} \rangle = \langle \rho', q', \sigma' \rangle$ with $Prg \vdash \langle \rho_k[x \mapsto u_k], q_k, \sigma_k \rangle \longrightarrow \langle \rho_{k+1}, q_{k+1}, \sigma_{k+1} \rangle$ for all $k \in \{1, \ldots, n\}$. For the semantics of the new term, we stipulate $\texttt{choose } Prg_1 \texttt{ or } Prg_2 \vdash I \longrightarrow O$ if $Prg_1 \vdash I \longrightarrow O$ or $Prg_2 \vdash I \longrightarrow O$. In all other cases, σ is merely passed on. E.g. $x := t \vdash \langle \rho, q, \sigma \rangle \longrightarrow \langle \rho[x \mapsto [\![t]\!]_I], q, \sigma \rangle$.

With these provisos, we can make the role of the two kinds of choices precise and define when a nondeterministic program accepts an input:

▶ **Definition 16.** A nondeterministic program *Prg accepts* an (L, S)-model M if for all I there exists O with $Prg \vdash_M I \longrightarrow O$ and $O(\mathit{result}) = \texttt{true}$. It rejects M if for all I and for all O with $Prg \vdash_M I \longrightarrow O$ one has $O(\mathit{result}) = \texttt{false}$.

Thus, in the positive case, M must find, for all traversals of the `forall`-loops, appropriate nondeterministic choices leading to result `true`. In the negative case, however, the program must yield result `false` no matter how the nondeterministic choices are made and how the `forall`-loops are being traversed.

Note that for programs without `choose`, this definition agrees with the one for PURPLE above. For programs without `forall`-loop it agrees with the definition of nondeterminism.

In [15] we have shown that PURPLE can evaluate formulae in DTC-logic. One may expect that with nondeterminism, this result generalises to TC-logic. We obtain the following proposition. Recall that any relational structure M can be understood as a pointer structure.

▶ **Lemma 17.** *For each closed* TC*-formula φ on a relational signature Σ, there exists a program P_φ in* PURPLE $_{c,nd}$ *such that, for any Σ-structure of Σ, $M \models \varphi$ holds iff P_φ recognises M.*

4.3.1 Tree isomorphism

The problem of determining whether rooted, directed trees (unlabeled, without edge labeling) are isomorphic is in LOGSPACE.

▶ **Theorem 18** ([17]). *Tree isomorphism is in* LOGSPACE.

The proof uses an algorithm to canonize such trees. The algorithm uses counting, and depth first search for which it relies on the edge ordering implicit in the encoding of the input tree. Etessami and Immerman [8] were able show that transitive closure logic with counting is unable to define tree isomorphism and thus confirmed that such ordering is intrinsically necessary. The same counterexample albeit with a rather different proof technique shows that PURPLE with counting and nondeterminism cannot decide tree isomorphism either.

▶ **Theorem 19** ([13]). PURPLE $_{c,nd}$ *cannot decide tree-isomorphism.*

Proof idea. The family of trees (AB-trees) used by Etessami and Immerman comprises two non-isomorphic tree structures which are defined by mutual induction to contain a large number of isomorphic subtrees. For any two non-isomorphic trees of the same height, say A_h and B_h, every immediate subtree is one of C_h, D_h or E_h. The only difference between A_h and B_h is how many subtrees of each type are present, i.e., when the immediate subtrees

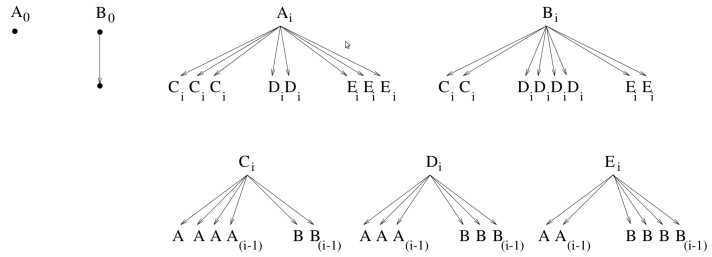

Figure 2 AB-trees

of A_h and B_h are grouped according to isomorphism, the cardinality of the groups differ (though the number of groups is the same). Unlike Etessami and Immerman who use a variant of Ehrenfeucht Fra issé games we rely on a simulation argument between two runs of a given PURPLE program on both A_h and B_h.

Intuitively, to differentiate between t_1 and t_2, PURPLE has to traverse each immediate subtree to group them according to isomorphism and determine the cardinality of each such group. So, reasoning recursively, this would need traversing the trees in a depth first manner. Intuitively, the `forall` -loop provides no additional strength beyond quantifiers here, since it is not required to enumerate the nodes in this order. The main work in the proof goes into showing this rigorously. With no edge ordering, remembering whether a subtree has been traversed or is yet to be traversed involves placing a pebble on the subtree. Given that it has a limited number of pebbles, this cannot be done for arbitrary depths. ◀

4.4 Recursion

PURPLE can be extended with procedures in the expected manner. We restrict the extension to Boolean functions which take a tuple of pointers as input, and allow mutual recursion but disallow global variables or side effects. The semantics of a (mutually) recursive function is defined in terms of least fixed points in the usual way. We do formally allow non-terminating functions but since the set of global states remains bounded such non-termination can be detected and thus we can assume w.l.o.g. that all procedures are total.

This extension does not subsume LOGSPACE since it is unable to count over discrete graphs (cf. Theorem 12).

▶ **Theorem 20.** *No* PURPLE *program with recursion can checking whether the input is a discrete graph with n nodes, where n is a power of two.*

Proof sketch. Non-pebbled nodes of discrete graph are indistinguishable, so the result of a function should be identical on them. This allows one to deduce a bound on recursion depth that is independent of the input size and thus reduce to Theorem 12. ◀

On the other hand, the fact that PURPLE with recursion can hold more than a constant number of pointers in memory albeit according to a stack discipline enables it to solve HORN.

▶ **Theorem 21.** PURPLE *with recursion can solve* HORN *and more generally evaluated formulas in LFP logic.*

5 Deterministic transitive closure logic

In the context of descriptive complexity theory deterministic transitive closure logic (DTC-logic) was introduced as a logical characterisation of LOGSPACE on ordered structures [16]. This logic is parametrized by a relational signature σ. Its syntax extends that of first-order logic with equality over the signature σ by a construct $\texttt{dtc}\varphi\vec{x}\vec{y}\vec{s}\vec{t}$ for deterministic transitive closure. The deterministic transitive closure of a binary relation R is the transitive closure of the relation $R_d = \{\langle x, y\rangle \mid xRy \wedge (\forall z.\, xRz \Rightarrow z = y)\}$ and the formula $\texttt{dtc}\varphi\vec{x}\vec{y}\vec{s}\vec{t}$ expresses that the pair $\langle \vec{s}, \vec{t}\rangle$ is in the deterministic transitive closure R_d of the binary relation on tuples that is defined by $\vec{x}R\vec{y} \iff \varphi(\vec{x}, \vec{y})$.

Note that R_d is a partial function in the sense that $R_d(x, y) \wedge R_d(x, y')$ implies $y = y'$ and that if R itself is a partial function then $R = R_d$.

Deterministic transitive closure logic captures LOGSPACE on finite structures with a total ordering, i.e. where σ contains a binary relation lt that is interpreted as a total ordering, see e.g. [4]. Informally, this is because with a total ordering one can do enough arithmetic in the logic to encode work-tapes of LOGSPACE Turing Machines and thus simulate the computation of such machines using the DTC-operator.

On unordered structures, however, DTC-logic is extremely weak, see [9].

An interesting, yet less studied, middle ground is DTC-logic on d-labelled graphs (called locally-ordered graphs in this context), where the edges emanating from any given node carry a linear order but the nodes themselves do not. This can be formally represented in a number of equivalent ways. On such inputs, DTC-logic can simulated (deterministic) JAGs by taking the transitive closure of the transition relation. On the other hand, PURPLE can evaluate DTC-formulas:

▶ Proposition 21.1. For each closed DTC-formula φ on locally ordered graphs there exists a program P_φ such that, for any finite locally ordered graph G, $G \models \varphi$ holds if and only if P_φ recognises G.

In order to evaluate quantifiers we use the `forall`-loop to search for witnesses or counter-examples. For transitive closure we first use `forall`-loops to find the (uniquely determined) successor of any tuple; the transitive closure can then be simulated using a while-loop.

The converse of this proposition is not true since the parity of the input size is not definable in DTC-logic [6]. The following direct consequence of Proposition 21.1 and Theorem 13 provides an answer to question left open by Etessami & Immerman [6].

▶ **Corollary 22** ([14]). USTCON *in locally-ordered graphs is not definable in* DTC*-logic.*

6 Conclusion and Future Works

Our starting point was the observation that LOGSPACE and NLOGSPACE algorithms operating on graph-like structures and then can hold a constant number of pointers into the input structure in memory. In addition, they can perform arithmetic up to the size of the input and, finally, can access the binary representation of the input nodes in the form of a fixed but arbitrary linear order.

We then considered that it might be possible to obtain "relativized" separation results if some of these features are removed by considering pointers as an abstract datatype.

We surveyed existing systems that are based on this idea namely jumping automata on graphs, transitive closure logic, and the PURPLE programming language. All of these systems are provably below PTIME but somewhat trivially so because they cannot solve USTCON which

lies in LOGSPACE. We thus considered various extensions of these systems with arithmetic, nondeterminism, iterators, recursion and examined the strength of the resulting systems.

In the light of these results, the original motivation of obtaining a "relativised" separation of LOGSPACE from PTIME has become somewhat elusive: most systems considered either coincide with LOGSPACE, NLOGSPACE, PTIME or there exists a LOGSPACE problem that can provably not be decided in them (USTCON, tree isomorphism) and can be reduced to HORN.

A possible way to address this, is to consider a weak system and add to it constructs that will solve LOGSPACE problems. For instance, to strengthen PURPLE with counting and nondeterminism over graphs with no edge ordering can be strengthened with recursion in tree-like structures in order to solve tree-isomorphism. Such a construct has been used in [11], but their motivation differs from ours. However, this might require us to add such constructs for many LOGSPACE complete problems since weaker systems would not necessarily capture the reductions between the different LOGSPACE complete problems.

Another option could be to look for a subset of HORN instances to which known LOGSPACE complete problems such as USTCON or tree isomorphism cannot be reduced yet still require a non-constant number of pebbles to be solved. Some initial progress has been made by restricting graph-theoretical parameters such as tree width of the graph induced by a HORN instance.

References

1. Daniel P. Bovet and Pierluigi Crescenzi. *Introduction to the theory of complexity*. Prentice Hall international series in computer science. Prentice Hall, 1994.
2. Stephen A. Cook and Charles Rackoff. Space lower bounds for maze threadability on restricted machines. *SIAM J. Comput.*, 9(3):636–652, 1980.
3. Stephen A. Cook and Ravi Sethi. Storage requirements for deterministic polynomial time recognizable languages. *J. Comput. Syst. Sci.*, 13(1):25–37, 1976.
4. H.D. Ebbinghaus and J. Flum. *Finite Model Theory*. Springer-Verlag, 1995.
5. J. Edmonds, C. Poon, and D. Achlioptas. Tight lower bounds for st-connectivity on the nnjag model. *SIAM Journal on Computing*, 28(6):2257–2284, 1999.
6. K. Etessami and N. Immerman. Reachability and the power of local ordering. *Theoretical Computer Science*, 148(2):261–279, 1995.
7. Kousha Etessami and Neil Immerman. Reachability and the power of local ordering. *Th. Comp. Sci.*, 148(2):261–279, 1995.
8. Kousha Etessami and Neil Immerman. Tree canonization and transitive closure. In *IEEE Symp. Logic In Comput. Sci*, pages 331–341, 1995.
9. E. Grädel and G.L. McColm. On the power of deterministic transitive closures. *Information and Computation*, 119(1):129–135, 1995.
10. Erich Grädel and Gregory L. McColm. On the power of deterministic transitive closures. *Inf. Comput.*, 119(1):129–135, 1995.
11. Martin Grohe, Berit Grußien, André Hernich, and Bastian Laubner. L-recursion and a new logic for logarithmic space. In *CSL*, pages 277–291, 2011.
12. Martin Hofmann and Ramyaa Ramyaa. Power of nondetreministic jags on cayley graphs, 2013. arXiv, 835993.
13. Martin Hofmann, Ramyaa Ramyaa, and Ulrich Schöpp. Pure pointer programs and tree isomorphism. In Frank Pfenning, editor, *FoSSaCS*, volume 7794 of *Lecture Notes in Computer Science*, pages 321–336. Springer, 2013.
14. Martin Hofmann and Ulrich Schöpp. Pointer programs and undirected reachability. In *LICS*, pages 133–142, 2009.

15 Martin Hofmann and Ulrich Schöpp. Pure pointer programs with iteration. *ACM Trans. Comput. Log.*, 11(4), 2010.

16 Neil Immerman. Nondeterministic space is closed under complementation. *SIAM J. Comput.*, 17(5):935–938, October 1988.

17 Steven Lindell. A logspace algorithm for tree canonization (extended abstract). STOC '92, pages 400–404, New York, NY, USA, 1992. ACM.

18 Lu, Zhang, Poon, and Cai. Simulating undirected *st*-connectivity algorithms on uniform jags and nnjags. In *Algo. and Comp.*, volume 3827 of *LNCS*. 2005.

19 Omer Reingold. Undirected connectivity in log-space. *J. ACM*, 55(4), 2008.

20 Ulrich Schöpp. A formalised lower bound on undirected graph reachability. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *LPAR*, volume 5330 of *Lecture Notes in Computer Science*, pages 621–635. Springer, 2008.