# Cyclone: A Heterogeneous Tool for Verifying Infinite Descent

Liron Cohen[1][0000−0002−6608−3000], Reuben N. S. Rowe[2][0000−0002−4271−9078], and Matan Shaked[1]

[1] Ben-Gurion University of the Neg'ev, Beer-Sheva, Israel
cliron@bgu.ac.il, shakedma@post.bgu.ac.il
[2] Royal Holloway, University of London, UK
reuben.rowe@rhul.ac.uk

**Abstract.** The Infinite Descent property underpins key verification techniques, such as size-change program termination and cyclic proofs. Deciding whether the Infinite Descent property holds of a given program or cyclic deduction is PSPACE-complete, with several exponential time algorithms in the literature. In this paper, we consider algorithms with better time complexity but which are (necessarily) *incomplete*. Concretely, we formulate and evaluate a number of alternative algorithms for semi-deciding Infinite Descent. Our aim is to improve average runtime performance by utilising more efficient algorithms for specific subclasses of input. We present Cyclone, a tool integrating these algorithms with an existing (complete) decision procedure. We evaluate Cyclone on a large suite of examples harvested from the Cyclist theorem prover, finding that the incomplete algorithms achieve extremely high coverage and afford substantial runtime improvement in practice. We thus believe that the Cyclone tool will foster broader adoption of techniques based on Infinite Descent and expand their practical applications.

**Keywords:** Infinite descent · Cyclic proof · Program termination

## 1 Introduction

Infinite Descent is an $\omega$-regular liveness property that has important practical applications in the verification of software. For instance, it underpins the size-change framework for checking program termination [16], in which a program's call-graph is used to produce an abstraction recording when the values manipulated by the program (e.g. numbers) decrease as they are passed between function calls. Infinite Descent then states that along all infinite paths through this call-graph, we can trace a value that (strictly) decreases infinitely often. If the call-graph satisfies this property then we know the program must terminate, since the order used to interpret the decrease of values is well-founded. This technique is used, for example, in the termination checker for the Agda proof assistant [1].

Infinite Descent also plays a crucial role in cyclic proof-theoretical techniques for reasoning about inductive (and coinductive) properties [7,10,11,12,13,21,23].

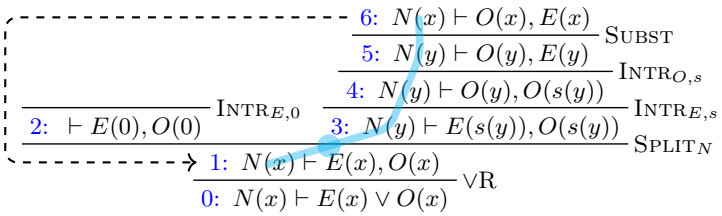



Fig. 1: A cyclic proof that every natural number is either even or odd. The blue trace witnesses the Infinite Descent, with progress marked by the blue circle.

Instead of using inference rules in which a (co)inductive invariant must be explicitly provided, the invariants can be 'discovered' by repeatedly decomposing a goal into subgoals that are either provable or reducible back to the original goal, forming a cycle in the proof. The Infinite Descent condition is then used to justify the soundness of such a cyclic proof graph. In proof-theoretic settings, the notion of 'decrease' is usually called 'progress', and commonly corresponds to particular logic-specific steps that 'unfold' instances of (co)inductive definitions.

For example, Fig. 1 shows a cyclic proof of the fact that every natural number is either even or odd, in a system for first-order logic with inductive predicates. The natural numbers predicate $N$ is (inductively) defined via the rules $N(0)$ and $N(s(x)) \Leftarrow N(x)$ (where $s$ stands for the successor). The predicates $E$ and $O$ denote even and odd numbers, respectively, and are mutually defined via the rules $E(0)$, $O(s(x)) \Leftarrow E(x)$ and $E(s(x)) \Leftarrow O(x)$. The $\text{Split}_N$ rule performs a case split on the predicate instance $N(x)$, guided by the defining rules for $N$, which substitutes 0 and $s(y)$ for $x$ in the left- and right-hand premises, respectively. In the latter, $y$ is a fresh variable (denoting the predecessor of $x$) for which we know from the definition of $N$ that $N(y)$ must hold. The other steps of the proof unfold the $E$ and $O$ predicate instances, and perform substitutions. A cyclic proof has the structure of a tree with 'backlinks', i.e. a tree in which some of the leaves (called *buds*) have an edge to another node (called its *companion*). In Fig. 1, node 6 is a bud whose companion is (the syntactically equal) node 1. When all buds link to an ancestor node (as in this case), the proof is said to be (in) *cycle normal (form)*, however this need not be the case in general. The proof satisfies the Infinite Descent property since along the infinite path traversing the cycle, we can trace a value (the instances of the $N$ predicate) that *progresses* (i.e. is unfolded by the $\text{Split}_N$ rule) infinitely often.

This kind of cyclic reasoning has been employed widely to create both theoretical frameworks for program verification and inductive theorem proving (e.g., [4,5,20,24,25,27,18]), as well the Cypress program synthesis tool [14] and the automatic theorem provers Cyclist [6], Songbird [8], Inductor [22] and CycleQ [15]. Since the problem of deciding the Infinite Descent property is PSPACE-complete [16,19], in practice these tools all implement one of a number of known algorithms that have worst-case exponential runtimes [9]. Although this worst-

case performance is not often encountered 'in the wild', these tools still rely on deciding large numbers of problem instances. Therefore any speed up of the Infinite Descent check has the potential to provide significant benefits in practice.

Our goal is to advance the state-of-the-art for deciding Infinite Descent and thus push forward the practical use of automated cyclic reasoning and termination checking. Our approach is to identify more efficient algorithms that only *semi-*(co)decide Infinite Descent, along with (efficiently decidable) characterisations of the subclass of instances on which they may return a definite answer. The idea is that, since they do not need to uniformly treat all problem instances, these algorithms may utilise particular structure in the input to return an answer more quickly than the full decision procedures. We then heuristically combine these semi-decision procedures into a heterogeneous pipeline, defaulting to a uniform decision procedure in case none of them can return an answer.

In this paper we describe three such algorithms. The first decides an existing criterion, proposed by Brotherston [3], called the Trace Manifold condition. The other two decide novel criteria that we have formulated and call "Flat Cycles" and "Descending Unicycles", respectively. We analyse the coverage of these algorithms by harvesting a large database of problem instances generated by the test suites of the Cyclist theorem prover, which comprise inductive entailments of First-Order Logic and Separation Logic. Guided by the analysis of these algorithms, we combine and implement them within a new tool dubbed Cyclone, which we integrate into Cyclist. We present an evaluation of our tool's performance on our harvested database, comparing it to the performance of the existing decision procedures. We found that Cyclone demonstrates significant runtime improvements, sometimes of several orders of magnitude.

*Paper Outline.* Sec. 2 formally defines the Infinite Descent problem, in a general form, and sumarises the existing decision procedures. Sec. 3 describes the database of problem instances that we harvested for evaluating our new algorithms. Sec. 4 then describes our novel semi-decision procedures. In Sec. 5 we present the implementation of our tool, Cyclone, and compare its performance to the existing methods implemented in Cyclist. Finally, Sec. 6 concludes.

## 2 Infinite Descent for Sloped Graphs

We begin by formally defining the Infinite Descent property in the abstract setting of *sloped graphs*, following [9], which captures the essence of the Infinite Descent problem in an application-independent way. That is, the formulation of Infinite Descent in size-change termination, or some given cyclic logical proof system, are special instances of the abstract definition we give here.

Infinite Descent tracks the ordering relationship between (abstract) values along paths in a graph. The following definition of *sloped graphs* thus augments the standard notion of a directed graph by associating with each node, or vertex, a collection of abstract *positions*, and with each edge a relation assigning (flat or downward) *slopes* between the positions associated with its end-points.

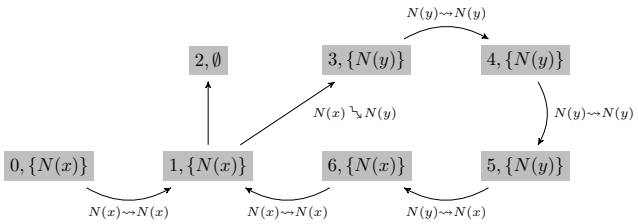



Fig. 2: Sloped graph from Example 1

**Definition 1 (Sloped graphs).** *We assume a set* $\mathsf{Pos}$ *of* positions *and a set* $\mathsf{S} = \{\rightsquigarrow, \searrow\}$ *of* slopes*, whose elements are called* flat *and* downward*, respectively.*
- *A* sloped relation $R \subseteq \mathsf{Pos} \times \mathsf{Pos} \times \mathsf{S}$ *is a partial function from pairs of positions to slopes.*
- *A* sloped graph $SG$ *is a tuple* $(V, E, Ps, (R_{(v,v')})_{(v,v')\in E})$ *such that:*
  *(1)* $(V, E)$ *is a directed graph with nodes* $V$ *and edges* $E$*;*
  *(2)* $Ps : V \to \wp(\mathsf{Pos})$ *is a function assigning a set of positions to every node;*
  *(3)* $(R_{(v,v')})_{(v,v')\in E}$ *is a family of sloped relations* $R_{v,v'} \subseteq Ps(v) \times Ps(v') \times \mathsf{S}$ *indexed by edges.*

*We call the quantity* $\max\{|Ps(v)| \mid v \in V\}$ *the* (vertex) width *of* $SG$*.*

*Example 1.* The cyclic proof shown in Fig. 1 can be abstracted by the sloped graph $SG = (V, E, Ps, (R_{(v,v')})_{(v,v')\in E})$ defined as follows, and depicted in Fig. 2, where we use inductive predicate instances as positions.
- $V = \{0, 1, 2, 3, 4, 5\}$.
- $E = \{(0, 1), (1, 2), (1, 3)(3, 4), (4, 5), (5, 6), (6, 1)\}$.
- $Ps(0) = Ps(1) = Ps(6) = \{N(x)\}$, $Ps(3) = Ps(4) = Ps(5) = \{N(y)\}$, $Ps(2) = \emptyset$.
- $R_{0,1} = R_{6,1} = \{(N(x), N(x), \rightsquigarrow)\}$, $R_{1,3} = \{(N(x), N(y), \searrow)\}$, $R_{3,4} = R_{4,5} = \{(N(y), N(y), \rightsquigarrow)\}$, and $R_{5,6} = \{(N(y), N(x), \rightsquigarrow)\}$.

Since a sloped graph is a form of directed graph, we also adopt the graph-theoretic notions of (finite and infinite) *paths* through the graph, which we denote by $(v_i)_{i\in\alpha}$ where $\alpha \leq \omega$ is the (ordinal that is the) length of the path.

A trace along a path in a sloped graph selects a position from each node in the path, making sure that each position is related to the next by the sloped relations associated with the edges that are traversed.

**Definition 2 (Traces).** *A* trace *along an infinite path* $(v_i)_{i\in\alpha}$ *in a sloped graph* $SG$ *is an infinite sequence of positions* $\tau = (p_i)_{i\in\alpha}$ *such that, for every* $i < \alpha$*, both* $p_i \in Ps(v_i)$ *and* $R_{v_i,v_{i+1}}(p_i, p_{i+1}, s)$ *for some (necessarily unique) slope* $s$*. When* $s = \searrow$ *we call* $i$ a progressing point *in the trace. We may also write* $\tau(v_i)$ *to denote the* $i^{th}$ *position,* $p_i$*. A trace is* decreasing *if it has infinitely many progressing points. An infinite path is* descending *if it has a tail along which there is a decreasing trace.*

**Definition 3 (Infinite Descent).** *A sloped graph is said to satisfy Infinite Descent if all of its infinite paths are descending.*

| Algorithm | Time Complexity Upper Bound |
|---|---|
| VLA | $\mathcal{O}(n^5 \cdot w^2 \cdot 2^{2nw\log(2nw)})$ |
| SLA | $\mathcal{O}(n^2 \cdot w \cdot Min(n^4, 3^{2w^2}) \cdot 2^{2w\log(2w)})$ |
| FWK | $\mathcal{O}(n{\cdot}w^4{\cdot}3^{3w^2} + n^3{\cdot}w^4{\cdot}3^{2w^2})$ |
| OR | $\mathcal{O}(n^3 \cdot w^4 \cdot 3^{2w^2})$ |

Table 1: Time complexity bounds for Infinite Descent decision procedures

The problem of deciding whether a given sloped graph satisfies Infinite Descent is PSPACE-complete [16,19]. There are two basic approaches for deciding the Infinite Descent property described in the literature.

**Automata-theoretic:** One approach is to encode the problem as an inclusion between $\omega$-automata: a 'path' automaton that recognises words corresponding to all infinite paths in the sloped graph and a 'trace' automaton that recognises words corresponding to all potential descending traces. A sloped graph satisfies Infinite Descent if and only if the former automaton is included in the latter. The problem can be encoded using $\omega$-words over either the vertices or the sloped relations of the sloped graph. We call these encodings the *Vertex-Language Automata* (VLA) and the *Slope-Language Automata* (SLA) encodings, respectively.

**Ramsey-theoretic:** An alternative approach is to compute for each pair of nodes the collection of sloped relations consisting of the compositions of the sloped relations along (finite) paths between the two nodes, with slopes combined according to the ordering $\rightsquigarrow < \searrow$. Once this 'composition closure' is computed, checking Infinite Descent amounts to verifying the presence of certain downward slopes in the relations representing loops in the sloped graph. This encoding is an instance of the algebraic path problem and can be solved using the Floyd-Warshall-Kleene (FWK) algorithm [17]. By taking advantage of the specific symmetric nature of the Infinite Descent setting, a so-called *order-reduced* (OR) optimisation of this procedure is possible [9].

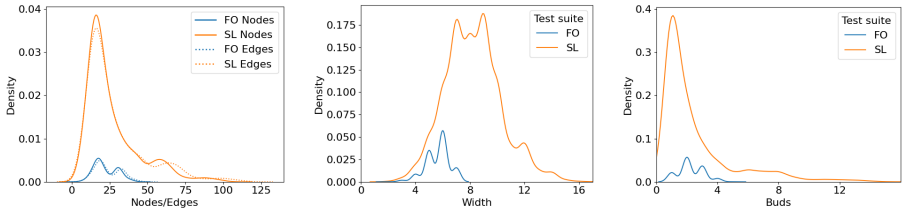
All these algorithms exhibit exponential runtime in the worst case. However the complexity profile of each method depends, in varying proportions, on two parameters of the sloped graph: the number of nodes $n$, and the vertex width $w$. The worst-case complexity bounds are summarised in Table 1. A comprehensive account of these methods and their comparative performance can be found in [9].

## 3 A Database of Sloped Graphs

In order to support our goal of identifying useful classes of sloped graphs for which Infinite Descent can be (semi-)decided (more) efficiently, we generated a large database of sloped graphs using the Cyclist automated theorem prover [6], which we believe to be representative of problem instances arising in real-world applications. Cyclist implements a generic engine for cyclic proof search, supporting arbitrary (cyclic) logical systems by exposing an API to this engine. It currently features a number of logics with inductively defined predicates, includ-

| | Satisfies Infinite Descent | Does not satisfy Infinite Descent | (all) |
|---|---|---|---|
| FOL | 260 | 6437 | 6697 |
| SL | 42692 | 27694 | 70386 |
| (all) | 42952 | 34131 | 77083 |

Table 2: Aggregated numbers of sloped graphs in our database




(a) Nodes, Edges (b) Width (c) Buds

Fig. 3: Distribution of various sloped graph metrics

ing first-order logic (FOL) and Separation Logic (SL), which come with test suites of representative (valid and invalid) logical entailments. During a search for a cyclic proof of a given entailment, Cyclist runs the Infinite Descent check for each intermediate (partial) candidate proof it encounters. We modified Cyclist to output a JSON representation of the corresponding sloped graphs. (Note that Cyclist also employs a preprocessing procedure for minimizing the proof graphs, which we discuss in Sec. 5.2.)

In total, we collected over 77,000 sloped graphs. Table 2 shows the numbers of sloped graphs aggregated by the logic they were generated from and whether or not they satisfy Infinite Descent. We also collated statistics pertaining to various metrics of sloped graphs, namely the number of nodes, edges, buds, and vertex width. Again aggregated by the logic, Fig. 3 shows the density functions of these metrics. Across the database, per graph, the number of nodes is at most 107 and the number of edges is at most 120, with graphs most frequently having around 20 nodes/edges. Additionally, graphs have a vertex width of at most 17, with the most frequent quantity being around 8. We see a maximum of 16 buds in any graph, with most of the graphs having around 2 buds. We observe that the vertex width is not particularly correlated with the number of edges/nodes and, in general, the width is low compared to the number of nodes. In contrast, in both test suites, we can see a high correlation between the number of nodes and edges with the two metrics being almost identical. We also see that, in general, the more nodes/edges in a graph, the more buds. However, the number of buds is generally *much* lower than the number of nodes. This is not surprising given that the graphs are (proof) trees with (few) backlinks, rather than generally highly connected graphs.

## 4 Effective Semi-algorithms for Infinite Descent

We now present the novel (incomplete) semi-decision procedures that we developed. Concretely we present the following heuristics, analysing their runtime complexity and coverage (the number of graphs in our database for which it returns a definitive answer, i.e. "yes" or "no"):

**Trace Manifold (TM):** a criterion from [3], for which we provide a novel algorithm and implementation. This heuristic returns "yes"/"don't know".

**Flat Cycles (FC):** a novel criterion based on the notion of *flat edges* in a sloped graph. This heuristic returns "no"/"don't know".

**Descending Unicycles (DU):** a novel criterion based on *non-overlapping cycles* in a sloped graph. This heuristic returns "yes"/"no"/"don't know".

We began by implementing the Trace Manifold criterion, but we found the coverage of this method on our database to be very low. This prompted us to formulate the other two criteria, which have better runtime complexity and verify Infinite Descent on a much larger class of sloped graphs.

### 4.1 The Trace Manifold Criterion

The *Trace Manifold* criterion (TM) is a property of basic cycles in a sloped graph, $SG$, that is a cycle normal tree with backlinks. In such graphs, each basic cycle can be identified with a bud $B$, being the unique path in the graph leading to $B$ from its companion, denoted $\mathcal{R}(B)$. A *structural connectivity* relation, $\leq_{SG}$, over the buds can also be defined by relating two buds $B_1$ and $B_2$ precisely when $\mathcal{R}(B_1)$ lies along the basic cycle associated with $B_2$.

A trace manifold comprises a set of traces for basic cycles.

**Definition 4 (Trace Manifold).** *A set of (finite) traces for paths in a sloped graph $SG$ is called a* trace manifold *when it has the following form*

$$\{\tau_{S,B} \mid S \text{ a strongly connected subgraph of } SG, B \in S \text{ is a bud}\}$$

*and satisfies the following.*

*(1) Each $\tau_{S,B}$ is a trace along the basic cycle of $B$.*

*(2) For all $\tau_{S,B_1}$ and $\tau_{S,B_2}$, if $B_1 \leq_{SG} B_2$ then $\tau_{S,B_1}(\mathcal{R}(B_1)) = \tau_{S,B_2}(\mathcal{R}(B_1))$.*

*(3) For every strongly connected subgraph, $S$, of $SG$ there is a bud $B \in S$ such that $\tau_{S,B}$ has at least one progressing point.*

The properties of a trace manifold entail that its constituent traces can be combined to yield descending traces for each infinite path.

**Proposition 1 (Trace Manifold Criterion [3, Prop. 7.2.3]).** *If a (cycle normal) sloped graph has a trace manifold, then it satisfies Infinite Descent.*

To our knowledge, the algorithm we now describe is the first concrete algorithm proposed to decide the trace manifold criterion. Ultimately, however, given its low coverage and potentially exponential runtime, we decided not to include it in CYCLONE's final pipeline. For this, and for space reasons, we describe the algorithm only informally; details are given in the appendix.

**Algorithm.** Firstly, check if the sloped graph, $SG$, is in cycle normal form, returning "don't know" if not. Otherwise, create the *trace manifold graph* of $SG$: a directed graph whose nodes are all pairs consisting of a bud, $B$, and a trace along its associated basic cycle, taking edges $((B,\tau),(B',\tau'))$ iff $B \leq_{SG} B'$ and $\tau(\mathcal{R}(B)) = \tau'(\mathcal{R}(B))$. Then, for each strongly connected subgraph, $S$, (wlog) choose a bud, $B \in S$ and check that, a DFS in the undirected trace manifold starting from some node $(B,\tau)$ has the following properties: (1) a node $(B',\tau')$ is visited for every bud $B' \in S$; (2) for every pair of buds $B_1, B_2 \in S$ such that $B_1 \leq_{SG} B_2$, an edge involving $B_1$ and $B_2$ is traversed; and (3) a node is visited containing a trace with a progression point. This check succeeds iff a trace manifold exists. So, return "yes" in this case, otherwise return "don't know".

**Complexity and Practical Runtime Evaluation.** The algorithm described above is exponential in both the number of buds and the number of nodes of the sloped graph, due to the size of the trace manifold graph: quantification over all strongly-connected subgraphs leads to the exponential dependency on the number of buds, and the quantification over all traces following basic cycles leads to the exponential dependency in the number of (sloped graph) nodes. Overall, the algorithm has a worst-case $\mathcal{O}(2^{\beta} \cdot (|V| \cdot w^{|V|} + \beta^2 \cdot w^{|V|}))$ runtime complexity, where $\beta$ is the number of buds of the sloped graph, and $w$ is the vertex width. Note that, given the set of buds (information that is provided by Cyclist), checking whether a graph is in cycle normal form takes only polynomial time. Despite the high complexity, our evaluation revealed that the algorithm performs well in practice, since the number of traces is usually fairly small.

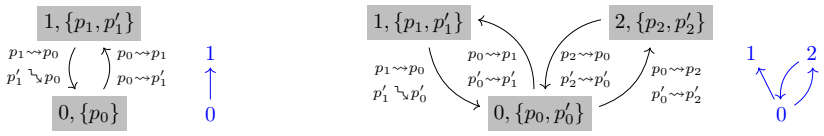
Evaluating the implementation of our algorithm on our database of sloped graphs we discovered that, on average, its runtime performance is significantly better than the state-of-the-art *complete* method (OR). We observed that the more edges in the input sloped graph, the faster our implementation of TM compared to that of OR. Our implementation was at most 29% slower than OR, and up to 1,970% faster. However, despite its favourable runtime performance it only covered 31.2% of the sloped graphs in our database that satisfy Infinite Descent, and thus only 17.38% of the database overall. Moreover, although 48.4% of graphs in the database are in cycle normal form, TM returns an answer on only 35.9% of these. Interestingly, though, almost all cycle normal graphs in our database satisfying Infinite Descent also satisfy TM.

### 4.2 Flat Cycles

This section presents a novel linear runtime method for checking if a sloped graph does *not* satisfy Infinite Descent. For this, we first define the notion of the *flat projection graph*. Intuitively, the flat projection graph of a sloped graph is a directed graph with edges only between two nodes that have no downward position slope between them.

**Definition 5 (Flat projection graph).** *The* Flat Projection Graph *of a sloped graph* $SG = (V, E, Ps, (R_{(v,v')})_{(v,v')\in E})$ *is the graph* $SG^{\rightsquigarrow} = (V, E^{\rightsquigarrow})$*, where*

$$E^{\rightsquigarrow} = \{(u,v) \mid (u,v) \in E \ \wedge \ \forall p \in Ps(u)\ \forall q \in Ps(v).\ (p, q, \searrow) \notin R_{u,v}\}$$




(a) Does not satisfy FC (b) Satisfies FC

Fig. 4: Sloped graphs and their Flat Projection graphs (in blue)

**Proposition 2 (Flat Cycles criterion (FC)).** *Let $SG$ be a sloped graph. If $SG^{\rightsquigarrow}$ has a cycle then $SG$ does not satisfy Infinite Descent.*

*Example 2.* Fig. 4 shows two examples of sloped graphs: one that does not satisfy the flat cycles criterion and one that does. Fig. 4a presents a sloped graph with two nodes: node 0 with one position, $p_0$, and node 1 with two positions, $p_1, p'_1$. Since there is a downward slope in $R_{1,0}$, $(1,0) \notin E^{\rightsquigarrow}$ and therefore the flat projection graph of this sloped graph (presented in blue) does not contain a cycle. This, in turn, entails that FC does not yield a decision on this graph. On the other hand, Fig. 4b presents a sloped graph with three nodes: node 0 having one position and nodes 1, 2, having two positions each. Again, $(1,0) \notin E^{\rightsquigarrow}$ because there is a downward slope in $R_{1,0}$. However, because there is no downward slope in either $R_{0,2}$ or $R_{2,0}$, we get that $(0,2) \in E^{\rightsquigarrow}$ and $(2,0) \in E^{\rightsquigarrow}$, thus, there is a cycle in the flat projection graph (presented in blue), which means that the sloped graph satisfies the flat cycles criterion. Indeed, the sloped graph portrayed in Fig. 4b does not satisfy Infinite Descent, as Prop. 2 entails.

While FC may seem like a strong condition, requiring an entire cycle in the graph with no downward slopes on any of its edges, in practice it is very frequent and covers 80.77% of the graphs in the database that do not satisfy Infinite Descent, and thus 35.76% of the entire database.

**Algorithm.** Algo. 1 checks if a sloped graph satisfies Infinite Descent using the FC criterion. It generates the flat projection graph and checks it for cycles using a depth-first search (DFS). If there is a cycle, we know from Prop. 2 that the sloped graph does not satisfy Infinite Descent and so return "no". Otherwise, the algorithm returns "don't know".

**Complexity and Practical Runtime Evaluation.** The algorithm goes over all edges in the sloped graph ($|E|$ iterations) and for each one it checks, in time quadratic in the vertex width of the input graph, whether its associated sloped relation has no downward slope. If the relation has no downward slope we add the edge to the flat edges set, which can be done in $\mathcal{O}(1)$ if we store this set as an adjacency linked list for every node in the graph. Then we perform a DFS on the flat projection graph which has runtime complexity of $\mathcal{O}(|V| + |E^{\rightsquigarrow}|)$. Since $E^{\rightsquigarrow} \subseteq E$, in the worst case $|E^{\rightsquigarrow}| = \mathcal{O}(|E|)$, which makes the worst-case runtime $\mathcal{O}(|E| \cdot w + |V| + |E|) = \mathcal{O}(|V| + |E| \cdot (w+1))$.

**Algorithm 1** Infinite Descent by the Flat Cycles Criterion

**Input:** Sloped Graph $SG = (V, E, Ps, (R_{(v,v')})_{(v,v')\in E})$, vertex width $= w$
**Output:** "no" if $SG$ has a flat cycle and "don't know" otherwise

$E^{\rightsquigarrow} := \emptyset$
**for all** $(u, v) \in E$ **do** ▷ $|E|$ iterations
  **if** $(p, q, \searrow) \notin R_{u,v}$ for all $p \in Ps(u), q \in Ps(v)$ **then** ▷ $\mathcal{O}(w^2)$
    $E^{\rightsquigarrow} \leftarrow E^{\rightsquigarrow} \cup \{(u, v)\}$ ▷ $\mathcal{O}(1)$
**if** $\mathsf{DFS}(V, E^{\rightsquigarrow})$ detects a cycle **then return** "no" ▷ $\mathcal{O}(|V| + |E^{\rightsquigarrow}|)$
**else return** "don't know"

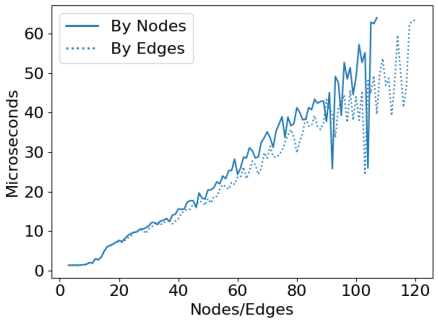



Fig. 5: Flat Cycles Runtime by Nodes and Edges

Fig. 5 shows the runtime of checking the criterion as a function of either the number of nodes or the number of edges in the graph. The values in the figure are an average of the runtime among all sloped graphs with each number of nodes/edges. We can see a linear growth in the runtime in both graphs, as expected from the runtime complexity analysis.

### 4.3 Descending Unicycles

Having defined in the previous section a linear-time algorithm that covers a significant amount of the graphs in the database that *do not* satisfy Infinite Descent, in this section, we present a novel, polynomial-time criterion that covers a significant amount of the graphs that also *do* satisfy Infinite Descent.

We first identify a class of sloped graphs we call *unicycles graphs*. We say that a path from cycle $c$ to cycle $c'$ is any path $v_0, ..., v_n$ such that $v_0 \in c$ and $v_n \in c'$.

**Definition 6 (Unicycles graph).** *A directed graph $G = (V, E)$ is a* unicycles graph *if for every two distinct basic cycles $c, c'$ in $G$, if there is a path from $c$ to $c'$, then there is no path from $c'$ to $c$.*

Note that a unicycles graph necessarily does not contain any overlapping cycles, i.e., two cycles with some shared node(s). Thus, e.g., the graph in Fig. 4b is not a unicycles graph since there is a path in both directions through the node


(a) Satisfies DU (b) Does not satisfy DU

Fig. 6: Unicycles Graphs

0. However, the graph in Fig. 4a is a unicycles graph because it has just one cycle. The graphs in Fig. 6 are unicycles graphs because, in both cases, although there is a path from $c_1 = 0, 1$ to $c_2 = 2, 3$, there is no path from $c_2$ to $c_1$.

The key insight is that if a sloped graph is a unicycles graph, then the infinite paths in the sloped graph are of the form $\pi = v_0, ..., v_m, (u_0, ..., u_k)^\omega$, with $u_0, ..., u_k$ a basic cycle in the graph. This means that checking Infinite Descent on unicycles graphs amounts to checking whether, for every basic cycle $c$ in the graph, the path $c^\omega$ has a progressing trace. We next formalize this requirement.

**Definition 7 (Simply descending graph).** *we define the following, given a sloped graph* $SG = (V, E, Ps, (R_{(v,v')})_{(v,v')\in E})$.

*(1) The* positions graph induced by *a path* $\pi = v_1, \ldots, v_m$ *in* $SG$*, denoted* $SG^{pos}_\pi$*, is a directed graph* $(V_\pi, E_\pi)$ *with a distinguished subset of* progressing *edges* $Prog_\pi \subseteq E_\pi$*, defined by:*
- $V_\pi = \{(v_i, p) \mid 1 \leq i < m \text{ and } p \in Ps(v_i)\}$
- $E_\pi = \{((v_i, p), (v_{i+1}, q)) \mid 1 \leq i \leq m \text{ and } \exists s.(p, q, s) \in R_{v_i,v_{i+1}}\}$
- $Prog_\pi = \{((v_i, p), (v_{i+1}, q)) \mid 1 \leq i < m \text{ and } (p, q, \searrow) \in R_{v_i,v_{i+1}}\}$

*(2) A basic cycle* $c$ *in* $SG$ *is said to be* descending *if* $SG^{pos}_c$ *has a basic cycle with at least one progressing edge (i.e., an edge in* $Prog_c$*).*

*(3) We say that* $SG$ *is* simply descending *if every basic cycle in* $SG$ *is descending.*

*Example 3.* The graph $SG$ in Fig. 4a is simply descending because the basic cycle $(0, p_0), (1, p'_1)$ of $SG^{pos}_{(0,1)}$ has a progressing edge. The graph $SG$ in Fig. 4b is not simply descending, because its positions graph $SG^{pos}_{(0,2)}$ has no progressing edge, and thus its basic cycle $0, 2$ is not descending. Fig. 6 illustrates two sloped graphs with the same underlying directed graph. The graph in Fig. 6a is simply descending because both $c_1 = (0, 1)$ and $c_2 = (2, 3)$ are descending cycles. However, the graph in Fig. 6b is not simply descending because $c_2$ is not descending.

For unicycles graphs, the simply descending criterion is both sound *and* complete for Infinite Descent.

**Proposition 3 (Descending Unicycles criterion (DU)).** *If* $SG$ *is a unicycles sloped graph, then* $SG$ *satisfies Infinite Descent iff it is simply descending.*

Like the FC criterion, the DU criterion seems to be a strong condition in requiring that cycles in the sloped graph do not overlap. However, again, in practice, this requirement is satisfied in 90.69% of all graphs in our database,

**Algorithm 2** Infinite Descent by the Descending Unicycles Criterion

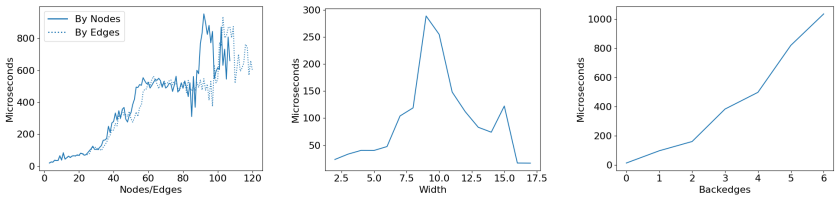
**Input:** Sloped Graph $SG = (V, E, Ps, (R_{(v,v')})_{(v,v')\in E})$, vertex width $= w$
**Output:** "don't know" if $SG$ is not a unicycles graph, "yes" if Descending Unicycles holds for $SG$ and "no" otherwise

1: $SCCs$, $backedgesLowLinks \leftarrow \mathsf{Tarjan}(V, E)$ ▷ $\mathcal{O}(|V| + |E|)$
2: **if** $\mathsf{hasDuplicates}(backedgesLowLinks)$ **then** ▷ $\mathcal{O}(|V|)$
3: **return** "don't know"
4: **for all** $SCC \in SCCs$ **do** ▷ $\mathcal{O}(|V|)$ iterations
5: **if not** $\mathsf{isDescendingCycle}(SCC, SG)$ **then** ▷ $\mathcal{O}(w^2 \cdot |SCC|)$
6: **return** "no"
7: **return** "yes"

which makes for almost complete coverage of the database. However, unlike the FC criterion, it can return both a definite "yes" and a definite "no" answer.

**Algorithm.** Algo. 2 checks if a sloped graph that is a tree with backlinks satisfies Infinite Descent using the DU criterion. First, it calculates the strongly connected components (SCCs) of the graph, together with the low link of each bud's destination using Tarjan's algorithm [26]. Note that a graph has overlapping cycles if and only if there are two buds whose destination nodes have equal low links. That is because two cycles overlap if and only if they form a strongly connected set and because the buds' destination nodes have the same low link if and only if they are in the same SCC. Thus, if there are duplicates in this $backedgesLowLinks$ list, then the graph is not a unicycles graph and we return a "don't know". Otherwise, the graph *is* a unicycles graph, which means that every strongly connected component is a basic cycle. Then, we go over all SCCs and check if they are descending cycles of $SG$. If and only if so, by Prop. 3 we get that SG satisfies Infinite Descent. Checking if a cycle $c$ is a descending cycle amounts to running Tarjan's algorithm on $SG_c^{pos}$ while also checking with each edge if it is progressing. We find a strongly connected component with a progressing edge in $SG_c^{pos}$ if and only if $c$ is a descending cycle in $SG$. That is because the progressing edge must be a part of a basic cycle and because every basic cycle is a part of a SCC.

**Complexity and Practical Runtime Evaluation.** Line 1 uses Tarjan's algorithm, which has a runtime complexity of $\mathcal{O}(|V| + |E|)$. It also returns some of the low links that are generated by Tarjan's algorithm, of which there are $\mathcal{O}(|V|)$. Line 2 looks for duplicates in the returned low links list, which is done in $O(|V|)$ (on average) by generating a hash set from the list and comparing its size to the low links list's size. Finally, line 4 iterates over all SCCs and for each one checks if it is descending using Tarjan's algorithm on $SG_{SCC}^{pos}$. $SG$ is a unicycles graph, and so in any SCC of $SG$ the number of nodes is equal to the number of edges. Further, since each edge in a strongly connected component $SCC$ has $\mathcal{O}(w^2)$ corresponding edges in $SG_{SCC}^{pos}$, Tarjan's algorithm on $SG_{SCC}^{pos}$ has runtime complexity of $\mathcal{O}(|SCC| + w^2 \cdot |SCC|)$. Since the SCCs




(a) By Nodes and Edges (b) By Width (c) By Buds

Fig. 7: Descending Unicycles Runtime

of a graph are a partition of its nodes, the runtime complexity of the loop in Line 4 is $\mathcal{O}(|V| + w^2 \cdot |V|)$. Overall, then, the runtime complexity of Algo. 2 is $\mathcal{O}((|V| + |E|) + |V| + (|V| + w^2 \cdot |V|)) = O(w^2 \cdot |V| + |E|)$.

Fig. 7 shows the runtime of checking the DU criterion as a function of the number of nodes, edges, and buds, as well as the vertex width of the graph, with each point in the figure averaging the runtime among all sloped graphs with the associated number of nodes/edges/buds or vertex width. Because of the high correlation between the number of nodes and edges in a sloped graph, we plot both metrics in a single graph. In Fig. 7b, we see the highest runtimes around width 10 because the graphs that have the most amount of nodes/edges in our database also all have a width of around 10. The trend in Fig. 7a is only somewhat linear because the algorithm only traverses the positions of nodes in cycles. A clear linear trend is observed in Fig. 7c which plots the runtime as a function of the number of buds in the graph, which, in unicycles graphs, is the number of cycles. This indicates that the algorithm performs what seems to be a constant amount of work for each cycle in the sloped graphs from our database. This implies that cycles have a consistent size, and together with Fig. 7a we can infer that as the size of cyclic pre-proofs grow, so does the number of cycles.

## 5 The CYCLONE Verifier and its Evaluation

Having reported above on the individual runtime performance of our implementations of each of the incomplete methods, we now present our integrated tool, CYCLONE, which combines these into a pipeline that defaults to a complete method (specifically, OR) for cases not covered by our new methods. We also present the results of our experimential evaluation, comparing CYCLONE with each of the complete decision procedures alone as they are implement in [9]. We implemented each of our new algorithms, as well as CYCLONE itself, in C++ and integrated into the Cyclist prover framework [6]. The experiments we report on, both in the current and previous section, were all performed on an Apple M1 CPU with 8GB of RAM, running macOS Sonoma.

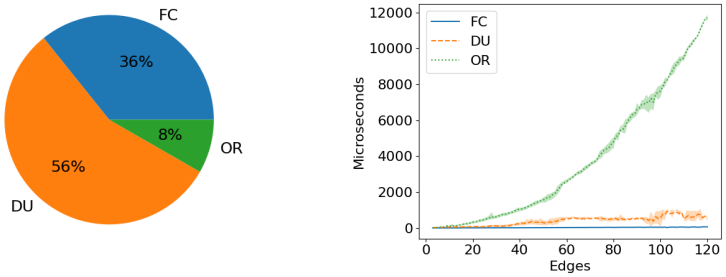
### 5.1 Composing the Methods in CYCLONE

The composition of the methods that CYCLONE uses is presented in Algo. 3. As mentioned in sec. 4.1, we do not use the TM method: CYCLONE only uses the

**Algorithm 3** Cyclone

**Input:** Sloped Graph $SG = (V, E, Ps, (R_{(v,v')})_{(v,v')\in E})$
**Output:** "yes" if $SG$ satisfies Infinite Descent and "no" otherwise

1: **if** InfDescByFC($SG$) = "no" **then return** "no"
2: $DU \leftarrow$ InfDescByDU($SG$)
3: **if** $DU \neq$ "don't know" **then return** $DU$
4: **return** InfDescByOR($SG$)



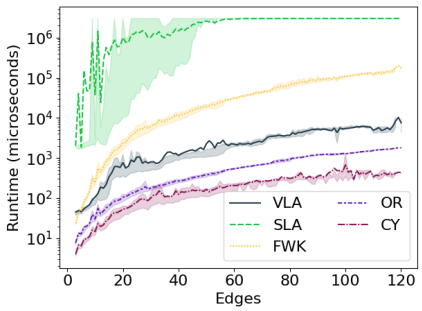

(a) Method Usage Distribution (b) Method Runtime

Fig. 8: Cyclone Methods Distribution and Runtime

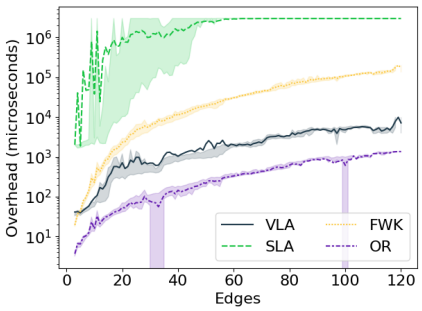
FC and DU methods. It first applies the FC criterion to try and return a very fast "no". If FC returns a "don't know", it then uses the DU criterion to try and return a (not as) fast "yes" or a "no". Finally, if DU also returns "don't know", Cyclone resorts to using the existing, complete and exponential OR algorithm to obtain a definitive answer.

Fig. 8a shows the distribution of methods from which Cyclone derived its answer when run on our database. Fig. 8b presents the average runtime of the different methods used in Cyclone, aggregated by number of edges, as well as the interquartile ranges. We plot the runtimes only as a function of the number of edges since we observed that the other parameters of the complexity analysis have a high correlation with this parameter. The FC method is considerably faster than the other two, with what appears to be a constant line. The DU method appears to be slower than the FC but is still much faster than OR, which looks at least polynomial. Note that the faster the method the lower its coverage, and recall that FC covers 35.76% of the graphs in the database, whilst DU covers 90.69%. The OR method, which has a complete coverage of the database, is the slowest method. We run the methods in ascending order of runtimes because even when FC returns a "don't know", this still happens fast enough that the overhead does not noticeably affect the overall runtime. The same is true for the DU method compared to the runtime of OR.

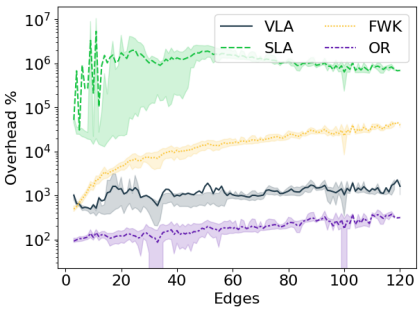
Fig. 8a shows that Cyclone decides Infinite Descent in polynomial time on around 92% of the sloped graphs in our database. This high coverage, together with the polynomial complexity of our incomplete methods, is what enables




(a) All Methods




(b) Absolute Overhead wrt CYCLONE




(c) % Overhead wrt CYCLONE

Fig. 9: Methods Runtime Comparison

the high performance of CYCLONE. Furthermore, as mentioned, even on the remaining 8% of graphs, the overhead of running them is so low compared to the runtime of the complete method that the overall performance is unaffected.

### 5.2 Comparison with State-of-the-Art Methods

We now report on our evaluation of CYCLONE against the state-of-the-art methods (VLA, SLA, FWK, and OR) for deciding Infinite Descent using the database described in Sec. 3. The figures in this section again present aggregated average runtime of each method by the number of edges, and the interquartile ranges. Additionally, we compare the runtime overhead of each of the existing methods with CYCLONE as the baseline.

Fig. 9a plots the runtime of all methods using a logarithmic scale. It shows a clear difference in the various methods' runtimes and, most importantly, that CYCLONE is the fastest among them. The runtime of every method grows somewhat sub-exponentially with the number of edges in the sloped graph. This might be because the sloped graphs in the database do not have many edges, so the exponential trend of the runtime does not yet manifest itself experimentally.

Fig. 9b and Fig. 9c present, respectively, the absolute overhead in milliseconds and the percentage overhead of each complete method with respect to CYCLONE.

Apart from the SLA method, the average percentage overhead increases as the graph size increases. This shows that the runtime complexity difference of the methods indeed manifests itself in the experimental results. Observe that Cyclone is between around 80% to around 350% faster than the fastest method (OR), between around 480% to around 2200% faster than VLA, between 480% and 43,000% faster than FWK.

Finally, looking at the SLA method we can see a constant line in the runtime. This is because we used a timeout of 3 seconds in our tests, and SLA seems to hit this timeout after a certain size of input. This explains why, in contrast with the other methods, the percentage overhead for SLA decreases as the number of edges increases. The timeout limit notwithstanding, we can still see a tremendous advantage to Cyclone, which is between around $3 \times 10^4\%$ and $5 \times 10^6\%$ faster than the SLA method. These results show that Cyclone significantly improves the practical runtime of the Infinite Descent check on real-world sloped graphs.

As noted in Sec. 3, our database contains the sloped graphs that Cyclist produced directly from concrete cyclic pre-proofs. However, Cyclist also pre-processes each sloped graph before handing it to the Infinite Descent check. This consists of pruning nodes that do not lie along cycles and collapsing non-branching paths, which considerably reduces the size of the graphs whilst maintaining the structure necessary for checking Infinite Descent. We also collected these minimised forms of the sloped graphs generated by Cyclist, and then evaluated Cyclone against the state-of-the-art methods on this preprocessed dataset. Here, Cyclone is still 90% to 170% faster than the best performing other method, which again is the OR method.

## 6 Conclusion

We introduced Cyclone, an efficient and general tool for deciding the Infinite Descent property, implemented by combining with existing exponential decision procedures two novel, incomplete but polynomial-time, algorithms exploiting statistically significant structural properties of sloped graphs. We demonstrated, on real-world data, Cyclone's superior runtime performance compared to existing approaches. Moreover, the Cyclone tool is open-ended in that it may incorporate additional semi-decision procedures as they are developed.

We evaluated Cyclone on a dataset generated by the Cyclist prover, consisting of graphs corresponding to cyclic pre-proofs from its test suites. To broaden coverage of real-world use cases, we plan to expand the dataset. In particular, since verifying Infinite Descent supports program termination verification via the size-change principle, we aim to create a dataset of termination instances, generated from, e.g. Agda's termination checker, for future evaluation. We also plan to explore preprocessing methods other than Cyclist's minimisation, which may better align with our methods and further improve Cyclone's performance.

## References

1. Agda Developers: Agda, https://agda.readthedocs.io/

2. Berardi, S., Tatsuta, M.: Classical system of martin-lof's inductive definitions is not equivalent to cyclic proofs. Log. Methods Comput. Sci. **15**(3) (2019). https://doi.org/10.23638/LMCS-15(3:10)2019
3. Brotherston, J.: Sequent Calculus Proof Systems for Inductive Definitions. Ph.D. thesis, University of Edinburgh (November 2006)
4. Brotherston, J., Bornat, R., Calcagno, C.: Cyclic proofs of program termination in separation logic. In: Necula, G.C., Wadler, P. (eds.) Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008. pp. 101–112. ACM (2008). https://doi.org/10.1145/1328438.1328453
5. Brotherston, J., Gorogiannis, N.: Cyclic abduction of inductively defined safety and termination preconditions. In: Müller-Olm, M., Seidl, H. (eds.) Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8723, pp. 68–84. Springer (2014). https://doi.org/10.1007/978-3-319-10936-7_5
6. Brotherston, J., Gorogiannis, N., Petersen, R.L.: A generic cyclic theorem prover. In: Jhala, R., Igarashi, A. (eds.) Programming Languages and Systems - 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7705, pp. 350–367. Springer (2012). https://doi.org/10.1007/978-3-642-35182-2_25
7. Brotherston, J., Simpson, A.: Sequent Calculi for Induction and Infinite Descent. Journal of Logic and Computation **21**(6), 1177–1216 (2010)
8. Cheng, K.S., Ngan, C.W., Trung, T.Q., Chanh, L.T., Sivaraman, A., Toan, N.T.: Songbird prover (2016), https://songbird-prover.github.io/
9. Cohen, L., Jabarin, A., Popescu, A., Rowe, R.N.S.: The complex(ity) landscape of checking infinite descent. Proceedings of the ACM on Programming Languages **8**(POPL), 1352–1384 (Jan 2024). https://doi.org/10.1145/3632888
10. Cohen, L., Rowe, R.N.S.: Non-Well-Founded Proof Theory of Transitive Closure Logic. ACM Trans. Comput. Logic **21**(4) (Aug 2020). https://doi.org/10.1145/3404889
11. Das, A.: On The Logical Complexity of Cyclic Arithmetic. Log. Methods Comput. Sci. **16**(1) (2020). https://doi.org/10.23638/LMCS-16(1:1)2020
12. Dax, C., Hofmann, M., Lange, M.: A Proof System for the Linear Time $\mu$-Calculus. In: Arun-Kumar, S., Garg, N. (eds.) FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science, 26th International Conference, Kolkata, India, December 13-15, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4337, pp. 273–284. Springer (2006). https://doi.org/10.1007/11944836_26
13. Doumane, A.: Constructive Completeness for the Linear-time $\mu$-calculus. In: Proceedings of the 32$^{\text{nd}}$ Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017. pp. 1–12 (2017). https://doi.org/10.1109/LICS.2017.8005075
14. Itzhaky, S., Peleg, H., Polikarpova, N., Rowe, R.N.S., Sergey, I.: Cyclic program synthesis. In: Freund, S.N., Yahav, E. (eds.) PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021. pp. 944–959. ACM (2021). https://doi.org/10.1145/3453483.3454087
15. Jones, E., Ong, C.H.L., Ramsay, S.: Cycleq: An Efficient Basis for Cyclic Equational Reasoning. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 395–409. PLDI 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3519939.3523731

16. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL01, ACM (Jan 2001). https://doi.org/10.1145/360204.360210
17. Lehmann, D.J.: Algebraic structures for transitive closure. Theor. Comput. Sci. **4**(1), 59–76 (1977). https://doi.org/10.1016/0304-3975(77)90056-1
18. Lepigre, R., Raffalli, C.: Practical subtyping for curry-style languages. ACM Trans. Program. Lang. Syst. **41**(1), 5:1–5:58 (2019). https://doi.org/10.1145/3285955
19. Nollet, R., Saurin, A., Tasson, C.: Pspace-completeness of a thread criterion for circular proofs in linear logic with least and greatest fixed points. In: Cerrito, S., Popescu, A. (eds.) Automated Reasoning with Analytic Tableaux and Related Methods - 28th International Conference, TABLEAUX 2019, London, UK, September 3-5, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11714, pp. 317–334. Springer (2019). https://doi.org/10.1007/978-3-030-29026-9_18
20. Rowe, R.N.S., Brotherston, J.: Automatic cyclic termination proofs for recursive procedures in separation logic. In: Bertot, Y., Vafeiadis, V. (eds.) Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017. pp. 53–65. ACM (2017). https://doi.org/10.1145/3018610.3018623
21. Santocanale, L.: A Calculus of Circular Proofs and Its Categorical Semantics. In: Nielsen, M., Engberg, U. (eds.) Proceedings of the $5^{\text{th}}$ International Conference on Foundations of Software Science and Computation Structures, FOSSACS 2002. pp. 357–371. Berlin, Heidelberg (2002)
22. Serban, C., Iosif, R.: An entailment checker for separation logic with inductive definitions. Electronic Communications of the EASST **76** (May 2019). https://doi.org/10.14279/tuj.eceasst.76.1073
23. Sprenger, C., Dam, M.: A note on global induction in a mu-calculus with explicit approximations. In: Ésik, Z., Ingólfsdóttir, A. (eds.) Fixed Points in Computer Science, FICS 2002, Copenhagen, Denmark, 20-21 July 2002, Preliminary Proceedings. BRICS Notes Series, vol. NS-02-2, pp. 22–24. University of Aarhus (2002)
24. Ta, Q., Le, T.C., Khoo, S., Chin, W.: Automated mutual explicit induction proof in separation logic. In: Fitzgerald, J.S., Heitmeyer, C.L., Gnesi, S., Philippou, A. (eds.) FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9995, pp. 659–676 (2016). https://doi.org/10.1007/978-3-319-48989-6_40
25. Ta, Q., Le, T.C., Khoo, S., Chin, W.: Automated lemma synthesis in symbolic-heap separation logic. Proc. ACM Program. Lang. **2**(POPL), 9:1–9:29 (2018). https://doi.org/10.1145/3158097
26. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM Journal on Computing **1**(2), 146–160 (Jun 1972). https://doi.org/10.1137/0201010
27. Tellez, G., Brotherston, J.: Automatically verifying temporal properties of pointer programs with cyclic proof. J. Autom. Reason. **64**(3), 555–578 (2020). https://doi.org/10.1007/s10817-019-09532-0

# A Additional Material for Section 4

## A.1 Details on The Trace Manifold Criterion

Here we present the details of the incomplete *Trace Manifold* criterion from sec. 4.1. Recall that the criterion applies to sloped graphs that are trees with backlinks, in cycle normal form, for which we have observed that the basic cycles are in one-to-one correspondence with the buds. In the formulation that follows, we thus refer to the basic cycle associated with a bud $B$ using the notation $C_B$. Formally, this cycle is the path from the companion of $B$, denoted $\mathcal{R}(B)$, to $B$ itself. We also reprise, formally, the definition of the structural connectivity relation.

**Definition 8 (Structural Connectivity).** *Let $SG$ be a sloped graph whose underlying graph is a tree with backlinks, in cycle normal form. The* structural connectivity *relation, $\leq_{SG}$, for $SG$ is the relation on the buds of $SG$ defined by $B_1 \leq_{SG} B_2$ iff $\mathcal{R}(B_2)$ appears on the basic cycle $C_{B_2}$.*

Notice it is immediate from this definition that $B \leq_{SG} B$ always holds, for any bud $B$.

We now recall the definition of a trace manifold for such a sloped graph.

**Definition 4 (Trace Manifold).** *A set of (finite) traces for paths in a sloped graph $SG$ is called a* trace manifold *when it has the following form*

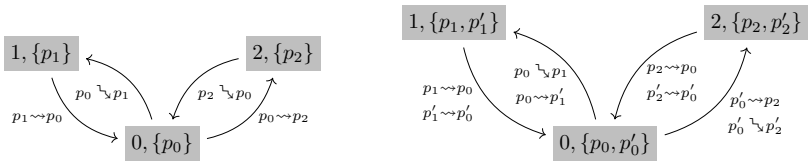
$$\{\tau_{S,B} \mid S \text{ a strongly connected subgraph of } SG, B \in S \text{ is a bud}\}$$

*and satisfies the following.*

*(1) Each $\tau_{S,B}$ is a trace along the basic cycle of $B$.*
*(2) For all $\tau_{S,B_1}$ and $\tau_{S,B_2}$, if $B_1 \leq_{SG} B_2$ then $\tau_{S,B_1}(\mathcal{R}(B_1)) = \tau_{S,B_2}(\mathcal{R}(B_1))$.*
*(3) For every strongly connected subgraph, $S$, of $SG$ there is a bud $B \in S$ such that $\tau_{S,B}$ has at least one progressing point.*

We note that this definition differs slightly from Brotherston's [3, Def. 7.2.1], which indexes the elements of trace manifolds by subgraphs defined as the union of the basic cycles associated with a set of buds that is weakly connected by the structural connectivity relation (i.e. a set of buds for which, when viewing the structural connectivity relation as a graph, there is an undirected path between any two buds in the set). Brotherston in fact shows ([3, Lemma 7.1.7]) that these are in one-to-one correspondence with the strongly connected subgraphs. Thus, we choose to define trace manifolds in terms of the latter, since we feel that this is a more natural definition.

*Example 4.* Fig. 10 shows two sloped graphs having same underlying directed graph, which has two overlapping basic cycles, $c_1 = 0, 1$ and $c_2 = 0, 2$, associated with bud nodes 1 and 2, respectively, both having companion node 0. This means that there are three strongly connected components: $\{0, 1\}$, $\{0, 2\}$ and $\{0, 1, 2\}$. Moreover, we have that $1 \leq_{SG} 2$ and $2 \leq_{SG} 1$. The sloped graph in Fig. 10a




(a) Satisfies TM (b) Does not satisfy TM

Fig. 10: Graphs with Overlapping Cycles


(a) Satisfies TM (b) Does not satisfy TM

Fig. 11: Unicycles Graphs

satisfies TM, since we can pick the following traces: $\tau_{\{0,1\},1} = \tau_{\{0,1,2\},1} = p_0, p_1$ and $\tau_{\{0,2\},2} = \tau_{\{0,1,2\},2} = p_0, p_2$. However, the sloped graph in Fig. 10b does not satisfy TM because there is no choice of traces for the strongly connected component $\{0, 1, 2\}$ that all satisfy condition (2) of Def. 4 at the same time: if we choose, e.g., $\tau_{\{0,1,2\},1} = p_0, p_1$ and $\tau_{\{0,1,2\},2} = p'_0, p'_2$, then the condition is satisfied for the choices $B_1 = B_2 = 1$ and $B_1 = B_2 = 2$, but not the choice $B_1 = 1$ and $B_2 = 2$ since $p_0 \neq p'_0$. Even though this sloped graph fails to satisfy TM, it *does* in fact satisfy Infinite Descent. In particular, this is a sloped graph corresponding to Berardi and Tatsuta's 2-hydra example [2].

*Example 5.* Note that when a sloped graph is a unicycles graph, the TM criterion is stricter than the DU criterion. That is because a trace manifold for a unicycles graph requires there to be a trace with a progression point along a *single* travsersal of a basic cycle, while the DU criterion more permissively allows several consecutive traversals of the cycle in order to witness a descent. Fig. 11 presents two sloped graphs having the same underlying directed graph, which is a unicycles graph, that are both simply descending. However, whilst the graph in Fig. 11a satisfies TM, the graph in Fig. 11b does not, because we cannot pick any trace $\tau_{\{0,1\},1}$ along the cycle $0, 1$ that satisfies both conditions (2) and (3) of Def. 4: although the trace $p_0, p'_1$ has a progression point, it does not return to the position $p_0$ in the companion 0 and so fails to satisfy condition (2); furthermore, although the trace $p'_0, p'_1$ returns to the position $p'_0$ in the companion 0, it has no progression points and so fails to satisfy condition (3).

**Algorithm.** Our algorithm for deciding the TM criterion essentially derives from the following notion a *trace manifold graph*.

**Definition 9 (Trace manifold graph).** *Let $SG$ be a sloped graph that is a tree with backlinks, in cycle normal form. The* trace manifold graph *of $SG$ is the directed graph $\mathcal{G} = (\mathcal{T}, \mathcal{E})$ defined by:*

$$\mathcal{T} = \{(B, \tau) \mid B \text{ is a bud of } SG \text{ and } \tau \text{ is a trace of } C_B\}$$
$$\mathcal{E} = \{((B, \tau), (B', \tau')) \in \mathcal{T} \times \mathcal{T} \mid B \leq_{SG} B',\ \tau(\mathcal{R}(B)) = \tau'(\mathcal{R}(B))\}$$

The following crucial property relates trace manifolds to the trace manifold graph just defined.

**Proposition 4.** *Let $SG$ be a sloped graph that is a tree with backlinks, in cycle normal, and suppose $\mathsf{Buds}(SG)$ is the set of its buds. Moreover, let $\mathcal{G} = (\mathcal{T}, \mathcal{E})$ be the trace manifold graph of $SG$. Then $SG$ has a trace manifold iff for every weakly $\leq_{SG}$-connected set $\mathcal{B} \subseteq \mathsf{Buds}(SG)$ we have that there is a weakly connected set $\mathcal{T}_\mathcal{B} \subseteq \mathcal{T}$ of trace manifold nodes such that:*

*(1) For each $B \in \mathcal{B}$, there is some trace $\tau$ such that $(B, \tau) \in \mathcal{T}_\mathcal{B}$.*
*(2) For all $B, B' \in \mathcal{B}$ such that $B \leq_{SG} B'$, there are traces $\tau$ and $\tau'$ such that $(B, \tau) \in \mathcal{T}_\mathcal{B}$, $(B', \tau') \in \mathcal{T}_\mathcal{B}$, and $((B, \tau), (B', \tau')) \in \mathcal{E}$*
*(3) There is some $(B, \tau) \in \mathcal{T}_\mathcal{B}$ such that $\tau$ has a progression point.*

*Proof.* ($\Rightarrow$) Suppose $SG$ has a trace manifold, and let $\mathcal{B} \subseteq \mathsf{Buds}(SG)$ be a weakly $\leq_{SG}$-connected set. Let $S = \bigcup_{B \in \mathcal{B}} C_B$ be the subgraph of $SG$ that is the union of the basic cycles $C_B$ associated with the buds $B \in \mathcal{B}$. Note that this is subgraph is strongly connected (cf. [3, Lemma 7.1.7]). Also, clearly, the subset of its nodes that are buds is exactly $\mathcal{B}$. Take the set of nodes $\mathcal{T}_\mathcal{B} = \{(B, \tau) \in \mathcal{T} \mid B \in \mathcal{B}\} \subseteq \mathcal{T}$. We now show that $\mathcal{T}_\mathcal{B}$ is a weakly connected set in $\mathcal{G}$. Let $(B, \tau), (B', \tau') \in \mathcal{T}_\mathcal{B}$. Thus, $B, B' \in \mathcal{B}$. Since $\mathcal{B}$ is weakly $\leq_{SG}$-connected, we know that there is an undirected $\leq_{SG}$-path $B_0, \ldots, B_m$ ($m > 0$), with $B = B_0$ and $B' = B_m$ such that for every $i < m$ we have $B_i \leq_{SG} B_{i+1}$ or $B_{i+1} \leq_{SG} B_i$. By (2) in the definition of a trace manifold, we also get that (w.l.o.g) $\tau_{S,k_p}(\mathcal{R}(B_{k_p})) = \tau_{S,k_{p+1}}(\mathcal{R}(B_{k_p}))$. Therefore, (w.l.o.g) $((C_{B_{k_p}}, \tau_{S,k_p}), (C_{B_{k_p+1}}, \tau_{S,k_{p+1}})) \in E_\tau$. Thus, we get an undirected path from $(C_{B_i}, \tau_{S,i})$ to $(C_{B_j}, \tau_{S,j})$ in $\mathcal{G}$, which means that $\mathcal{T}_S$ is a weakly connected set in $\mathcal{G}$. Now we show that the three requirements in Prop. 4 hold:

1. By the definition of $\mathcal{T}_S$ we know that for every $B_i \in \mathcal{B}_S$ we have that $(C_{B_i}, \tau_{S,i}) \in \mathcal{T}_S$.
2. Let $B_i, B_j \in \mathcal{B}_S$ such that $B_i \leq_{SG} B_j$. By the definition of $\mathcal{T}_S$, we have that $(C_{B_i}, \tau_{S,i}), (C_{B_j}, \tau_{S,j}) \in \mathcal{T}_S$ and also, because $B_i \leq_{SG} B_j$, by (2) in the definition of the trace manifold we know that $\tau_{S,i}(\mathcal{R}(B_i)) = \tau_{S,j}(\mathcal{R}(B_i))$, which means that by the definition of $\mathcal{E}$ we have $((C_{B_i}, \tau_{S,i}), (C_{B_j}, \tau_{S,j})) \in \mathcal{E}$.
3. From (3) in the definition of the trace manifold, we get that there is an $i$ such that $\tau_{S,i}$ has a progress point, which means that $(C_{B_i}, \tau_{S,i}) \in \mathcal{T}_S$ and that $\tau_{S,i}$ is progressing

($\Leftarrow$) Let $\mathcal{B}_S \subseteq Bud(\mathcal{D})$ be a weakly $\leq_{SG}$-connected set, $\mathcal{C}_S := \{C_B \mid B \in \mathcal{B}_S\}$ and $S := \bigcup \mathcal{C}_S$. Because $\mathcal{B}_S$ is weakly $\leq_{SG}$-connected, we know that there is a

weakly connected set $\mathcal{T}_S \subseteq \mathcal{T}$ in $\mathcal{G}$ that satisfies all the conditions in Prop. 4. We will now show that $\tau := \{\tau_B \mid \exists C_B \in \mathcal{C}_S.\ (C_B, \tau_B) \in \mathcal{T}_S\}$ is a part of a trace manifold:

1. Note that, by the definition of $\mathcal{T}_S$, we know that for every $B_i \in \mathcal{B}_S$, there is a $\tau_{B_i} \in \tau$ such that $\tau_{B_i}$ is a trace of $C_{B_i}$.
2. Let $B_i, B_j \in \mathcal{B}_S$ and suppose that $B_j \leq_{SG} B_i$. By (2) in the definition of the trace manifold graph, we get that $\exists \tau_{B_j}, \tau_{B_i}.\ (C_{B_j}, \tau_{B_j}), (C_{B_i}, \tau_{B_i}) \in \mathcal{T}_S$ and that $((C_{B_j}, \tau_{B_j}), (C_{B_i}, \tau_{B_i})) \in E_\tau$. By the definition of $E_\tau$ we get that $\tau_i(\mathcal{R}(B_j)) = \tau_j(\mathcal{R}(B_j))$.
3. By (3) in the definition of the trace manifold graph we get that $\exists \tau_{B_i}$ $\exists C_{B_i}.\ (C_{B_i}, \tau_{B_i}) \in \mathcal{T}_S$, that $\tau_{B_i}$ has a progress point and also that $\tau_{B_i}$ is a trace of $B_i$ □

The nodes of a trace manifold graph can be generated using the following notion of the *positions graph*.

**Definition 10 (Positions graph).** *Let $SG = (V, E, Ps, (R_{(v,v')})_{(v,v')\in E})$ be a sloped graph. The* positions graph *of $SG$ is the directed graph $SG^{pos} = (V', E')$, with a distinguished subset of* progressing *edges $Prog \subseteq E'$, defined by:*

$$V' = \{(v, p) \mid v \in V,\ p \in Ps(v)\}$$
$$E' = \{((v, p), (u, q)) \mid v, u \in V,\ \exists s.\ (p, q, s) \in R_{v,u}\}$$
$$Prog = \{((v, p), (u, q)) \mid v, u \in V,\ (p, q, \searrow) \in R_{v,u}\}$$

The nodes of the trace manifold graph can be computed via a DFS on the positions graph to find the traces of each basic cycle, and the edges by checking which traces of two $\leq_{SG}$-related cycles agree on the traced element of the source bud's companion.

We can thus formulate an algorithm for deciding the trace manifold criterion by generating the trace manifold graph and traversing it to check the requirements in Prop. 4. Specifically Algo. 4, which takes a sloped graph $SG$ that is a tree with backlinks (the root node and its set of buds thus being part of the input), performs exactly this check and thus returns a "yes" if the graph has a trace manifold (in which case it satisfies Infinite Descent, by Prop. 1) or a "don't know" otherwise.

- First, the algorithm checks whether the input graph is in cycle normal form also calculating, if so, the structural connectivity relation $\leq_{SG}$ and each bud's basic cycle and companion.
  - The companions can be found using a DFS which starts from the root of the sloped graph: when the DFS gets to a bud, it recognizes its only neighbour as its companion.
  - If the out-edge of the bud is node previously encountered in the DFS, then the basic cycle of the bud is recognized as the path from its companion to itself. Otherwise, the companion is not an ancestor of the bud, which means that the sloped graph is not in cycle normal form. If this is the case, the algorithm immediately returns a "don't know".

**Algorithm 4** Infinite Descent by the Trace Manifold Criterion

**Input:** Sloped Graph $SG = (V, E, Ps, R)$, a buds set $Bud$
**Output:** "yes" if $SG$ has a trace manifold, "don't know" otherwise

```
1: ≤_SG, basicCyc, companions, isCNF ← getStrCon(SG, Bud)        ▷ O(|V| + |E| + |Bud|^2)
2: if ¬isCNF then
3:     return "don't know"
4: T, progTraces ← getTracesAllCycles(SG, basicCyc)              ▷ O(|V||Ps|^|V|)
5: E ← getTMGraphEdges(SG, T, ≤_SG, companions)                  ▷ O(|Bud|^2 |Ps|^|V|)
6: WCSs ← getWeaklyConnectedSets(≤_SG)                           ▷ O(|Bud|^2 · 2^|Bud|)
7: for all WCS ∈ WCSs do                                         ▷ 2^|Bud| iterations
8:     if ¬hasSubmanifold(WCS, E, progTraces, ≤_SG) then         ▷ O(|V||Ps|^|V| + |Bud|^2 |Ps|^|V|)
9:         return "don't know"
10: return "yes"
```

- The $\leq_{SG}$ relation is then calculated by going over every two basic cycles and their companions and checking whether the companion of one cycle is a part of the other basic cycle.

– If the graph is in cycle normal form, the algorithm generates the trace manifold graph using the positions graph, as described above.
– Then, the algorithm finds all of the weakly $\leq_{SG}$-connected sets, and for every such set, it checks if it has a submanifold. A submanifold of a set of basic cycles $\mathcal{B}_S$ is a set of traces for each basic cycle in $\mathcal{B}_S$ that satisfies the three requirements of the trace manifold, just without the "for all $S$" quantification. Finding a submanifold amounts to choosing a basic cycle in the current weakly $\leq_{SG}$-connected set, running an un-directed DFS on the trace manifold graph from every trace of this basic cycle, and checking if the following hold:
  1. the DFS traverses all of the basic cycles in the current set;
  2. every two $\leq_{SG}$-connected basic cycles have an edge traversed by the DFS;
  3. the DFS encountered at least one progressing trace.

  If such a DFS exists, then by Prop. 4 it is easy to see that the nodes of the trace manifold graph that it visited form a submanifold.
– If for every weakly $\leq_{SG}$-connected component there exists such a DFS, then all of those DFS's form a trace manifold, and thus, in that case, the algorithm would return a "yes". Otherwise it returns a "don't know", as the absence of a trace manifold does not imply that the graph does not satisfy Infinite Descent.

**Complexity.** In terms of complexity, note that a trace manifold has a trace for every pair $(\mathcal{B}_S, i)$ such that $\mathcal{B}_S$ is a subset of weakly $\leq_{SG}$-connected buds and $B_i \in \mathcal{B}_S$. This means that in the worst case, the trace manifold has a size that is exponential in the number of buds (if they are all weakly $\leq_{SG}$-connected). In

addition, for each such subset, we would need to go over all of the traces inside each basic cycle, which, again, in the worst case would be exponential, this time in the amount of nodes in each cycle. That would be the case when every position of every node in the basic cycle is connected to every position of the next node in the cycle, which would make $\mathcal{O}(|Ps|^{|c|})$ when $|c|$ is the number of nodes in the cycle. While this gives an upper bound on the size of the trace manifold, we are also interested in an upper bound on the runtime of finding such a trace manifold.

The overall runtime complexity of the algorithm is:

$$\mathcal{O}(|E| + 2^{|Bud|}(|V||Ps|^{|V|} + |Bud|^2|Ps|^{|V|}))$$

Line 1 uses a DFS on the nodes graph and also checks if each pair of basic cycles are $\leq_{SG}$-connected, which takes $\mathcal{O}(|V| + |E| + |Bud|^2)$. Line 4 simply explores all positions paths along each basic cycle, which, for every cycle $C$, takes $\mathcal{O}(|Ps|^{|C|})$. In the worst case, in which the graph is a string of companions with buds as leaves along the string, and the sizes of the basic cycles are $1, 2, ..., |V|$, we get that Line 4 takes $\sum_{i\in[1..|V|]} |Ps|^i = \mathcal{O}(|V||Ps|^{|V|})$ for all cycles together. This also means that in the worst case, the trace manifold graph has at most $|V||Ps|^{|V|}$ nodes ($|\mathcal{T}| = \mathcal{O}(|V||Ps|^{|V|})$). In Line 5 the algorithm goes over every two $\leq_{SG}$-connected basic cycles and over every pair of their traces. In the worst case, a basic cycle of length $|C|$ has $|Ps|^{|C|}$ traces if every position of one node in the cycle is connected to every position of the next node in the cycle. Note that, since $\leq_{SG}$ is a binary relation over $Bud$, we have that $|\leq_{SG}| = \mathcal{O}(|Bud|^2)$. This means that Line 5 takes:

$$\sum_{(B_i,B_j)\in\leq_{SG}} \mathcal{O}(|Ps|^{|C_i|}|Ps|^{|C_j|}) =$$
$$\sum_{(B_i,B_j)\in\leq_{SG}} \mathcal{O}(|Ps|^{|V|}|Ps|^{|V|}) =$$
$$\mathcal{O}(|\leq_{SG}||Ps|^{|V|}) = \mathcal{O}(|Bud|^2|Ps|^{|V|})$$

This also means that in the worst case, the trace manifold graph has at most $|\leq_{SG}||Ps|^{|V|}$ edges ($|\mathcal{E}| = \mathcal{O}(|\leq_{SG}||Ps|^{|V|}) = \mathcal{O}(|Bud|^2 \cdot |Ps|^{|V|})$). Then in Line 6 we get the weakly $\leq_{SG}$-connected sets by going over every subset of $Bud$ and using a DFS on that subset, which takes $\mathcal{O}(|\leq_{SG}|2^{|Bud|}) = \mathcal{O}(|Bud|^2 \cdot 2^{|Bud|})$. Line 8 runs an undirected DFS from every trace of some set basic cycle in the current weakly $\leq_{SG}$-connected set, which takes $\mathcal{O}(|\mathcal{T}| + |\mathcal{E}|) = \mathcal{O}(|V||Ps|^{|V|} + |Bud|^2 \cdot |Ps|^{|V|})$. In addition, Line 8 checks if the weakly $\leq_{SG}$-connected set has an edge between any two $\leq_{SG}$-connected basic cycles, which takes $\mathcal{O}(|\leq_{SG}|)$ if $\mathcal{E}$ is stored as a hash map. Finally Line 8 checks if the DFS traversed a progressing trace, which takes $\mathcal{O}(1)$ per trace traversed in the DFS, which in the worst case is the amount of traces of the current weakly $\leq_{SG}$-connected set, which is $\mathcal{O}(|\mathcal{T}|)$. All together,

Line 8 takes

$$\mathcal{O}(|\mathcal{T}| + |\mathcal{E}| + | \leq_{SG} | + |\mathcal{T}|) =$$
$$\mathcal{O}(|\mathcal{T}| + |\mathcal{E}| + | \leq_{SG} |) =$$
$$\mathcal{O}(|V||Ps|^{|V|} + |Bud|^2 \cdot |Ps|^{|V|} + |Bud|^2) =$$
$$\mathcal{O}(|V||Ps|^{|V|} + |Bud|^2 \cdot |Ps|^{|V|})$$

Since there are $\mathcal{O}(2^{|Bud|})$ iterations running Line 8, we get that the loop in Line 7 takes $\mathcal{O}(2^{|Bud|}(|V||Ps|^{|V|} + |Bud|^2|Ps|^{|V|}))$.
Looking at the whole algorithm, it takes:

$$\mathcal{O}(|V| + |E| + |Bud|^2 + |V||Ps|^{|V|} + |Bud|^2|Ps|^{|V|} +$$
$$|Bud|^2 \cdot 2^{|Bud|} + 2^{|Bud|}(|V||Ps|^{|V|} + |Bud|^2|Ps|^{|V|})) =$$
$$\mathcal{O}(|V||Ps|^{|V|} + |E| + |Bud|^2|Ps|^{|V|} + 2^{|Bud|}(|Bud|^2 + |V||Ps|^{|V|} + |Bud|^2|Ps|^{|V|})) =$$
$$\mathcal{O}(|E| + 2^{|Bud|}(|V||Ps|^{|V|} + |Bud|^2|Ps|^{|V|}))$$

Note that if the graph is not in cycle normal form Algo. 4 returns early after only $\mathcal{O}(|V| + |E| + |Bud|^2)$.

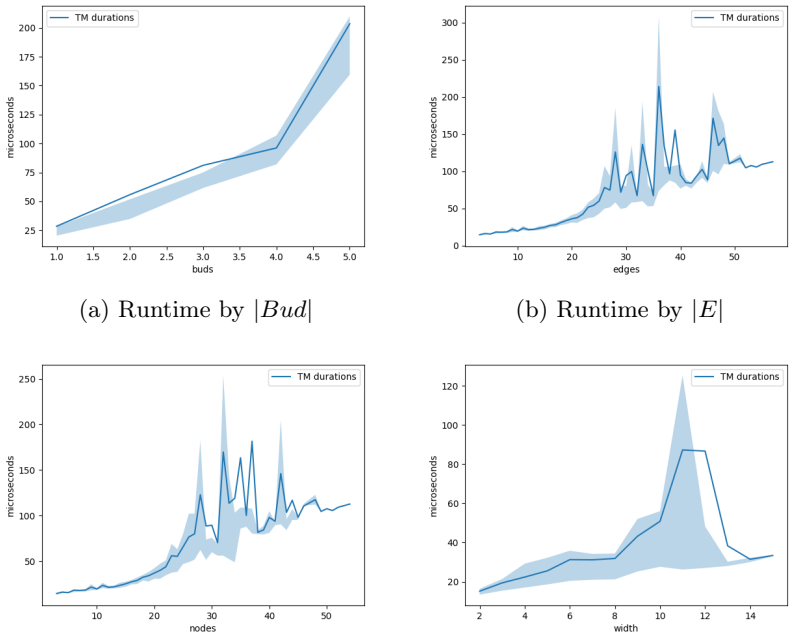
**Practical Runtime Evaluation.** Fig. 12 plots the average runtime of Algo. 4 on graphs in cycle normal form, grouped by the number of buds, the number of nodes and edges in the sloped graph and by its width. In Fig. 12a we can see that our database does not contain sloped graphs in cycle normal form with many buds, only at most 5, so a trend in $|Bud|$ is difficult to see experimentally. We can see a rising trend in the runtime by every parameter, but it is difficult to see how the runtime complexity is reflected in the figures. That is because the runtime complexity is tight when viewed in terms of $|\mathcal{T}|$, $|\mathcal{E}|$ and $| \leq_{SG} |$, as the trace manifold graph and the structural connectivity relation are the data structures that the algorithm traverses and not the sloped graph.

When analysed through this lens, the runtime complexity of the algorithm is $\mathcal{O}(2^{|Bud|}(| \leq_{SG} | + |\mathcal{T}| + |\mathcal{E}|))$, and Fig. 13 plots the runtime of the algorithm by these parameters. We can see a somewhat linear trend in Fig. 13a, Fig. 13b and Fig. 13c which reflects the linearity in $| \leq_{SG} |$, $|\mathcal{T}|$ and $|\mathcal{E}|$ in the runtime complexity analysis.

Fig. 14 presents the runtime of Algo. 4 only on graphs not in cycle normal form. On these graphs, our runtime complexity analysis presents a linear complexity in both $|V|$ and $|E|$, which is reflected in Fig. 14a and Fig. 14b respectively. Again, our database does not contain sloped graphs with many buds, only at most 16 in graphs not in cycle normal form, so again, it is difficult to see a trend in the runtime by $|Bud|$.

### A.2 Proofs for the Flat Cycles Criterion

**Proposition 2 (Flat Cycles criterion (FC)).** *Let SG be a sloped graph. If $SG^{\rightsquigarrow}$ has a cycle then SG does not satisfy Infinite Descent.*




(a) Runtime by $|Bud|$

(b) Runtime by $|E|$

(c) Runtime by $|V|$

(d) Runtime by $|Ps|$

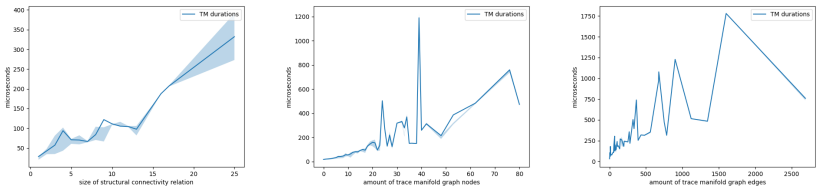
Fig. 12: Trace Manifold Runtime on Graphs in CNF

*Proof.* Let $v_1, ..., v_n$ be a cycle in $SG^{\rightsquigarrow}$. Then, let $(p_i)_{i\in\mathbb{N}}$ be a trace for the infinite path $(v_i)_{i\in\mathbb{N}} := (v_1, ..., v_n)^\omega$. By the construction of $SG^{\rightsquigarrow}$, this infinite path is also an infinite path in $SG$ and $\forall i \in \mathbb{N}$ we have that $(p_i, p_{i+1}, \searrow) \notin R_{v_i,v_{i+1}}$, which means that the trace $(p_i)_{i\in\mathbb{N}}$ is *not* decreasing. This is true for all traces along the infinite path $((v_i)_{i\in\mathbb{N}})^\omega$ and thus it is not descending. Hence, the sloped graph $SG$ has an infinite path that is not descending, which means it does not satisfy Infinite Descent. □

### A.3 Proofs for the Descending Unicycles Criterion

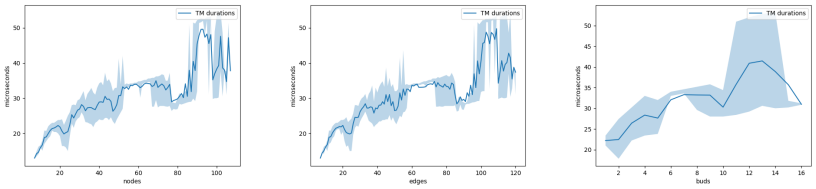
**Proposition 3 (Descending Unicycles criterion (DU)).** *If $SG$ is a unicycles sloped graph, then $SG$ satisfies Infinite Descent iff it is simply descending.*

*Proof.* ($\Rightarrow$) Let $SG$ be a unicycles sloped graph which satisfies Infinite Descent and let $c$ be a cycle in $SG$. Observe the infinite path $\pi := c^\omega$. Because $SG$ satisfies Infinite Descent we get that there is a decreasing trace $\tau$ for $\pi$. Denote $\pi^{Ps} := ((\pi_i, \tau_i))_{i\in\mathbb{N}}$ and note that it is a path in $SG_c^{pos}$. Because $\pi^{Ps}$ is infinite, we get that it has a cycle at its limit, which we denote as $c^{Ps}$. Also, because $\tau$ is descending, we get that there is a progressing edge $((v, p), (u, q))$ in $c^{Ps}$. Note that while $c^{Ps}$ is not necessarily a *basic* cycle, every edge in $c^{Ps}$ is contained




(a) Runtime by $|\leq_{\mathcal{P}}|$ (b) Runtime by $|\mathcal{T}|$ (c) Runtime by $|\mathcal{E}|$

Fig. 13: Trace Manifold Runtime on Graphs in CNF




(a) Runtime by Nodes (b) Runtime by Edges (c) Runtime by Buds

Fig. 14: Trace Manifold Runtime on Graphs Not in CNF

in some basic cycle in $SG_c^{pos}$. Thus, there is some basic cycle in $SG_c^{pos}$ which contains the aforementioned progressing edge, which means that $c$ is a descending cycle. This holds for all cycles in $SG$, which makes it a simply descending graph. ($\Leftarrow$) Let $SG$ be a simply descending unicycles sloped graph and let $\pi = v_0, v_1, ...$ be an infinite path in $SG$. Note that because $SG$ is a unicycles graph, $\pi$ has a single basic cycle $c$ at its limit, as otherwise, there would be a path between two basic cycles and back in $SG$, which would make it not a unicycles graph. Because $SG$ is a simply descending graph, $SG_c^{pos}$ has a basic cycle $c^{Ps}$ which has at least one progressing edge. Now, denote $\tau_c$ as the right component of each node in the path $(c^{Ps})^{\omega}$. Note that, by the definition of $SG_c^{pos}$, we get that $\tau_c$ is a trace for the infinite path $c^{\omega}$. Note, also, that $\tau_c$ is decreasing, since $c^{Ps}$ has at least one progressing edge. Thus, we get that $\tau_c$ is a decreasing trace for the infinite path $c^{\omega}$, which is a tail of the infinite path $\pi$, which means that $\pi$ is a descending infinite path. This holds for all infinite paths in $SG$, hence $SG$ satisfies Infinite Descent. □