# Pointer Programs and Undirected Reachability

Martin Hofmann
Institut für Informatik
Ludwig-Maximilians-Universität München
Munich, Germany

Ulrich Schöpp
Institut für Informatik
Ludwig-Maximilians-Universität München
Munich, Germany

## Abstract

*Pointer programs are a model of structured computation within* LOGSPACE. *They capture the common description of* LOGSPACE *algorithms as programs that take as input some structured data (*e.g. *a graph) and that store in memory only a constant number of pointers to the input (*e.g. *to the graph nodes). In this paper we study undirected* **s**-**t**-*reachability for a class of pure pointer programs in which one can work with a constant number of abstract pointers, but not with arbitrary data, such as memory registers of logarithmic size. In earlier work we have formalised this class as a programming language* PURPLE *that features a* `forall`*-loop for iterating over the input structure and thus subsumes other formalisations of pure pointer programs, such as Jumping Automata on Graphs (*JAG*s) and Deterministic Transitive Closure logic (*DTC*-logic) for locally ordered graphs.*

*In this paper we show that* PURPLE *cannot decide undirected* **s**-**t**-*reachability, even though there does exist a* LOGSPACE*-algorithm for this problem by Reingold's theorem. As a corollary we obtain that* DTC*-logic for locally ordered graphs cannot express undirected* **s**-**t**-*reachability.*

## 1 Introduction

The complexity class LOGSPACE is defined by Turing Machines that have read-only access to their input tape and that besides their finite control may use a constant number of tapes of logarithmic size. Many LOGSPACE-algorithms, however, are intended to take as input some structured data and they use their logarithmic memory only to store pointers into the input structure. A typical input structure would be a graph; in logarithmic space one can store a constant number of references to graph nodes ('pointers').

This observation leads to the idealisation of LOGSPACE-algorithms as pointer programs that access a structured read-only input by means of a constant number of abstract pointers and that compute their result by pointer manipulation alone. Many typical LOGSPACE-algorithms are presented informally in this way, see *e.g.* [1].

In this paper we study a restriction of LOGSPACE in which one can work with a constant number of abstract pointers, but not with other data, such as counting registers of logarithmic length. This is motivated by the observation that while many typical LOGSPACE-complete algorithms are presented as abstract pointer programs, there also exist natural problems in LOGSPACE for which no such presentation is known. One example is **s**-**t**-reachability in undirected graphs, where the question is whether two nodes **s** and **t** are connected by a path. This problem can be decided in logarithmic space using Reingold's algorithm, but the algorithm uses counting registers of logarithmic size in addition to abstract pointers. We are thus led to ask if natural LOGSPACE problems such as undirected **s**-**t**-reachability can be solved by using abstract pointers alone.

The main result in this paper is that no program with a fixed number of abstract pointers and boolean variables can decide reachability in undirected graphs. Moreover, since formulas in *locally-ordered deterministic transitive closure logic* (DTC-logic) can be evaluated by such programs we obtain as a corollary that undirected reachability cannot be defined in DTC-logic [4], thus answering a question left open by Etessami & Immerman [5].

Of course, in order to be able to prove inexpressibility results of this kind one needs to be precise about what 'computation with a constant number of abstract pointers' means. In our previous paper [6] we have introduced the language PURPLE (for PURe Pointer LanguagE) to make that intuition precise.

PURPLE extends previous formalisms with a similar intuition, such as *jumping automata on graphs*

(JAGs) [2] or DTC-logic, with an iteration construct which traverses all nodes of the input structure in an unspecified order.

Thus PURPLE overcomes some artificial weaknesses of these previous formalisms, which prevent them from fully capturing the notion of computation with abstract pointers. In particular, a JAG can only ever visit the part of the input that is reachable from its initial configuration; it can therefore not determine whether the input has a self loop or similar. The iteration construct of PURPLE visits all nodes and allows one to test for such properties. The quantifiers of DTC-logic provide a similar iteration facility, yet seem to be unnecessarily restricted from a programming perspective. Concretely, with PURPLE-iteration we can compute the parity of the input size, while this cannot be done in DTC-logic.

These weaknesses have been recognized before and various add-ons have been proposed, all of which however compromise abstractness of pointers. For example, DTC-logic has been enriched with a total order on the input structure thus capturing all of LOGSPACE. Similarly, JAGs have been extended to allow access to the internal representation of pointers. For example, node-named JAGs (NNJAGs) [8] can inspect the internal structure of pointers qua bitstrings. See [6] for a more detailed discussion of related formalisms.

Graph reachability problems have been studied for JAGs by Cook & Rackoff [2] and for extensions such as NNJAGs by Poon, Edmonds and others [8, 3]. In this paper we build on the results of Cook & Rackoff for JAGs. Existing techniques for more expressive models such as NNJAGs do not appear to be directly applicable to PURPLE, since the primitives that PURPLE and NNJAGs add to JAGs are quite different in nature. Moreover, since NNJAGs can inspect the internal structure of pointers, they do not seem to be the right tool for studying the expressivity of DTC-logic.

### 1.1 Pure Pointer Language

The syntax of PURPLE for locally ordered graphs appears in Figure 1. The terms of boolean type are given by the usual boolean operations together with an equality test for pointer terms. Pointer terms are built from pointer variables, which denote graph nodes, the two node constants **s** and **t** and the operation $t.succ(i)$, which denotes the node obtained by moving along edge number $i$ from the node denoted by $t$.

We remark that the variable $x$ is not bound in the program `forall` $x$ `do` $M$. A PURPLE-program has a fixed number of global variables, which include the ones in `forall`-loops.

The evaluation of PURPLE programs is standard except for the `forall`-loop. A loop `forall` $x$ `do` $M$ is evaluated by setting the pointer variable $x$ successively to all nodes in the input structure *in some arbitrary order* and evaluating the body $M$ after each such setting.

In general, the final state reached after executing such a program will depend on the order of traversal chosen for the `forall`-loops in it. However, for a program to decide a given property we require that it will yield the correct result (`true` or `false`) irrespective of the traversal order.

To define precisely what it means for a program to recognise a set of graphs, we choose a boolean variable *result* that indicates the outcome of a computation. A program $M$ then *recognises* a set $X$ of finite graphs if all runs of $M$ with an input graph in $X$ halt in a configuration with *result* = `true`; and all runs of $M$ with an input graph not in $X$ halt in a configuration with *result* = `false`. For example, the program *result* := `false`; `forall` $x$ `do` *result* := $\neg$*result* recognises the set of graphs with an even number of nodes; for more examples see [6]. A program whose result depends on the ordering chosen in the `forall`-loops, or that does not terminate, does not recognise any set of graphs at all.

The independence from the evaluation order in the `forall`-loops ensures that pointers remain pure and that their internal representation cannot be inspected by the program. We have shown [6] that pure pointer programs with iteration, while being able to express interesting problems, do in general only have a very limited ability for counting and therefore are not able to encode the tapes of a Turing Machine in the pointers. This is important, since our aim is to find out what can be done if one can store only abstract pointers and not arbitrary data.

An encoding of tapes as pointers is possible for example in DTC-logic with a total order, which then restores the full strength of LOGSPACE for that system.

## 2 Overview

The purpose of this paper is to show that one cannot write a PURPLE-program that decides whether or not two nodes **s** and **t** in an undirected graph are connected by a path. Before we embark on the proof, let us give a brief proof outline.

The basic idea is to construct, for any given PURPLE program, a graph and a loop-free program that simulates the original PURPLE program on this graph. This means that from an arbitrary start configuration the computation of the loop-free program ends in a config-

$$
\begin{array}{lllll}
\text{Boolean terms} & t^{\mathtt{bool}} & ::= & x^{\mathtt{bool}} \mid \mathtt{true} \mid \mathtt{false} \mid \neg t^{\mathtt{bool}} \mid t^{\mathtt{bool}} \wedge t^{\mathtt{bool}} \mid t^{\mathtt{bool}} \vee t^{\mathtt{bool}} \mid t^{\Gamma} = t^{\Gamma} \\
\text{Pointer terms} & t^{\Gamma} & ::= & x^{\Gamma} \mid \mathtt{s} \mid \mathtt{t} \mid t^{\Gamma}.succ(i) \qquad (\text{where } i \in \mathbb{N}) \\
\text{Programs} & M & ::= & \mathtt{skip} \mid M; M \mid x^{\Gamma} := t^{\Gamma} \mid x^{\mathtt{bool}} := t^{\mathtt{bool}} \\
 & & & \quad \mid \mathtt{if}\ t^{\mathtt{bool}}\ \mathtt{then}\ M\ \mathtt{else}\ M \mid \mathtt{while}\ t^{\mathtt{bool}}\ \mathtt{do}\ M \mid \mathtt{forall}\ x^{\Gamma}\ \mathtt{do}\ M
\end{array}
$$

**Figure 1. Syntax of Pure Pointer Programs**

uration that is also reached by at least one run of the original PURPLE-program.

With this simulation, it then suffices to show that the simulating loop-free program cannot decide undirected reachability, as the acceptance condition of PURPLE is such that a program must yield the same result in all runs – and the loop-free program implements one such run.

Without the `forall`-loop a program can reach new nodes only by moving pointers along edges. Therefore, a loop-free program can only visit nodes within a certain radius of the initial positions of the pointer variables. This radius is trivially bounded by the size of the program.

Our construction of the graph and loop-free program will be such that the graph consists of two disjoint copies of the same graph and that the radius of the loop-free program is less than half of the diameter of that graph. In this situation, the loop-free program cannot distinguish whether we place `s` and `t` far apart on one connected component or whether we place them on different components. It can therefore not decide undirected reachability, which implies the same for the original PURPLE program.

The main work in the proof therefore goes into eliminating loops from PURPLE programs on certain graphs and into constructing graphs on which the thus obtained simulating program has radius less than half the graph's diameter.

For the elimination of loops, we first note that the `while`-construct can be eliminated from PURPLE-programs by replacing it with an appropriate number of nested `forall`-loops; see [6, 10] for details. We can therefore restrict our attention to programs without `while`-loops.

## 2.1 Elimination of `forall`-Loops

Consider thus a program `forall` $x$ `do` $M$ in which all loops have already been eliminated from $M$.

The graph we run this program on has an astronomic number of nodes compared to radius of $M$ and was chosen already with the elimination of this last loop in mind.

This graph also has a very homogeneous structure. To a program all nodes look exactly the same and it is hard to avoid going around in circles. Take the perpetual repetition of a loop-free program $N$ of small radius, *i.e.* consider $N^{\$} \equiv \mathtt{while\ true\ do}\ N$. Let us call the contents of the variables prior to the evaluation of a program its *known nodes*. Then the homogeneous graph structure traps $N^{\$}$ within a neighbourhood of radius $R$ around its known nodes, where $R$ is a number that is exponential in the radius of $N$ but that is still small compared to the overall graph size.

Even though the original program `forall` $x$ `do` $M$ is not confined to a small area of the graph, its freedom is severely constrained by the homogeneous graph structure. The constraints are such that some runs of the original program can in essence be simulated by a program of the form $N^{\$}$, where $N$ is loop-free. These runs are therefore constricted by the graph structure, much like $N^{\$}$ is.

To explain which runs of `forall` $x$ `do` $M$ can be simulated by $N^{\$}$, let us concretely define the loop-free program $N$. It has access to $k+1$ additional copies of the graph, where $k$ is the number of variables in $M$. We assume that a new variable is placed on a certain fixed location on each new copy of the graph – it does not matter where, due to the uniform structure of the graph. The program $N$ first places $x$ on one of the new components of the graph that does not contain any variable from $M$ and then just executes $M$. It records in boolean variables which of the connected components contain variables from $M$, so that even in a repeated execution of $N$ the variable $x$ is always first placed on an empty component.

By construction, the program $N^{\$}$ simulates the runs of `forall` $x$ `do` $M$ in which $x$ is always placed infinitely far away from all other variables, *i.e.* on a different connected component. By using the fact that $N^{\$}$ is confined to neighbourhoods of some small radius $R$, we can show that $N^{\$}$ also simulates the runs of `forall` $x$ `do` $M$ in which any node presented as destination for $x$ is more than $2R$ apart from both the known nodes and the nodes presented earlier in the computation. In runs of `forall` $x$ `do` $M$ of this kind the pointer variables must at any time lie in an $R$-neighbourhood either around a

known node or around a node chosen previously as destination for $x$. That the destinations for $x$ are always chosen to have distance at least $2R$ from the known nodes and the previous destinations for $x$, makes runs of this kind therefore look just like runs in which $x$ is always placed infinitely far away from all other nodes.

We can now use these observations on the relation of the programs `forall` $x$ `do` $M$ and $N^{\$}$ to estimate the final position of the pointer variables in certain runs of `forall` $x$ `do` $M$. Let us call a sequence of nodes $v_1, \ldots, v_k$ *distant* if each $v_i$ is more than $2R$ apart from the known nodes and from $v_j$ for $i \neq j$. If we evaluate (abusing notation) $x := v_1;\ M;\ \ldots;\ x := v_k;\ M$ then the final state will consist of nodes that are $R$-close to the known nodes and the $v_i$. But in fact, due to periodicity, $M$ will only be able to remember the first few $v_i$ and the last few $v_i$, so that the final state will be $R$-close to the known nodes and these first and last few $v_i$ (Lemma 2).

If we use a distant sequence whose first and last 'few' nodes are in the vicinity of the known nodes, then we can simulate $x := v_1;\ M;\ \ldots;\ x := v_k;\ M$ by a loop-free program that is large enough to fully explore that vicinity. This loop-free program will thus consist of a big case distinction over the possible neighbourhoods of some radius around the known nodes. In each case the corresponding final state is produced in one go by an appropriate sequence of assignments. In section 3.2 we introduce a 'macro notation' for loop-free programs of this special kind and call them *abstract local programs*.

Alas, no distant sequence can enumerate the entire graph. However, again appealing to the short memory of $M$, we can use instead of a distant sequence a *sightseer sequence*, which has the property that each of its nodes is more than $2R$ apart from the known nodes, the first few nodes, and from the last few nodes preceding it in the sequence.

This then suggests to choose an iteration sequence for the `forall`-loop in the following way. We begin with a sightseer sequence whose few initial nodes and few last nodes (those that might be remembered) are in the vicinity of the known nodes. We choose $u$ large enough so that the $u$-neighbourhood of the known nodes contains that vicinity. The sightseer sequence can be chosen so that it traverses all nodes outside this $u$-neighbourhood. After the iteration through the sightseer sequence all variables will therefore refer to nodes that are $u$-close to the known nodes. We then iterate through the remaining $n$ nodes in the $u$-neighbourhood in an arbitrary order, thus completing the `forall`-iteration. The periodicity argument no longer applies to those last steps, which means that variables could cover a further distance of $nR$ steps. The final state will therefore consist of nodes that are $u + nR$-close to the known nodes. The run of `forall` $x$ `do` $M$ that we have chosen in this way can therefore be implemented by an abstract local program of radius $u + nR$ (Theorem 3). Here, we have glossed over the possibility that during the iteration $M$ might explore nodes a little, *i.e.* $R$, beyond the $u$-neighbourhood of known nodes.

# 3 Locality of Iteration on Action Graphs

In this section we make precise the outlined argument for simulating general PURPLE programs by loop-free programs. We do so on a certain general class of action graphs, and give conditions under which the iteration `forall` $x$ `do` $M$ of a loop-free program $M$ can be implemented itself by a loop-free program. Since `while`-loops can be expanded into `forall`-loops, we can then use this result to simulate all PURPLE-programs by loop-free programs on these graphs.

Moreover, we establish an upper bound on the size of the loop-free program that implements `forall` $x$ `do` $M$. Note that on general graphs it will not be possible to derive good bounds. For instance, with the `forall`-loop one can write a program that places a pointer variable on some node with a self-loop. But a graph may have just a single self-loop, which may be arbitrarily far away from the start configuration. Hence, in the worst case a loop-free program that simulates this PURPLE program will have to be about as large as the graph itself.

However, this issue does not arise if we consider only graphs in which all nodes have isomorphic neighbourhoods. In this paper we work with a class of such graphs, the *free action graphs*, which we define next.

## 3.1 Free Action Graphs

Action graphs are locally ordered graphs defined by a group action on a finite set. They are interesting for our purposes since we can use existing results [9] to analyse the behaviour of programs on them, allowing us to prove Theorem 1 below.

All groups in this paper are finite. We write them multiplicatively and denote the unit element of group $G$ by $e_G$ or just $e$. The *exponent* $\exp(G)$ of a group $G$ is the least number $n$ such that $g^n = e$ holds for all $g \in G$.

A *group action* of a group $G$ on a set $V$ is a function $(-) \cdot (-) \colon V \times G \to V$ that satisfies $v \cdot e = v$ and $(v \cdot g) \cdot h = v \cdot (g \cdot h)$ for all $v \in V$ and $g, h \in G$. A group action is *free* if $v \cdot g = v$ implies $g = e_G$ for all $v \in V$ and $g \in G$.

**Definition 1** (Free Action Graph)**.** Let $V$ be a non-empty finite set, $G$ be a group, $S = g_1, \ldots, g_d$ be a list of generators for $G$ that for each of its elements also contains the inverse, and let $(-) \cdot (-) \colon V \times G \to V$ be a free group action of $G$ on $V$. The *free action graph* $A(V, G, S, \cdot)$ is the undirected locally ordered graph of degree $d$, whose node set is $V$ and in which the $i$-th edge from node $v$ goes to $v \cdot g_i$. We say that $G$ is the *edge group* of $A(V, G, S, \cdot)$.

In a free action graph there exists for any two nodes $v$ and $w$ in the same connected component a unique element $(w/v) \in G$ such that $v \cdot (w/v) = w$ holds.

Cayley graphs are prominent examples of free action graphs. If $G$ is a group and $S$ is a list of generators for $G$ that is closed under inverses, then the *Cayley graph* $C(G, S)$ is the undirected graph with node set $G$ and edge set $\{(g, gs) \mid g \in G, s \in S\}$. In fact, any free action graph with edge group $G$ and generators $S$ is isomorphic to a number of disjoint copies of $C(G, S)$.

We write $d(v, w)$ for the distance of the two graph nodes $v$ and $w$. We define the $r$-ball around a node $v$ in $A$ by $\mathrm{B}_A(v, r) = \{w \mid d(v, w) \leq r\}$ and extend this notion to sets of nodes by $\mathrm{B}_A(W, r) = \bigcup_{w \in W} \mathrm{B}_A(w, r)$. Notice that an element $\pi \in \mathrm{B}_{C(G,S)}(e_G, r)$ corresponds to a move along a path of length at most $r$ in $A = A(V, G, S, \cdot)$, since we have $v \cdot \pi \in \mathrm{B}_A(v, r)$ for all $v \in V$.

We have decided to postpone the concrete definition of the graphs used until Section 4.1 and to work with universally quantified graphs whose properties are required as needed until then. It is possible to read Section 4.1 first if a concrete instance is preferred.

### 3.2 Abstract Local Programs

To simplify the further development, we introduce *abstract local programs* as a syntax-free abstraction of loop-free programs on free action graphs. This abstraction allows us to consider the computation of a loop-free program as one single large step rather than a sequence of (for our purpose uninteresting) small steps. Also, the important parameters influencing the necessary graph constructions are more directly visible for abstract local programs.

An abstract local program comprises a finite set of states $Q$, a finite set of pebbles $P$, a radius $r \in \mathbb{N}$ and a transition function which given a state $q \in Q$ and a description of the $r$-neighbourhood around the pebble locations returns a new state $q' \in Q$ and for each pebble an instruction that displaces it to some position within the $r$-neighbourhood of the initial pebble positions. We define the transition function in terms of an edge group $G$ with generators $S$, so that a single abstract local program can be used on any free action graph with edge group $G$ and generators $S$.

In particular, the $r$-neighbourhood of the current pebble positions is presented in the form of a partial function of type $P \times P \rightharpoonup \mathrm{B}_{C(G,S)}(e_G, 2r)$ which for any two pebbles provides their distance in the form of a group element in $\mathrm{B}_{C(G,S)}(e_G, 2r)$ or is undefined meaning that the two pebbles in question have distance greater than $2r$. Every concrete environment $\rho \in A^P$ for graph $A$ induces such a map $[\rho]_r$ given by $[\rho]_r(x, y) = \rho(y)/\rho(x)$ if $d(\rho(x), \rho(y)) \leq 2r$ and $[\rho]_r(x, y)$ undefined otherwise. We write $\Sigma_{G,S}(P, r)$ for the set of all such maps that arise as $[\rho]_r$ for some $\rho \in A^P$ and some free action graph $A = A(V, G, S, \cdot)$. It is possible to characterise $\Sigma_{G,S}(P, r)$ axiomatically, but there is no need to do so here.

**Definition 2** (Abstract local program)**.** Let $G$ be a finite group and $S$ be a list of generators that is closed under inverses. An *abstract local program* over $G$ and $S$ with finite state set $Q$, finite pebble set $P$ and local radius $r \in \mathbb{N}$ is a function

$$f \colon Q \times \Sigma_{G,S}(P, r) \longrightarrow Q \times P^P \times \mathrm{B}_{C(G,S)}(e_G, r)^P.$$

We write $L_{G,S}(Q, P, r)$ for the set of all such abstract local programs. For an action graph $A$ with edge group $G$ and generators $S$, we also write $L_A(Q, P, r)$ for $L_{G,S}(Q, P, r)$.

The intended meaning of the abstract local program $f$ is that it first looks at the state in $Q$ and the $r$-ball around the pebble locations. With this input, the function $f$ yields a final state in $Q$ and two functions $j \in P^P$ and $m \in \mathrm{B}_{C(G,S)}(e_G, r)^P$ that describe pebble moves in the following sense: each pebble $x \in P$ is first jumped to the position of pebble $j(x)$ in the start configuration and then moved along the path $m(x)$.

Formally, this behaviour of $f$ on a free action graph $A$ is captured as follows. We call any pair $(q, \rho) \in Q \times A^P$ a *configuration*. The behaviour of $f$ is then given by function $[\![f]\!]_A \colon Q \times A^P \to Q \times A^P$ such that $[\![f]\!]_A(q, \rho)$ is the configuration $(p, \tau)$ defined by $(p, j, m) := f(q, [\rho]_r)$ and $\tau(x) := \rho(j(x)) \cdot m(x)$.

In this way, abstract local programs can describe in one single step the effect of fully executing a loop-free program. It is straightforward to construct for each loop-free program an abstract local program (whose state set consists of valuations of boolean variables and whose pebbles are the pointer variables) with the same effect and vice versa.

**Why Abstract Local Programs?** It would be possible to replace abstract local programs by loop-free

PURPLE-programs or indeed JAGs, but we find that the advantage of saving a construct would be outweighed by higher complexity in subsequent calculations. For example, even to explore a neighbourhood of radius $r$ a JAG needs to be iterated, its transition function being able to inspect only the adjacency relation of the pebbles. One would then have to distinguish this pro forma iteration from actual, possibly non-terminating iteration. Loop-free PURPLE programs, on the other hand, have elements of the finite state both in the boolean variables and in the program counter. One would thus need to define some artificial complexity measure for them. Moreover, the radius of a loop-free program is not explicit and would have to be defined essentially by repeating the concept of abstract local programs.

### 3.3 Iterating Abstract Local Programs

We next define *range* and *modulus* for abstract local programs with state set $Q$, pebble set $P$ and local radius $r$ on free action graphs with edge group $G$ and generators $S$. The range bounds how far the pebbles may be moved in a repeated execution of the program and the modulus bounds the number of configurations appearing in this computation.

**Definition 3** (Range)**.** Define $range_{G,S}(Q,P,r)$ to be the smallest number such that, for all $f \in L_{G,S}(Q,P,r)$ and $k \in \mathbb{N}$ there exists $g \in L_{G,S}(Q,P,range_{G,S}(Q,P,r))$ satisfying $[\![f]\!]^k_A = [\![g]\!]_A$ for any free action graph $A$ with edge group $G$ and generators $S$.

Let us illustrate the definition of range on the example of an abstract local program $f$ with local radius 1 that implements the simple PURPLE program $x := x.succ(i)$. Such an abstract local program $f$ will be defined by a transition function that maps $(q,\sigma) \in Q \times \Sigma_{G,S}(P,r)$ to $(q, id, m) \in Q \times P^P \times \mathrm{B}_{C(G,S)}(e_G,1)^P$, where $m$ maps $x$ to $g_i$, the $i$-th generator of the free action graph, and everything else to the unit element $e_G$. The question is now: how far can pebbles be moved if we execute $f$ a certain number of times, say $k$ times? For this simple $f$, it is easy to write down explicitly how its $k$-fold application behaves, *i.e.* we can define directly an abstract local program $g$ satisfying $[\![f]\!]^k_A = [\![g]\!]_A$ for any free action graph $A = A(V,G,S,\cdot)$. We may define $g$ by $g(q,\sigma) = (q, id, m)$, where $m(x)$ is $g_i^k$ and $m(y) = e$ for all $y \neq x$. We can take the local radius of $g$ to be the exponent of the group $G$, as any $g_i^k$ must be in an $\exp(G)$-ball around $e_G$ in $C(G,S)$ by definition of the exponent. This then shows that the exponent $\exp(G)$ of the edge group is an upper bound for the range of $f$. If we run $f$ over and over again, the pebble $x$ can therefore never leave the $\exp(G)$-ball of its initial position, even though it will be moved in each step.

**Definition 4** (Modulus)**.** We let $modulus_{G,S}(Q,P,r)$ be the smallest number such that, for all $f \in L_{G,S}(Q,P,r)$ and all $(q,\rho) \in Q \times A^P$, the set $\{[\![f]\!]^k(q,\rho) \mid k \in \mathbb{N}\}$ has cardinality at most $modulus_{G,S}(Q,P,r)$.

**Theorem 1.** *There exists a constant $c$, such that for all $G$, $S$, $Q$, $P$ and $r$ we have:*

$$\begin{aligned} range_{G,S}(Q,P,r) &\leq ((|S|+1)^r \cdot |Q| \cdot \exp(G))^{c^{|P|}} \\ modulus_{G,S}(Q,P,r) &\leq \\ & |Q| \cdot |P|^{|P|} \cdot (|S|+1)^{range_{G,S}(Q,P,r)\cdot|P|} \end{aligned}$$

*Proof idea.* The first inequality is a consequence of Prop. 3 of [9]. This proposition asserts the existence of a constant $d$ such that a JAG with $q$ states and $p$ pebbles can visit no more than $(q \cdot \exp(G))^{d^p}$ nodes in the course of any computation on an action graph with edge group $G$. Now, an abstract local program with local radius $r$ can be implemented by a JAG that has enough states to fully explore the $r$-neighbourhood of its start configuration ($(|S|+1)^r$ bounds the number of nodes therein). The inequality then follows by applying Prop. 3 of [9] to that JAG.

The second inequality follows as an immediate consequence of the first. The right-hand side is an upper bound on the number of configurations that place all pebbles within a radius $range_{G,S}(Q,P,r)$ of some start configuration. □

### 3.4 Locality of Iteration

Our aim is thus to show that (one run of) each PURPLE-program can be implemented by an abstract local program with a small enough local radius. In this section we give conditions under which any `forall`-loop can be implemented by an abstract program whose local radius is small enough for our purposes (Theorem 3 below).

Suppose we have a PURPLE program `forall` $x$ `do` $M$ and we have already compiled the loop body $M$ into an abstract local program $f \in L_{G,S}(Q,P,r)$. We would then like to find a local program $g$ that implements some run of the whole program `forall` $x$ `do` $M$. To do this, it suffices to implement a `forall`-like iteration of $f$, defined as follows.

**Definition 5.** Let $A$ be a free action graph, $f \in L_A(Q,P,r)$ be an abstract local program and $\boldsymbol{z} \in P$ be a pebble. We say that $g \in L_A(Q,P,l)$ *implements* `forall` $\boldsymbol{z}$ `do` $f$ *on* $A$, if for all $(q,\rho) \in Q \times A^P$ there is an

enumeration $v_1, \ldots, v_n$ of all nodes in $A$, so that if we define a sequence of configurations $(q_0, \rho_0), \ldots, (q_n, \rho_n)$ by $(q_0, \rho_0) = (q, \rho)$ and $(q_{i+1}, \rho_{i+1}) = [\![f]\!]_A(q_i, \rho_i[\boldsymbol{z} := v_{i+1}])$, then we have $[\![g]\!]_A(q, \rho) = (q_n, \rho_n)$.

In the following we let $Q$ range over sets of states, $P$ over sets of pebbles and $r$ over local radii.

**Definition 6.** A set of nodes $U$ in a graph $\Gamma$ is *$r$-scattered* if $\mathrm{B}_\Gamma(u, r) \cap \mathrm{B}_\Gamma(v, r) = \emptyset$ holds for all $u \neq v \in U$.

**Definition 7.** For any graph $\Gamma$ and any set of nodes $U$, the *$r$-size of $U$* is the size of the largest $r$-scattered subset of $U$.

**Definition 8.** A *sightseer enumeration with radius $r$ and memory $n$* is an enumeration $v_1, v_2, \ldots, v_k$ of the nodes of a graph such that the set $\{v_j \mid i - n \leq j \leq i \text{ and } 1 \leq j \leq k\}$ is $r$-scattered for all $i$.

This terminology is chosen by analogy with a sightseer who wants to go everywhere but never wants to see a place twice. If he can remember only the last $n$ places he visited and his range of vision is $r$, then he will be satisfied with an enumeration in which at any time his position and the $n$ previously visited nodes are an $r$-scattered set.

To simulate a `forall`-like iteration by a local program, we construct an ordering of the graph nodes such that if we present the graph nodes to the `forall`-loop in this order, then at the end of the computation all pebbles will lie in a small enough neighbourhood around the initial pebble positions. Even though the `forall`-loop will have temporarily placed pebbles outside this neighbourhood, this is not visible anymore in the final configuration and the move of the `forall`-loop from start to end configuration looks like the move of a local program that can only make moves within this neighbourhood. Thus we show that the `forall`-loop can be implemented by an abstract local program whose local radius is large enough to contain this neighbourhood.

To realise this simple plan, we need an understanding of where the pebbles will lie at the end of a `forall`-like iteration. To this end we prove the following lemma.

In the proof of this lemma and in the sequel we write $A + B$ for the tagged disjoint union of two sets $A$ and $B$, defined by $A + B = \{\mathrm{inl}\, x \mid x \in A\} \cup \{\mathrm{inr}\, y \mid y \in B\}$.

**Lemma 2.** *Let $f \in L_{G,S}(Q, P, r)$ be an abstract local program, $\boldsymbol{z} \in P$ be a pebble and $A$ be a free action graph with edge group $G$ and generators $S$. Assume*

$$R \geq \mathit{range}_{G,S}\left(Q \times (|P| + 2)^P, 2 \times P + 1, r\right),$$
$$M \geq \mathit{modulus}_{G,S}\left(Q \times (|P| + 2)^P, 2 \times P + 1, r\right).$$

*Then, for each $q \in Q$, each $c \in \Sigma_{G,S}(P, 2R)$ and each $k \in \mathbb{N}$, there exist a state $p \in Q$ and two functions $j \colon P \to (P + \mathbb{N})$ and $m \colon P \to \mathrm{B}_{C(G,S)}(e, R)$ with the following properties:*

1. *For any list of nodes $v_1, \ldots, v_k$ in $A$ and all $\rho \in A^P$ with $[\rho]_{2R} = c$, if the sequence $(q_0, \rho_0), \ldots, (q_k, \rho_k)$ defined by $(q_0, \rho_0) = (q, \rho)$ and $(q_{i+1}, \rho_{i+1}) = [\![f]\!]_A(q_i, \rho_i[\boldsymbol{z} := v_{i+1}])$ satisfies $\mathrm{B}_A(v_{i+1}, 3R) \cap (\mathrm{B}_A(\rho_i, 3R) \cup \mathrm{B}_A(\rho, 3R)) = \emptyset$ for all $i < k$, then we have $q_k = p$ and*

$$\rho_k(x) = \begin{cases} \rho(y) \cdot m(x) & \text{if } j(x) = \mathrm{inl}\, y, \\ v_i \cdot m(x) & \text{if } j(x) = \mathrm{inr}\, i. \end{cases}$$

2. *If $\mathrm{inr}\, i$ is in the image of $j$ then we have $1 \leq i \leq M$ or $k - |P| \cdot M < i \leq k$.*

*Proof sketch.* The idea is that the essence of a `forall`-like iteration of $f$ can be captured without knowing precisely where $\boldsymbol{z}$ is placed by the `forall` loop, so long as $\boldsymbol{z}$ is always placed far enough away from the other pebbles. This means that the iteration is adequately simulated if we always place $\boldsymbol{z}$ on a fresh copy of $C(G, S)$ instead of to the location chosen in the `forall`-loop. Such a simulation is useful, as it can be analysed using Theorem 1.

Let $q \in Q$, $c \in \Sigma_{G,S}(P, 2R)$ and $k \in \mathbb{N}$ be given. Choose $\rho^c \in A^P$ with $[\rho^c]_{2R} = c$ (we elide the trivial case where this is not possible). Let $B$ be the free action graph consisting of $A$ and $k$ disjoint copies of the Cayley graph $C(G, S)$. The node set of $B$ is $\{(v, 0) \mid v \in A\} \uplus \{(\pi, i) \mid \pi \in C(G, S),\ 1 \leq i \leq k\}$ and the group action is $(v, i) \cdot g = (v \cdot g, i)$. We identify $v \in A$ with $(v, 0) \in B$, so that, *e.g.*, we can view any $\rho \in A^P$ as an element of $B^P$. For $i \in \{0, \ldots, k\}$ we call the set of nodes of the form $(v, i)$ the *$i$-th component* of $B$.

Define a sequence $(q_0^c, \rho_0^c), \ldots, (q_k^c, \rho_k^c)$ of pairs in $Q \times B^P$ by $(q_0^c, \rho_0^c) := (q, \rho^c)$ and $(q_{i+1}^c, \rho_{i+1}^c) := [\![f]\!]_B(q_i^c, \rho_i^c[\boldsymbol{z} := (e, i + 1)])$. We then define the required state $p \in Q$ to be $q_k^c$ and choose the functions $j \colon P \to P + \mathbb{N}$ and $m \colon P \to \mathrm{B}_{C(G,S)}(e_G, R)$ so as to satisfy

- if $\rho_k^c(x) \in A$ then $j(x) = \mathrm{inl}\, y$ and $m(x) = \rho_k^c(x)/\rho_0^c(y)$ for some $y$ with $\rho_k^c(x) \in \mathrm{B}_A(\rho_0^c(y), R)$.
- if $\rho_k^c(x) = (\pi, l) \notin A$ then $j(x) = \mathrm{inr}\, l$ and $m(x) = \pi$.

We must show that such a choice is possible and that $p$, $j$ and $m$ have the required properties.

The proof of these points rests on showing the following property for all $i \leq k$:

$$\rho_i^c(P) \subseteq \mathrm{B}_B(\rho^c(P) \cup \{(e_G, i) \mid 1 \leq i \leq k\}, R) \qquad (1)$$

To show (1) we observe that there is an abstract local program $g$ such that the iteration of $[\![g]\!]$ on $(q_0^c, \rho_0^c)$ generates – up to permutation of the new copies of $C(G,S)$ – the sequence $(q_0^c, \rho_0^c), \ldots, (q_k^c, \rho_k^c)$. This program uses only the first $|P|+1$ new copies of $C(G,S)$ in $B$. It first places $\boldsymbol{z}$ on the node $(e,i)$, where $i > 0$ is the least number such that no other pebble from $P$ is placed on the $i$-th component copy of $B$, and then behaves like $f$. To implement the jump to $(e,i)$, the program has access to $|P|+1$ new pebbles, which are placed on $(e,1), \ldots, (e,|P|+1)$ and which are never moved. The program knows which component of $B$ to jump $\boldsymbol{z}$ to by keeping in its state a function $f \in \{0, \ldots, |P|+1\}^P$, such that $f(x) = i$ tells that pebble $x$ lies in the $i$-th component of $B$.

Precisely, $g$ is an abstract local program in

$$L_{G,S}\left(Q \times (|P|+2)^P, P \uplus \{x_1, \ldots, x_{|P|+1}\}, r\right)$$

such that, for any $i \leq k$, if we define $(\bar{q}_i^c, \bar{\rho}_i^c)$ to be $[\![g]\!]^i((q, \lambda x.0), \rho^c[x_1 := (e,1)] \ldots [x_{|P|+1} := (e, |P|+1)])$ then $q_i^c = \bar{q}_i^c$ and there exists a permutation $\sigma \in \mathbb{N}^{\mathbb{N}}$ with $\sigma(0) = 0$ such that $\rho_i^c(x) = (v,i)$ implies $\bar{\rho}_i^c(x) = (v, \sigma(i))$ for all $x \in P$. We omit the details of the definition of $g$ but observe that (1) follows immediately from its existence since $R$ bounds the range of $g$, by definition.

It now remains to show that $p$, $j$ and $m$ have the required properties. To this end let $\rho \in A^P$ and $v_1, \ldots, v_k$ be given as in the lemma. We now relate the sequence $(q_i, \rho_i)_i$ defined in the lemma to the sequence $(q_i^c, \rho_i^c)_i$ by proving the following four points for all $i \leq k$.

(a) $q_i^c = q_i$;

(b) if $\rho_i^c(x) \in A$ then $\rho_i(x) = \rho(y) \cdot \rho_i^c(x)/\rho^c(y)$ whenever $d(\rho_i^c(x), \rho^c(y)) \leq R$;

(c) if $\rho_i^c(x) = (\pi, l)$ then $\rho_i(x) = v_l \cdot \pi$;

(d) if $\rho_i^c(x) = (\pi, l)$ and $\rho_i^c(x') = (\pi', l')$ and $0 < l < l'$ then $d(v_l, v_{l'}) > 5R$.

These items are proved by induction on $i$, making use also of (1). The four items imply $[\rho_i]_r = [\rho_i^c]_r$, so that, up to the difference in the jump destination of $\boldsymbol{z}$, in both sequences the same the pebble moves are made to get to the next configuration. Item (d) is necessary to obtain $[\rho_i]_r = [\rho_i^c]_r$. To prove (d), we use the assumption on $v_1, \ldots, v_k$ that for any $i < k$ the jump destination $v_{i+1}$ is far enough away from $\rho_i$ and $\rho$. Claim (1) of the lemma then follows immediately from items (a)–(d).

For item (2), we use the above program $g$ to show that the pebble moves in the sequence $(q_i^c, \rho_i^c)_i$ become periodic after at most $M$ steps. Using this periodicity it can be shown that after the $i$-th step each pebble must be in the components $0, \ldots, M$ or $i - |P| \cdot M, \ldots, i$. □

With this lemma, we can now give sufficient conditions for the implementation of `forall`-loops by local programs of relatively small radius.

**Theorem 3.** *Let $f \in L_{G,S}(Q, P, r)$ be an abstract local program, $\boldsymbol{z} \in P$ be a pebble and $A$ be a free action graph with edge group $G$ and generators $S$. Assume*

$$R \geq 3 \cdot range_{G,S}\left(Q \times (|P|+2)^P, 2 \times P + 1, r\right),$$
$$M \geq modulus_{G,S}\left(Q \times (|P|+2)^P, 2 \times P + 1, r\right)$$

*and further that $u \geq 3R$ is a natural number such that the following two conditions hold:*

1. *The graph $A$ has a sightseer enumeration with radius $u$ and memory $|P| \cdot (M+1)$.*

2. *For any node $v \in A$ the $3R$-size of $\mathrm{B}_A(v, u - 3R)$ is at least $|P| \cdot (M+1)$.*

*Then there exists an abstract local program*

$$g \in L_{G,S}\left(Q, P, u + r \cdot |P| \cdot |\mathrm{B}_{C(G,S)}(e_G, u)|\right)$$

*that implements* `forall` $\boldsymbol{z}$ `do` $f$ *on $A$.*

*Proof.* Let $l := u + r \cdot |P| \cdot |\mathrm{B}_{C(G,S)}(e_G, u)|$. For the definition of $g$ let arguments $q \in Q$ and $c \in \Sigma_{G,S}(P, l)$ be given. We choose $\rho^c \in A^P$ with $[\rho^c]_l = c$ (if none exists then we can define $g(q,c)$ arbitrarily) and define $k = |A \setminus \mathrm{B}_A(\rho^c, u)| + (|P|+1)M$.

Define an enumeration $v_1^c, \ldots, v_n^c$ of the nodes in $A$ by:

- $v_1^c, \ldots, v_M^c, v_{k-|P|\cdot M+1}^c, \ldots, v_k^c$ form a $3R$-scattered subset of $\mathrm{B}_A(\rho^c, u-3R) \setminus \mathrm{B}_A(\rho^c, 3R)$ of size $|P| \cdot M$. We obtain such a set from a $3R$-scattered subset of $\mathrm{B}_A(\rho^c, u-3R)$ of size $|P|(M+1)$ by deleting all nodes in $\mathrm{B}_A(\rho^c, 3R)$, of which there are at most $|P|$.

- $v_{M+1}^c, \ldots, v_{k-|P|\cdot M}^c$ form a sightseer enumeration of $A \setminus \mathrm{B}_A(\rho^c, u)$ with radius $u$ and memory $|P| \cdot M$. To construct one such we start with a sightseer enumeration of $A$ with radius $u$ and memory $|P| \cdot (M+1)$ and delete all nodes in $\mathrm{B}_A(\rho^c, u)$ from that sequence.

- $v_{k+1}^c, \ldots, v_n^c$, finally, are an arbitrary enumeration of the remaining nodes, *i.e.* those in the set $\mathrm{B}_A(\rho^c, u) \setminus \{v_1^c, \ldots, v_M^c, v_{k-|P|\cdot M+1}^c, \ldots, v_k^c\}$.

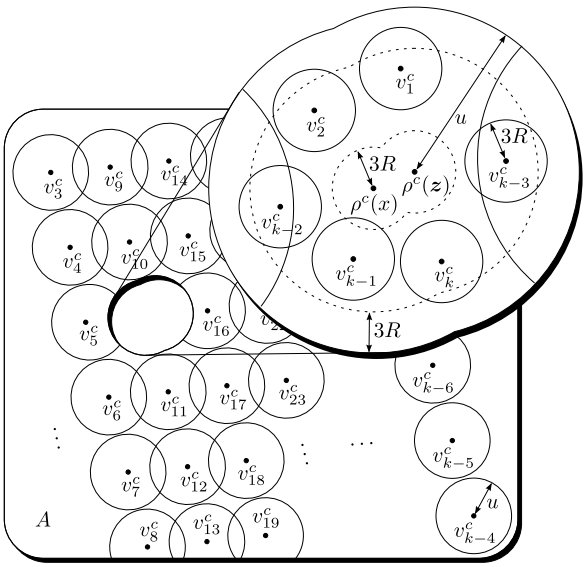



**Figure 2. Iteration Strategy**

The choice of $v_1^c, \ldots, v_k^c$ is illustrated for $P = \{x, \boldsymbol{z}\}$ and $M = 2$ in Figure 2, in which we depict graph nodes as if they were points in the plane and use the euclidean distance to symbolise their distance in the graph.

Consider now the `forall`-like iteration of $f$ with iteration jumps to $v_1^c, \ldots, v_n^c$. It can be shown using Lemma 2 that after the $i$-th step in this computation where $i \leq k$, all pebbles lie in an $R$-neighbourhood of the node set $V = \rho^c(P) \cup \{v_1^c, \ldots, v_M^c\} \cup \{v_{i-|P|\cdot M+1}^c, \ldots, v_i^c\}$. This follows by induction on $i$. By definition the next jump destination $v_{i+1}^c$ is far enough away from all nodes in $V$ for this lemma to be applicable. The requirement that the radius of the sightseer sequence be $u$ is generously dimensioned and could be replaced by $O(R)$ at the expense of some extra boundary conditions.

From this observation it follows that after the jumps to $v_1^c, \ldots, v_k^c$, all nodes lie in $\mathrm{B}_A(\rho^c, u)$. The remaining nodes $v_{k+1}^c, \ldots, v_n^c$ are also chosen from this neighbourhood. Hence, in each of these last steps, we first place pebble $\boldsymbol{z}$ on some node in $\mathrm{B}_A(\rho^c, u)$ and then apply the function $[\![f]\!]$. Since $f$ has local radius $r$, this implies that at the end of the iteration all pebbles will lie in a $u + r \cdot |\mathrm{B}_A(\rho^c, u)|$-neighbourhood around $\rho^c$, which is contained in an $l$-neighbourhood around $\rho^c$.

We can therefore choose $g(q, c)$ so that $[\![g]\!]$ maps $(q, \rho^c)$ to the final configuration of the iteration (note that the local radius of $g$ is $l$). This completes the definition of $g$.

It now remains to show that for each $\rho \in A$ with $[\rho]_l = c$ we can find an enumeration $v_1, \ldots, v_n$ of the graph nodes such that if we define $(q_0, \rho_0), \ldots, (q_n, \rho_n)$ by $(q_0, \rho_0) = (q, \rho)$ and $(q_{i+1}, \rho_{i+1}) = [\![f]\!](q_i, \rho_i[\boldsymbol{z} := v_{i+1}])$ then we have $(q_n, \rho_n) = [\![g]\!](q, \rho)$. Because of $[\rho^c]_l = [\rho]_l$ the $l$-neighbourhoods of $\rho^c$ and $\rho$ are isomorphic. Since the sequence $v_1^c, \ldots, v_M^c, v_{k-|P|\cdot M+1}^c, \ldots, v_n^c$ forms an enumeration of $\mathrm{B}_A(\rho^c, u)$, we can use this isomorphism to obtain an enumeration $v_1, \ldots, v_M, v_{k-|P|\cdot M+1}, \ldots, v_n$ of $\mathrm{B}_A(\rho, u)$. For $v_{k+1}, \ldots, v_{k-|P|\cdot M}$ we choose a sightseer enumeration of $A \setminus \mathrm{B}_A(\rho, u)$ by the same method as above. The required $(q_n, \rho_n) = [\![g]\!](q, \rho)$ can now be shown by replaying the above argument on the final pebble positions in the iteration starting from $(q, \rho^c)$ and making use of the isomorphism between $\mathrm{B}_A(\rho^c, l)$ and $\mathrm{B}_A(\rho, l)$. □

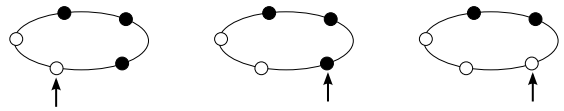
## 4 Undirected Reachability

We now use Theorem 3 to show that PURPLE cannot decide **s**-**t**-reachability in undirected graphs. Given a purported PURPLE program for undirected reachability, we construct an action graph that consists of two disjoint copies of an appropriate Cayley graph and then consider the computation on this graph. Using Theorem 3 we successively eliminate the `forall`-loops in the given program and thus obtain an abstract local program that implements the original PURPLE program on this graph. As part of this process of applying Theorem 3, we obtain an upper bound on the local radius of the simulating abstract local program. We will observe that the action graph constructed at the beginning has diameter greater than twice the radius of the abstract local program. Hence, we can place **s** and **t** on the same connected component with distance larger than twice the local radius. Therefore, the simulating abstract local program will not be able to distinguish whether **s** and **t** are placed in this way or whether they lie on different connected components. From this we then conclude that the original PURPLE program cannot decide reachability.

### 4.1 Lamplighter Graphs

We obtain graphs with appropriate properties by iterating a construction of *lamplighter graphs*. The nodes of such a graph represent the position of a lamplighter and the state (on/off) of the lamps he is responsible for. The lamplighter can either toggle the lamp he is at or move along to the next lamp. The edges of a lamplighter graph join two nodes that are related by one such action.

Pictured below are three nodes of the lamplighter graph with five lamps in a circular arrangement. The balls represent the lamps, the white ones being lit, and the arrow indicates the position of the lamplighter. From left to right, the three nodes are connected by a path in the lamplighter graph.



Lamplighter graphs arise as Cayley graphs, provided the arrangement of lamps is itself a Cayley graph. In the example the lamp arrangement is the Cayley graph of $\mathbb{Z}/5\mathbb{Z}$.

Let $G$ be a group representing the lamp arrangement. Then the set $|G| \to |\mathbb{Z}/2\mathbb{Z}|$ of all functions from the underlying set of $G$ to that of $\mathbb{Z}/2\mathbb{Z}$ becomes a commutative group when addition is defined pointwise by $(f+g)(x) = f(x)+g(x)$ and $0(x) = 0$. We write $\delta_x$ for the function in $|G| \to |\mathbb{Z}/2\mathbb{Z}|$ defined by $\delta_x(x) = 1$ and $\delta_x(y) = 0$ if $y \neq x$. There is a left-action of $G$ on the set $|G| \to |\mathbb{Z}/2\mathbb{Z}|$ with definition $(x \cdot f)(y) = f(x^{-1} \cdot y)$. For example, we have $\delta_x = x \cdot \delta_e$.

Using this notation, we define $L(G)$—the *lamplighter group on* $G$ — to be the group with underlying set $(|G| \to |\mathbb{Z}/2\mathbb{Z}|) \times |G|$ and with group multiplication $(f, x) \cdot (g, y) = (f + x \cdot g,\ x \cdot y)$. An element $(f, x)$ of this group corresponds to a node in a lamplighter graph, where $f$ represents the lighting state of the lamps and $x$ represents the position of the lamplighter. Neutral element and inverses in $L(G)$ are $e = (0,\ e_G)$ and $(f, x)^{-1} = (-x^{-1} \cdot f,\ x^{-1})$. This definition of the lamplighter group is an example of a *wreath product*, *i.e.* $L(G) = (\mathbb{Z}/2\mathbb{Z}) \wr G$.

If $G$ is generated by $S$ then $\{(0, s) \mid s \in S\} \cup \{(\delta_e, e_G)\}$ generates $L(G)$. Moreover, if $G$ has exponent $m$ then $L(G)$ has exponent $2m$.

**Definition 9.** (Lamplighter Graph) Let $C = C(G, S)$ be a Cayley graph whose list of generators $S = g_1, \ldots, g_d$ is closed under inverses. The *lamplighter graph on* $C$ is the Cayley graph of $L(G)$ with respect to the list of generators $(0, g_1), \ldots, (0, g_d), (\delta_e, e_G)$. We write $\Lambda(G, S)$ or just $\Lambda(C)$ for this graph.

The degree and the cardinality of lamplighter graphs obey the following laws.

$$deg(\Lambda(G, S)) = deg(C(G, S)) + 1 \tag{2}$$

$$|\Lambda(G, S)| = |C(G, S)| \cdot 2^{|C(G,S)|} \tag{3}$$

By iterating the lamplighter construction, we obtain graphs in which the range of abstract local program is very small compared to the overall size of the graph. Consider the iterated lamplighter graphs $\Lambda^i(m) := \Lambda(\ldots\Lambda(\mathbb{Z}/m\mathbb{Z})\ldots)$ ($i$ times) on the cyclic group of order $m > 2$ with the standard generating set. Depending on $i$, the graph $\Lambda^i(m)$ can get very large, having at least $\exp_2^i(m)$ nodes, where $\exp_2^0(m) = m$ and $\exp_2^{i+1}(m) = 2^{\exp_2^i(m)}$. In contrast, the exponent of $\Lambda^i(m)$ is $m \cdot 2^i$ only. Its degree is just $i + 2$, by (3). Theorem 1 then tells us that an abstract local program with $q$ states, $p$ pebbles and local radius $r$ has a range of no more than $\left((i+3)^r \cdot q \cdot m \cdot 2^i\right)^{c^p}$ nodes. The point is that this term grows much slower than the size of the graph when we increase $i$. This allows us to apply Theorem 3 repeatedly and still end up with an abstract local program whose radius is less than half the diameter of the graph.

Define now $\Delta^i(m)$ to be the free action graph consisting of two disjoint copies of $\Lambda^i(m)$. The class of graphs on which we will show PURPLE cannot decide reachability then consists of all $\Delta^i(m)$.

In order to use Theorem 3 with the graph $\Delta^i(m)$, we must show that $\Delta^i(m)$ satisfies the two assumptions in that theorem. This can be done by choosing $i$ large enough according to the following two lemmas.

**Lemma 4.** *The graph $\Delta^i(m)$ has a sightseer enumeration with radius $r$ and memory $n$ whenever we have $\exp_2^{i-1}(m) \geq (2r+1) \cdot \max(n, 12)$.*

**Lemma 5.** *For any node $v$ in $\Delta^i(m)$, the $r$-size of $\mathrm{B}_{\Delta^i(m)}(v, l)$ is at least $n$, whenever both $\exp_2^i(m) \geq rn$ and $l \geq r(n+1)$ hold.*

The proofs must be omitted for space reasons. We refer to the technical report [10] for details.

## 4.2 Reachability on Iterated Lamplighter Graphs

We now use the properties of iterated lamplighter graphs to prove our main result that no PURPLE program can decide undirected reachability. In a nutshell, for large enough $i$ and $m$, a given PURPLE program can be implemented on $\Delta^i(m)$ by an abstract local program whose local radius is an exponential tower $\exp_2^r(i+m)$ of constant height. The diameter of $\Delta^i(m)$, however, is at least an exponential tower $\exp_2^{i-2}(m)$ whose height grows linearly with $i$.

Thus by choosing $i$ and $m$ large enough, we can conclude that the PURPLE program cannot decide reachability. The choice of $i$ will depend only on the nesting-depth of the `forall`-loops in the PURPLE program, while $m$ will depend on the length of the program and number of variables in it. It is useful to be able to manipulate the parameters $i$ and $m$ independently

(rather than working only with graphs of the form $\Delta^i(2)$, say), since this allows us to defeat any purported PURPLE program for reachability on a graph whose degree depends only on the maximum nesting depth of the `forall`-loops in the program (Theorem 7 below). Our proof of Corollary 8 depends on this.

**Lemma 6.** *There exist natural numbers $a$ and $b$, such that whenever $f \in L_{\Delta^i(m)}(Q, P, r)$ and the two inequations $m \geq \max(|Q|, |P|)$ and $\exp_2^i(m) \geq \exp_2^a(i+m+r)$ hold then for any $\boldsymbol{z} \in P$ there exists an abstract local program $g \in L_{\Delta^i(m)}\left(Q, P, \exp_2^b(i+m+r)\right)$ that implements* `forall` $\boldsymbol{z}$ `do` $f$ *on $\Delta^i(m)$.*

*Proof Idea.* Write $\mathcal{E}[\vec{x}]$ for the set of *polynomials with exponentiation* that is obtained by closing the polynomials $\mathbb{N}[\vec{x}]$ under exponentiation, see *e.g.* [7].

Using Lemmas 4 and 5 it can be shown that under the given assumptions the graph $\Delta^i(m)$ satisfies the premises of Theorem 3 with $u \in \mathcal{E}[i, m, R, M]$ and $R = 3 \cdot \mathit{range}_{G,S}\left(Q \times (|P|+2)^P, 2 \times P + 1, r\right)$ and $M = \mathit{modulus}_{G,S}\left(Q \times (|P|+2)^P, 2 \times P + 1, r\right)$.

Theorem 3 then provides an abstract local program $g \in L_{G,S}\left(Q, P, u + r \cdot |P| \cdot |\mathrm{B}_{C(G,S)}(e_G, u)|\right)$ that implements `forall` $\boldsymbol{z}$ `do` $f$ on $\Delta^i(m)$. Using Theorem 1, we can bound the local radius of $g$ from above by some $e \in \mathcal{E}[i, m, r]$. But for each $e \in \mathcal{E}[i, m, r]$ there exists a constant $b$ such that $e \leq \exp_2^b(i + m + r)$ holds for all variable assignments, which implies the assertion. □

In the following main theorem, the `forall`-depth of a program is the maximum nesting depth of its `forall`-loops.

**Theorem 7.** *For all $k$ there is a number $d$ such that no* `while`*-free* PURPLE *program with* `forall`*-depth $k$ decides reachability on undirected graphs of degree $d$.*

*Proof idea.* For a PURPLE program $M$ write $|M|$ for its length. Write $Q(M)$ for the set of all functions from the boolean variables in $M$ to booleans. Write $P(M)$ for the set of all graph variables in $M$.

First we show that for each $k$ there exist $i_0$ and $r$ such that for any $i \geq i_0$, any `while`-free PURPLE program $M$ with `forall`-depth $k$ and any $m \geq \max(|Q(M)|, |P(M)|)$ there exists an abstract local program $f$ with local radius $\exp_2^r(i+m+|M|)$ that implements $M$ on $\Delta^i(m)$ with any choice of $\mathbf{s}$ and $\mathbf{t}$. The proof of this goes by induction on $k$ and uses Lemma 6 to deal with `forall`-loops.

Now, given $i_0$ and $r$ as obtained from this observation, we can choose $i \geq i_0$ such that $\exp_2^i(x) > \exp_2^r(i + 2 + 2x)$ holds for all $x$. Let $d := i + 4$, the degree of $\Delta^{i+2}(m)$ for any $m$.

Suppose for a contradiction that some `while`-free program $M$ of `forall`-depth $k$ decides reachability on graphs of degree $d$. Choose $m \geq \max(|M|, |Q(M)|, |P(M)|)$ and observe that we have $\exp_2^i(m) > \exp_2^r(i + 2 + 2m) \geq \exp_2^r(i + 2 + m + |M|)$ by the choice of $i$ and $m$. The above observation now yields a program $f$ with local radius $\exp_2^r(i+2+m+|M|)$ that implements (one run of) $M$ on $\Delta^{i+2}(m)$. But the local radius of $f$ is less than $\exp_2^i(m)$ and it is not hard to show that the diameter of $\Delta^{i+2}(m)$ is at least $2\exp_2^i(m)$. Hence, we can place $\mathbf{s}$ and $\mathbf{t}$ on $\Delta^{i+2}(m)$ so that they have distance $2\exp_2^i(m)$ and $f$ will not be able to distinguish this situation from the one where $\mathbf{s}$ and $\mathbf{t}$ lie on different connected components. But then $f$ cannot decide undirected $\mathbf{s}$-$\mathbf{t}$-reachability and neither can $M$, since $f$ implements $M$. □

**Corollary 8.** *No* PURPLE *program decides undirected $\mathbf{s}$-$\mathbf{t}$-reachability for 3-regular graphs.*

*Proof Idea.* The proof goes by a standard degree reduction as in [2]. Assume there is a PURPLE program $M$ deciding undirected reachability for 3-regular graphs. For any $d$, we can transform $M$ into a program $M_d$ that decides undirected reachability of $d$-regular graphs. Intuitively, $M_d$ simulates the computation of $M$ on the 3-regular graph obtained by replacing each node in the $d$-regular input graph by a cycle of length $d$. While $M_d$ will be larger than $M$, it can be defined to have the same `forall`-depth $k$ as $M$. By choosing the degree $d$ to be the number from Theorem 7, this theorem then yields the desired contradiction. □

For any DTC-formula $\varphi$ we can construct a PURPLE program that recognises precisely the models of $\varphi$. This is done by induction on $\varphi$, the crucial cases being quantification and deterministic transitive closure. For the former, we use a `forall`-loop to explore all possible witnesses; the latter uses nested `forall`-loops to search for the next tuple to continue a deterministic path. For details see Prop. 2 of the full version [10]. We therefore obtain:

**Corollary 9.** *There is no* DTC*-formula for locally ordered graphs (one- or two-way-ordered) that expresses $\mathbf{s}$-$\mathbf{t}$-reachability for undirected graphs of degree 3.*

## 5 Conclusion

By showing that PURPLE-programs cannot decide undirected reachability we have given further evidence that LOGSPACE is strictly more than 'a constant number of pointers'. As an important by-product we

could settle an open question about the expressivity of locally-ordered DTC-logic. Our proof thus constitutes an interesting application of programming-theoretic methods to a purely logical question. In fact, we do not know how to prove Corollary 9 directly without using PURPLE and in particular the 'refinement' of quantification into iteration.

Methodologically, our proof presents a number of innovations, among them the consequent use of group exponents as a graph parameter, the iteration of the lamplighter construction, and sightseer enumerations.

Conceptually, our results can be seen as an analysis of a popular programming methodology: the use of iterators to traverse large data structures. We hope to see more of such analyses in the future; our work (among others) shows that it is possible to answer non-trivial and relevant questions about the intrinsic power of programming concepts without hitting hitherto inaccessible complexity-theoretic questions.

## Acknowledgement

This work was supported by the DFG project *programming language aspects of sublinear space complexity classes*.

## References

[1] S. Cook and P. McKenzie. Problems complete for deterministic logarithmic space. *J. Algorithms*, 8(3), 385–394, 1987.

[2] S. Cook and C. Rackoff. Space lower bounds for maze threadability on restricted machines. *SIAM J. Comput.*, 9(3), 636–652, 1980.

[3] J. Edmonds, C.K. Poon and D. Achlioptas. Tight lower bounds for st-connectivity on the NNJAG model. *SIAM J. Comput.*, 28(6), 2257–2284, 1999.

[4] H. Ebbinghaus and J. Flum. *Finite Model Theory*. Springer-Verlag, 1995.

[5] K. Etessami and N. Immerman. Reachability and the power of local ordering. *Theor. Comput. Sci.*, 148(2), 261–279, 1995.

[6] M. Hofmann and U. Schöpp. Pure pointer programs with iteration. In *CSL08*, LNCS 5213, 79–93, 2008.

[7] H. Levitz. An ordinal bound for the set of polynomial functions with exponentiation. *Algebra Universalis*, 8, 233–243, 1978.

[8] C.K. Poon. Space bounds for graph connectivity problems on node-named JAGs and node-ordered JAGs. In *FOCS93*, 218–227, 1993.

[9] U. Schöpp. A formalised lower bound on undirected graph reachability. In *LPAR08*, LNAI 5330, 621-635, 2008.

[10] U. Schöpp and M. Hofmann. Pointer programs and undirected reachability. ECCC-Report TR08-090. `http://www.tcs.ifi.lmu.de/~schoepp/Docs/reach.pdf`.