Implementation of solving Unboundedness in 1-VASS with disequality guards in polynomial time

Quentin De Haes

June 25, 2021

This report is for a research project of the university of Antwerp. In this report, We discuss about deciding unboundedness of 1-vector addition system with states (1-VASS) with disequality guards in polynomial time. We mainly provide and discuss its implementation, which is my contribution for this research. The exact implementation in Python can be found in the following github repository: `https://github.com/QuentinDeHaes/Unboundedness-for-1-VASS`

1 Introduction

In this paper, we discuss 1-Vector addition systems with states and disequality guards(1-VASS). A 1-VASS can be described as a directed graph with a singular counter value and weights among the edges. We additionally introduce disequality constraints on the nodes of the graph. In [1], they concluded that the unboundedness of a 1-VASS with disequality guards could be decided in polynomial time. This paper expands upon that reasearch by creating and discussing an actual implementation of the algorithms proposed in [1].

The problems studied can be formalised as problems for types of one-counter machines, and the majority of the related work has been presented in this context. Under this correspondence, the value of the counter represents the accumulated weight along a path, and tests on the counter encode constraints on allowable paths. Algorithmic properties of one-counter machines have been studied by many authors over several decades [2, 3, 4, 5, 6, 7, 8, 9]. The above references are a small subset of the extensive literature on one-counter machines, but they illustrate that there are many variations on the basic model of a one-counter machine and that these variations can lead to substantially different algorithmic properties. Particular features mentioned in the references above, driven by applications to automated verification and program analysis, include equality tests, disequality tests, inequality tests, parametric tests, binary updates, polynomial updates, and parametric updates.

In this paper we consider 1-VASS with disequality tests, but no equality tests. In [1] the autors show that the unboundedness problem is solvable in

polynomial time for 1-VASS with disequality tests. Intuitively, we can say that if a configuration is unbounded, any configuration that can reach that unbounded configuration using a valid path is also unbounded. The main technical challenge the proposing paper had, is to obtain a polynomial-time bound when a run witnessing the given control state is unbounded. That run may have a length exponential in the size of the counter machine. In the algorithm, the presence of disequality tests proves to be rather disruptive: it destroys the monotonicity of the transition relation and prevents from freely iterating positive-weight cycles.

The proofs that validate the various techniques used to ensure a polynomial complexity will not be provided within this paper as they have been previously discussed [1] and are not my contribution to the research. My contribution is the implementation of the algorithm proposed in another paper [1] for calculating unboundedness with polynomial time-complexity in a 1-vector addition system with states and disequality guards. The implementation was made entirely in python 3.

2 Defintions[1]

One-Dimensional Vector Addition Systems with States and Tests (1-VASS). A 1-VASS with disequality tests can be defined as a tuple $\mathcal{V} = (Q, D, \Delta, w)$, where Q is a set of states, $D = \{D_q\}_{q \in Q}$ is a collection of cofinite subsets $D_q \subseteq \mathbb{N}$, $\Delta \subseteq Q \times Q$ is a set of transitions, and $w : \Delta \to Z$ is a function that assigns an integer weight to each transition. In the special case that each $D_q = \mathbb{N}$, $\forall q \in Q$. We simply call $\mathcal{V}$ a 1-VASS (and we omit the collection D).

A *configuration* of $\mathcal{V}$ is stated as a pair (q, z) with state $q \in Q$ and $z \in \mathbb{N}$, which we will refer to as the *counter value*. We write $Conf$ for the set $Q \times \mathbb{N}$ of all configurations. A partial order on $Conf$ will be imposed: $(q, z) \leq (q', z')$ if and only if $q = q'$ and $z \leq z'$. A configuration (q, z) is valid if $z \in D_q$ (and $q \in Q$).

A path in $\mathcal{V}$ is a sequence of states $\pi = q_1, ..., q_n$ such that $(q_i, q_{i+1}) \in \Delta$ for all $i \in \{1, ..., n-1\}$. When we have a path π, where $q_1 = q_n$ this path will be considered a *cycle*. The *weight* of π is defined to be $weight(\pi) := \sum_{i=1}^{n-1} w(q_i, q_{i+1})$. When we have a cycle of path π and $weight(\pi) > 0$, we have a *positive cycle*.

A (possibly empty) prefix of $\pi = q_1, ..., q_n$, a sequence $q_1, ..., q_m$ where $m \leq n$, is said to be *minimal* if it has minimal weight among all prefixes of π. Define $pmin(\pi)$, the *pmin value* of π, to be the weight of a minimal prefix of π.

A run is a sequence $(q_1, z_1), ..., (q_n, z_n)$ of configurations of $\mathcal{V}$ such that there is a path $\pi = q_1, ..., q_n$ with $z_{i+1} = z_i + w(q_i, q_{i+1})$ for $i = 1, ..., n-1$. We write $(q_1, z_1) \xrightarrow{\pi} (q_n, z_n)$ to denote such a run. Observe that a run is *valid* when it does not reach a negative counter value during any configuration along the run. Intuitively, a valid run through q can proceed if and only if the current counter value is in D_q, i.e. it does not violate any disequality guards. We say that a configuration (q', z') is *reachable* from (q, z) if there is a valid run π such that $(q, z) \xrightarrow{\pi} (q', z')$ that does not violate any constraints on the counter

value. Given a state q we represent the cofinite set D_q as the complement of an explicitly given subset of $\mathbb{N}$. Given this convention, we can assume without loss of generality that for all states q the set D_q is either $\mathbb{N}$ or $\mathbb{N} \setminus \{g\}$ for some $g \in \mathbb{N}$. For states q with $D_q = \mathbb{N} \setminus \{g\}$, we refer to the single missing value g in the domain as the *disequality guard* on q.

Coverability and Unboundedness. Let $\mathcal{V} = (Q, D, \Delta, w)$ be a 1-VASS with disequality tests, and let s and t be two distinguished states of $\mathcal{V}$. The *Coverability Problem* asks whether a valid run from $(s, 0)$ to (t, z), with $z \in \mathbb{N}$, exists. The *Unboundedness Problem* asks whether from configuration $(s, 0)$, we can reach an infinite amount of configurations, in which case we would call s *unbounded*. The Coverability problem can be reduced to the *unboundedness problem* by simply adding a positive cycle upon t and removing all states that cannot reach t in the underlying graph of $\mathcal{V}$. In this paper, we focus on the complexity of deciding the Unboundedness Problem.

3 Cycle detection

In this section we detect positive cycles within the graph. The algorithm uses positive cycles, since we could continue taking the cycle infinitely to achieve unboundedness. We need to ensure that detecting these cycles is done in polynomial time. We do not need to detect all positive cycles, we just need to look for the optimal positive cycle for each node. The optimal positive cycle is the cycle with the highest pmin value. If a node is not in any positive cycle, it does not have an optimal positive cycle.

3.1 Acquiring the optimal pmin value for each node

We start by looking for a bound in which the pmin value must be. We can acquire this bound by setting W as the largest absolute weight of the edges. Since a cycle cannot have a node occurring twice, we know that the length of a cycle would, at most, be all nodes of the 1-VASS, thus $|V|$. The pmin value would thus be lower bounded by $-W \cdot |V|$ and upper bounded by 0. 0 is the highest pmin value any positive cycle can get because we start the calculation of the pmin value of a cycle with the counter value initialised at 0. Trying all these bounded pmin values is not polynomial due to the fact that W is an exponential value if the input is in binary encoding. We can however reduce the complexity to polynomial by using a logarithmic algorithm to calculate the optimal pmin value. We do so by implementing a version of binary search, we look for a cycle using the center-most value between the bounds, we call that value x. If a cycle that does not go lower than x is found, the lower bound is updated to x. If no cycle is located, the upper bound is updated to x. And this is continued until the upper bound and lower bound have become the same. Then, we have found the optimal cycle along with its pmin value for a given node, or we can be certain that no positive cycle incorporating the node exists.

3.2 Detecting the cycle using a prospective pmin value

To detect the cycle, we use a variation of the Bellman-Ford algorithm.

3.2.1 Bellman-Ford algorithm

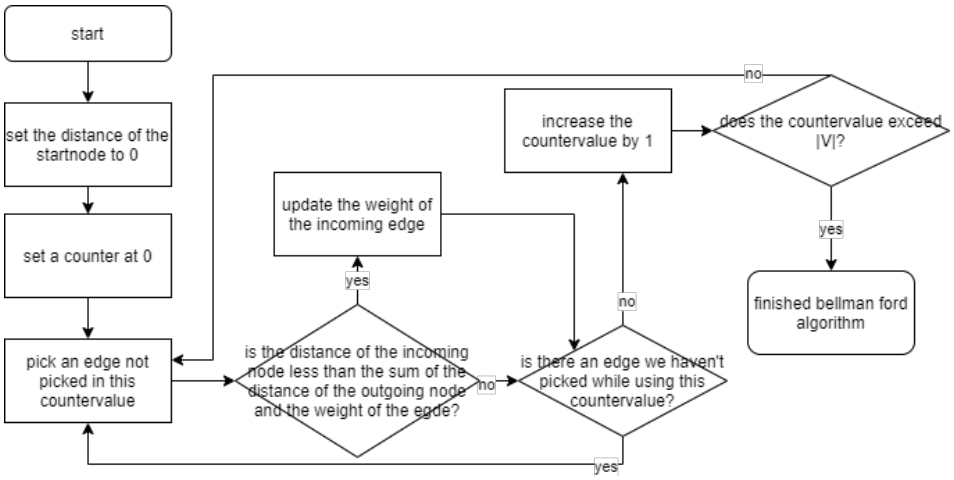

Figure 1: Flowchart of Bellman-Ford algorithm

The Bellman-Ford algorithm is an algorithm used for finding the shortest (lowest cost) path to each node in a directed weighted graph. A directed weighted graph is a graph where the edges have a cost provided to take them, and only go in a single direction. We will be using $|V|$ to denote the amount of nodes the graph has and $|E|$ to denote the amount of edges the graph has. Initially, each node is given a distance of $+\infty$, except for the start-node, this one is given a distance of 0 as done in step one of the flowchart (Section 3.2.1). Afterwards, we pass over every edge and check if taking that edge makes the cost of reaching the new node smaller than it currently is, i.e. if the sum of a nodes cost and the cost of the edge is less than the cost of the newly reached node, we edit the cost of the newly reached node to the smaller value. A single pass over all edges is clearly of efficiency $\mathcal{O}(|E|)$. But because the order of edges we pass over is random, with a single pass, we would only have all nodes with correct distance in the best case, if all edges are exactly how we want them. In the worst case, only a single extra node is given its correct distance. With this knowledge, we know how many times we need to run it in order to make sure each node has minimal distance. As each run makes at least one extra node have the correct value, if we run it once for each node ($|V|$ times), all will be minimal. The total efficiency is thus $\mathcal{O}(|V| \cdot |E|)$. There is a single caveat to this algorithm, when the graph has a negative cycle, we could take this cycle infinitely, and continue to get smaller cost values on the nodes. After running over all edges $|V|$ times,

if the next time we run over all edges, there is still a nodes cost that can be reduced, we know there exists a negative cycle.

3.2.2 Alterations to the Bellman-Ford algorithm

The first issue we face is that the Bellman-Ford algorithm locates negative cycles, while our algorithm requires positive cycles. This can be fixed easily by simply negating all weights assigned to the edges. When we do this, each negative cycle becomes a positive cycle and vice versa. This way, the existence of the positive cycles can be verified.

The second issue is that the basic Bellman-Ford algorithm can be used to simply verify the existence of cycles, not locate them exactly. But we can reason a way to locate them. After running over each edge $|V|$ times, if the next run an edge still causes a change in the costs, is this edge part of a cycle? The answer is possibly: this edge is either part of a cycle, or caused by a cycle. As the distances within the cycle keep getting updated, the distances of nodes that are reachable from the cycle will also update, due to the better distances the cycle offers. This knowledge can be used to continue updating the cost of nodes until the node for which we are looking for a cycle is the one that causes the next update. We can be sure that this is done within at most $|V|$ runs over all edges, because each run updates at least one node in the cycle to have a new, better score. Knowing a cycle has at most $|V|$ nodes in it, if during all $|V|$ runs, the node does not get an updated score, we know for certain that the node does not belong to a positive cycle.

After we have updated the score for our node, we can start looking for a positive cycle, we update the score for all outgoing edges of our node and keep a current score and the node from where the update came on all our updated nodes. However, if their score would reach below our minimal pmin, we do not update that node. Then we update all nodes for outgoing edges from all updated nodes from the previous iteration as long as it still does not violate the minimal pmin, and the new node either has yet to be reached previously, or the newly acquired score for that node is higher than any previously stored values. If any of these updates update the score for our starter node, then we located a cycle, we return this successful cycle along with its weight. We need to try this at most $|V|$ times since at that point, the cycle would have been located due to the size of the largest cycle being $|V|$, and every iteration, at least one node that is actually within the cycle would be updated.

3.3 Algorithm

Algorithm 1 Bellman ford update

```
1: function singleBellmanFordUpdate(graph)
2:     for node_from, node_to, weight in graph.edges do
3:         node_to.distance ← min(node_to.distance, node_from.distance −
   weight)          ▷ we use substraction to simulate negative edge weights
4:     end for
5: end function
```

Algorithm 2 Locate cycle of a single node

```
1: function locateCycleSingleNode(graph, startnode, minimal_score)
2:     ResetBelmanFordDistances(graph)        ▷ reset the distances from previous runs (set startnode to 0 and everything else to ∞)
3:     for _ in graph.nodes do                ▷ run the basic Bellman-Ford algorithm
4:         singleBellmanFordUpdate(graph)
5:     end for
6:     current_value ← startnode.distance
7:     for _ in graph.nodes do
8:         singleBellmanFordUpdate(graph)
9:         if startnode.distance has updated then
10:            break        ▷ keep running BF until our startnode will cause the future updates
11:        end if
12:    end for
13:    if startnode.distance has not updated in second loop then
14:        return                    ▷ our startnode is not in any (positive) cycle
15:    end if
16:    found_nodes ← [startnode]
17:    for _ in graph.nodes do
18:        new_found_nodes ←empty set
19:        for node_from, node_to, weight in graph.edges do
20:            if node_from in found_nodes AND node_to.distance > node_from.distance - weight AND node_from.pmin + weight ≥ minimal_score then
21:                node_to.distance ← node_from.distance − weight
22:                node_to.pmin ← node_from.pmin + weight
23:                node_to.origin ← node_from
24:                new_found_nodes.add(node_to)
25:            end if
26:            if startnode.origin has been updated then    ▷ a cycle has been found
27:                return generateCycle(startnode)    ▷ cycle can be found in reverse by following node.origin
28:            end if
29:        end for
30:    end for
31: end function
```

Algorithm 3 Locate cycles using binary search

```
1: procedure LOCATECYCLES(graph)
2:     max_weight ← −max(abs(graph.edges))  ▷ get the largest weight of an edge
3:     for all nodes in the graph node do
4:         maxval ← 0
5:         minval ← len(graph.nodes) · max_weight
6:         while minval ≤ maxval do    ▷ do binary search on possible minval
7:             possible_pmin ← (maxval − minval)/2
8:             cycle ←LOCATECYCLESINGLENODE(graph, node, possible_pmin)
9:             if a cycle is found then
10:                minval ← possible_pmin
11:            else
12:                maxval ← possible_pmin
13:            end if
14:        end while
15:        if cycle using node has been found then
16:            add cycle with highest pmin to list of cycles
17:        end if
18:    end for
19: end procedure
```

3.4 Examples

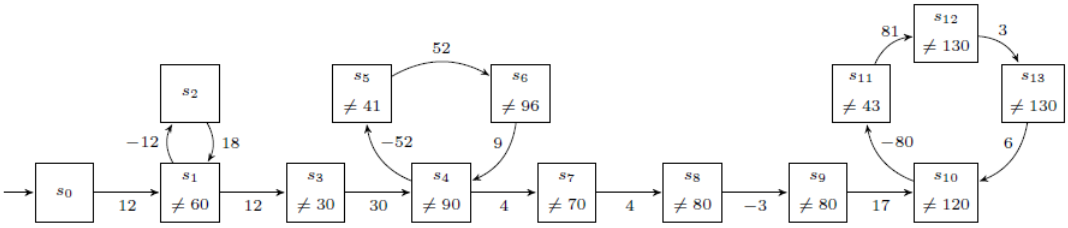

Figure 2: 1-VASS with 3 positive cycles

In the figure (Section 3.4), we can easily see the 3 cycles, and with some basic math, we can see that the first cycle (from left to right) is a positive cycle with weight 6, the second cycle is positive with a weight of 9 and the final cycle is positive as well with a weight of 10.

When running the algorithm, it will first locate the value $W = 81$ as the largest absolute weight of an edge. We try to locate the optimal cycle for s_0 first, but since this node is not part of a cycle, and no cycle occurs before it, this node never becomes a node causing the next update. As such, we never even

attempt to locate the cycle for any of the values for *pmin* we look for, and since with $pmin = -W \cdot |V|$, no cycle is found, we say that s_0 has no positive cycle. We try the same for s_1. After resetting the distance for each node and running the base Bellman-Ford algorithm (with negatives of their weights), we try to make s_1 the cause for any next update of distances in at most $|V|$ iterations of updating the value of all nodes, in this case (since s_1 is actually part of a cycle) this actually occurs and we can continue to the next step. We acquire all updated nodes from here, which is $(s_2, score = -12)$ and $(s_3, score = 12)$, since the -12 score is still above $-W \cdot |V|/2 (= 567)$, this update still occurs. Both s_2 and s_3 set s_1 as the node from where they got their optimal pmin. If we now try to look for new updateable nodes, we have found that both $(s_1, score = 6)$ and $(s_4, score = 42)$ have been found. s_1 set s_2 as the origin of its optimal score, and for s4, s3 is set as its best origin. Since s_1 is our starternode, we have located a cycle. We now try to locate our cycle through backtracking the optimal origin for each node starting from s_1. s_1's origin is s_2, and s_2's origin is s_1. We have now found our cycle (backwards),so we return it. Since our algorithm has returned a cycle, we try doing this for $-W \cdot |V|/2 + W \cdot |V|/4$ until eventually, we try for a pmin value above -12 and are unable to find a cycle. We then try to use a lower value again, and we eventually find our minimum and maximum pmin value to converge at -12. We continue doing this until we have have found the optimal positive cycles for each node.

3.5 Solving problems through examples

3.5.1 problem 1

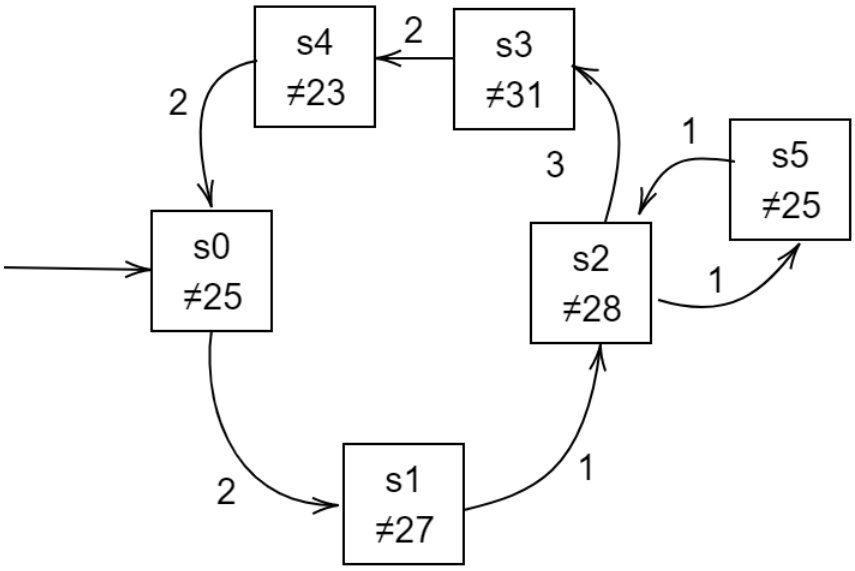

Figure 3: 1-VASS with 2 linked positive cycles

If we try to get the optimal cycle for s0 in this graph (Section 3.5.1), we run into an issue. We ignore the minimal pmin value since all edges have positive weights, so the pmin value of a cycle will always be 0. After running the bellman ford algorithm, and updating nodes until after $s0$ is the most recently updated. We update s1 and set its origin to s0, after that, $s2$ is updated with $s1$ as optimal origin. Now, both $s3$ and $s5$ update with $s2$ as its origin. the next iteration, $s4$ updates with $s3$ as origin and $s2$ updates with $s5$ as origin. finally, $s0$ updates with $s4$ as origin, and once again $s2$ updates with $s5$ as origin. Since $s0$ has been updated, we have officially located the cycle. as such, we now locate the cycle by backtracking the origins of the node. With $s0$, we have $s4$ as origin. with $s4$, we have $s3$,and $s3$ has $s2$. Now $s2$ has $s5$ and $s5$ has $s2$. which we can no longer backtrack. We will never reach our original node because we are stuck in an infinite loop. The problem here is that a smaller cycle has embedded itself into our original cycle during our search for the later parts of the cycle. Our current system no longer retains the information to return to s0. We can solve this issue by keeping the entire partial cycle instead of only the previous node. This way, we can check that we also do not add embedded cycles into our partial cycles. So if we now try the altered algorithm, we see that after doing

the initial bellman-ford and making $s0$ the last updated node. we can update $s1$ and set its partial cycle to $(s0)$, We update $s2$ and set its partial cycle as $(s0, s1)$. We update both $s5$ and $s3$ and their partial cycle is $(s0, s1, s2)$. We try updating $s2$ through $s5$, but due to the fact that $s2$ is already in the partial cycle of $s5$, this update does not happen. $s4$ is updated through $s3$ and gets the partial cycle $(s0, s1, s2, s3)$. $S0$ gets updated once more, with the "partial" cycle $(s0, s1, s2, s3, s4)$. Since the starter node is once again updated, we have located the cycle. With this new implementation, backtracking is no longer required, as the entire cycle is already located as the partial cycle of the starter node. We redo this until the minimal pmin value converges towards 0. We do the same for all other nodes, where similar results are received.

3.5.2 Example 2

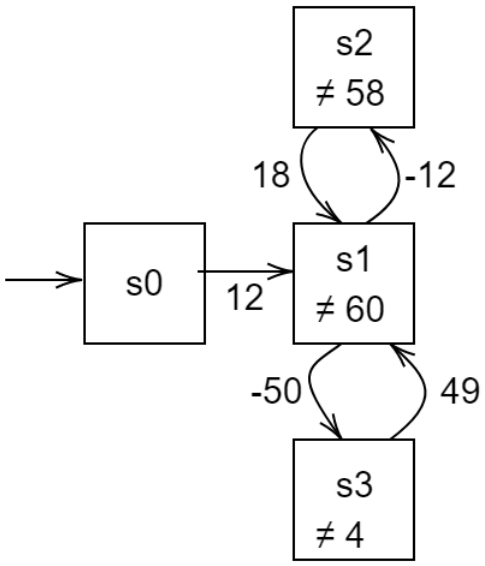

Figure 4: 1-VASS with 1 positive and 1 negative cycle

In the second figure here (section 3.5.2) we can see 2 cycles, the cycle containing node $s1$ and $s3$ is a negative cycle however. For $s0$, $s1$ and $s2$ we get the same results as when we did it previously with Example 1. When we try to detect a positive cycle containing $s3$. We start by running the bellman-ford algorithm as well as try to make $s3$ the node with the most recently updated distance. We do update the distance of $s3$ due to the influence of the neighbouring positive cycle. After that, we can update s1 with a partial cycle $(s3)$. The only node that is updated in the next iteration is $s2$, with $(s3, s1)$ as partial cycle. The

edge from $s1$ to $s3$ ends up with a lower score for bellman-ford, and thus, it does not cause an update. We attempt to update $s1$ once more, through the edge coming out of $s2$. However, with $s1$ already in the partial cycle reaching $s2$, the update does not go through. Since we can no longer update anything in the next iterations, we have not located any cycles. We try the same with a lower minimal pmin value, but regardless of the value, we do not find any positive cycle containing $s3$.

3.6 Rewrite function

We rewrite the function to locate a single node to store the entire origin instead of just the previous node.

Algorithm 4 Improved Locate cycle of a single node

```
1: function locateCycleSingleNode(graph, startnode, minimal_score)
2:     ResetBelmanFordDistances(graph)        ▷ reset the distances from previous runs (set startnode to 0 and everything else to ∞)
3:     for _ in graph.nodes do                ▷ run the basic Bellman-Ford algorithm
4:         singleBellmanFordUpdate(graph)
5:     end for
6:     current_value ← startnode.distance
7:     for _ in graph.nodes do
8:         singleBellmanFordUpdate(graph)
9:         if startnode.distance has updated then
10:            break        ▷ keep running BF until our startnode will cause the future updates
11:        end if
12:    end for
13:    if startnode.distance has not updated in second loop then
14:        return                    ▷ our startnode is not in any (positive) cycle
15:    end if
16:    found_nodes ← [startnode]
17:    for _ in graph.nodes do
18:        new_found_nodes ←empty set
19:        for node_from, node_to, weight in graph.edges do
20:            if node_from in found_nodes AND node_to.distance > node_from.distance - weight AND node_from.pmin + weight ≥ minimal_score AND node_to not in node_from.origin then
21:                node_to.distance ← node_from.distance − weight
22:                node_to.pmin ← node_from.pmin + weight
23:                node_to.origin ← node_from.origin + node_from
24:                new_found_nodes.add(node_to)
25:            end if
26:            if startnode.origin has been updated then    ▷ a cycle has been found
27:                return startnode.origin        ▷ cycle exists within node.origin
28:            end if
29:        end for
30:    end for
31: end function
```

3.7 Complexity of the algorithm

For each of our nodes, we reset the distances on every node to infinity, this takes $\mathcal{O}(|V|)$ Then, the algorithm does the actual Bellman-Ford algorithm, Which is $\mathcal{O}(|V| \cdot |E|)$. Afterwards, if any changes can still occur, we do at most the entire Bellman-Ford algorithm again to ensure our given node is the one that causes change, which once again is $\mathcal{O}(|V| \cdot |E|)$. Since we have our node, we update the

edges of the node up to at most $|V|$, since the largest possible cycle possesses at most each node. We do this $\mathcal{O}(log(W))$ times since we need to find the pmin value using the binary search algorithm. The total complexity for finding the optimal positive cycle for a node is $(\mathcal{O}(|V|) + (\mathcal{O}(|V| \cdot |E|) + \mathcal{O}(|V| \cdot |E|)) + \mathcal{O}(|V| \cdot |E|))) \cdot \mathcal{O}(log(W))$, which ends up as $\mathcal{O}(|V| \cdot |E| \cdot log(W))$, but we need to do this for each node so our complexity for the entire algorithm is

$$\mathcal{O}(|V|^2 \cdot |E| \cdot log(W))$$

3.8 Discussing Implementation

In this section, we will be discussing certain implementation details and their reasoning regarding the cycle location algorithm, as well as some of the basic objects used throughout the entire algorithm.

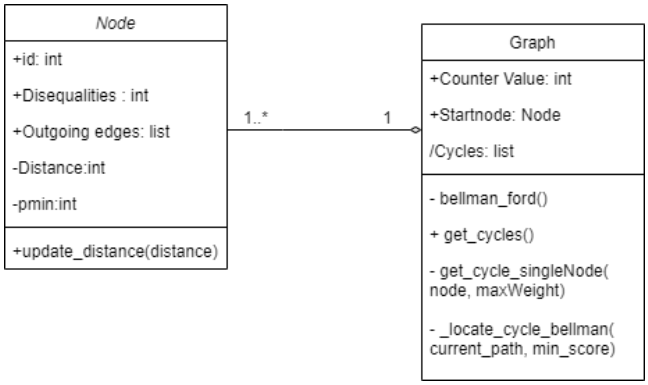

Figure 5: Class diagram of the classes and methods used to detect the positive cycles

As the class diagram (Section 3.8) states, our *graph* object stores a list of all *nodes*. The exact implementation of these objects can be found here: `https://github.com/QuentinDeHaes/Unboundedness-for-1-VASS/blob/master` within graph.py and node.py respectively. The edges of the *graph* are stored within the *nodes*, where each node keeps track of their outgoing edges. This makes little of an efficiency loss when we have to loop over all edges, as we would simply need to loop over all *nodes* and loop over their outgoing edges. This ends up with an identical amount of edges to pass over, all of them. The edges are simply split up over the various nodes. When we need the outgoing edges of a subset of the *nodes*, acquiring these edges is done more efficiently.

In the case of the *Bellman-Ford* algorithm, using the alteration where we search for the minimal distance using the negative weights compared to searching for the maximal distance using positive weights is identical. Both variations have the same efficiency so it merely falls on personal preference. The exact implementation of the algorithm can be found in graph.py from line 101 until line 112

All parts that use the Bellman-Ford can be found in a single internal method used for finding a cycle for a single given node along with a minimal allowed pmin value. The reason for this is that, originally, a method using Depth First Search (DFS) was developed to ensure the surrounding parts were correctly implemented as a DFS-method was more straight-forward to develop. The implemented DFS-method could not be used in the final implementation however, as it was not polynomial. The DFS implementation has since been removed from the project.

3.8.1 algorithm for detecting all optimal positive cycles and their pmin value

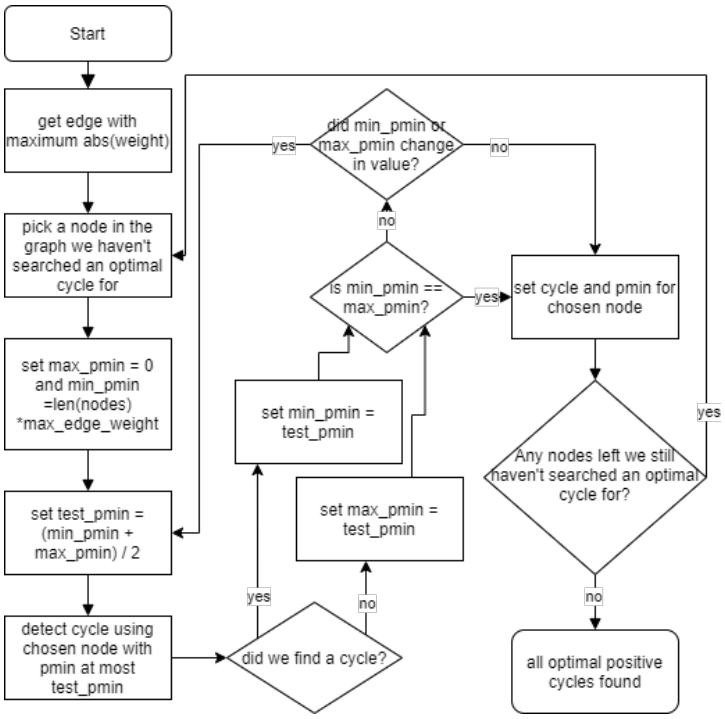

Figure 6: Flowchart of the algorithm for detecting cycles

The method used to detect the cycles is implemented in graph.py from line 114 until line 114. The method of binary search for the pmin value is implemented on line 143 until line 171. They are visualised in the flowchart above (section 3.8.1). We start by looking for W, next we pick a node to detect a cycle for. We set the edges of the binary search as upper bound 0 and lower bound $-|V| \cdot W$. We then use the middle value of these 2 bounds as pmin value. If we detect a cycle using our chosen node with a pmin value never going lower than the middle of our bounds, we make the middle our lower bound. Else

our upper bound gets updated. We then try locating the cycle again with the new middle of the bounds, and continue doing this until the 2 bounds have converged. When the 2 bounds have converged, if at any moment we actually located a cycle, we have detected the optimal cycle and its pmin value. We then pick a new node and reset the upper and lower bound to 0 and $-|V| \cdot W$ respectively and locate the cycle for this node in the same manner. We continue doing this until we have searched the optimal cycle for each node. Along with checking if the bounds are equal, the implementation also checks if any changes occured in either of the bounds. We do this because the loss of data in integer division could make the values never converge. An example would be, a cycle with actual pmin value of -4. Assume we did the algorithm until we got -4 as our lower bound and -3 as our upper bound. The middle value would be -3, and would update the upper bound to -3. This would cause the algorithm to never converge and run infinitely.

3.8.2 algorithm for detecting a singular optimal cycle given a startnode and a pmin value

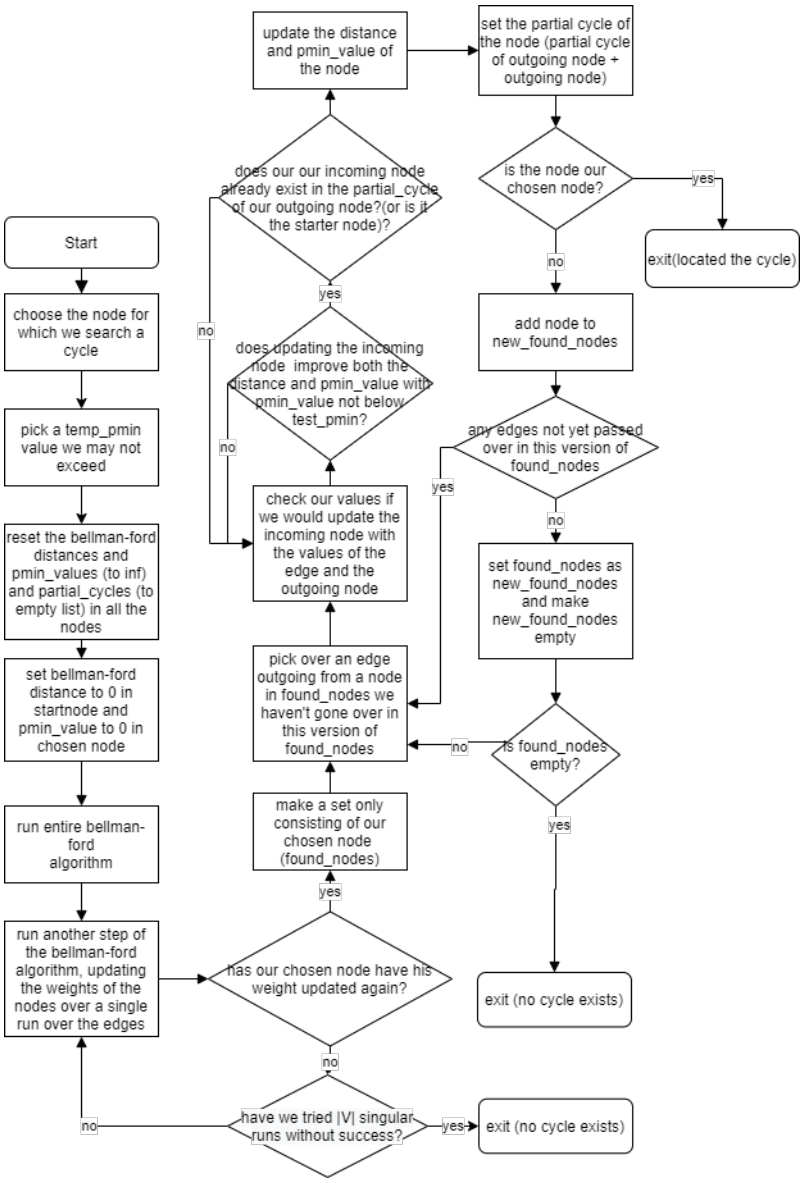

Figure 7: Flowchart of the algorithm for detecting a single cycle based on its startnode and a pmin value

The method to locate a cycle for a single node given a pmin value is implemented on line 173 until line 246. As seen in the flowchart (Section 3.8.2), this is one of the more complex methods in the program. We start by resetting the distance, pmin value and partial_cycle value in each node(distance and pmin value to ∞ and partial_cycle to empty list). Next we set the distance of our startnode to 0 and run the Bellman-Ford algorithm. We continue updating the distances of nodes until either we tried updating all nodes $|V|$ times without actually updating our chosen node, or our chosen node is just updated. We set the pmin value of our chosen node to 0. We place our chosen node into an empty list we will call found_nodes. We try to update the distance of all nodes that are outgoing of any node in found_nodes. In this stage however, we update the pmin value, the distance and the partial cycle. However, we only update if the distance and the pmin value improve, the pmin value does not exceed the allowed pmin value, and the node is not in the partial_cycle of the outgoing node. If the node updates, we add it to a new version of found_nodes. After we have updated all nodes based on the original found_nodes, we do this again with the new found_nodes until found_nodes is empty. An empty found_nodes means there is no cycle containing our chosen node not exceeding the given pmin value. If during any of our runs we update the pmin value of our chosen node, we detected a cycle with our chosen node never exceeding the given pmin value. We can return the found cycle.

4 Locating bounded chains (non-trivial q-residue classes)

In this section, we will locate bounded chains based on the positive cycles. We need to locate the values for each node in a positive cycle where we are unable to infinitely keep taking the cycle towards unboundedness, we call the set of configurations which can already reach unboundedness U_n. The reason we cannot continue taking the cycle in these configurations is either because a disequality guard is preventing us from taking the cycle at a certain point, or because taking the positive cycle means reaching a negative counter value at some point. So for each node in a positive cycle we need to acquire the values where we cannot take the cycle and the minimal value to take the optimal cycle. For each node, we need to calculate which values the optimal positive cycle cannot be taken with. This is relatively easily done by simply finding the change in the counter value taking edges between our original node, and each other node in the cycle, and subtracting that difference from the disequalities of the other node, along with keeping the lowest negative difference, i.e. the pmin value. The easiest way to do this is starting from the node and folowing the cycle keeping a counter initialised at 0, and calculating which value would violate the disequality guard for each node when we pass that node. Since we now have the values where we cannot take the cycle due to the disequalities and we have the absolute lowest value for taking the cycle, along with the weight of

the cycle. We can use these various values to generate these bounded chains. We cannot simply make a list from all values between a disequality acquired value and the minimal value with a difference of the weight between all values as this is not polynomial. As such, we use *Closures* to represent these lists, where we have a minimal value, a maximal value, and all values in between the two, which have the same residue modulo the cycles weight are assumed to be part of the closure without having to explicitly denote all these values.

4.1 Complexity

We need to go over all optimal positive cycles in the graph, So there is at most 1 different cycle per node, so $\mathcal{O}(|V|)$. For each cycle we need to run over each node in the cycle to acquire all necessary values. this is once again $\mathcal{O}(|V|)$ Since we need to loop over the nodes in the cycle , and no node can be in the cycle more than once. Since the amount of disequalities per node is polynomially bounded, we can be certain that the amount of closures generated is also polynomially bounded, as these values are directly based on the disequalities, and since these are currently just one per node, we can set it as $\mathcal{O}(1)$. so in total this method is

$$\mathcal{O}(|V|) \cdot \mathcal{O}(|V|^2) \cdot \mathcal{O}(1) = \mathcal{O}(|V|^3)$$

4.2 Algorithm

Algorithm 5 Locate bounded chains

```
1: procedure LOCATECHAINS(graph, cycles)
2:     for all nodes present in any cycle node do
3:         optimal_cycle ← node.optimal_cycle ▷ the cycle calculated for node in the previous algorithm
4:         non_cyclables ← NON_CYCLABLES(optimal_cycle, node) ▷ get the values where in node the cycle cannot be taken
5:         pmin ← PMIN(optimal_cycle, node) ▷ get the pmin value in node of the cycle
6:         weight ← GET_WEIGHT(optimal_cycle) ▷ get the weight of the cycle
7:         CLEAN_NON_CYCLABLES(non_cyclables) ▷ group values by their residue class and order them in size
8:         for all sorted values in Residue_classes in cleaned_non_cyclables val do
9:             Chain_min_value ← pmin + Residue_class mod weight
10:            GENERATE_CHAINS(Chain_min_value,weight, val )
11:        end for
12:    end for
13: end procedure
```

4.3 Examples

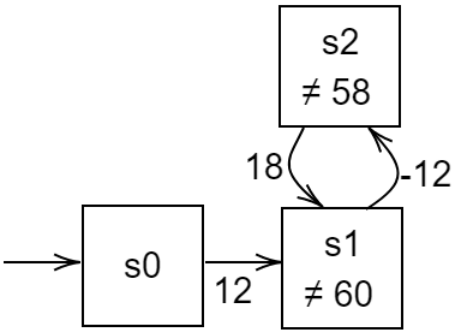

Figure 8: 1-VASS with a single positive cycle

As we can clearly see, the above figure (Section 4.3) we have only one cycle containing 2 nodes, and each node containing 1 disequaltiy guard. So for both of these nodes, the only positive cycle will obviously be the optimal cycle. So if we now manually calculate with which values we cannot take the cycle, we end up that in $s1$ we cannot take the cycle with 70 (would end up in $s2$ with 58) or with 54(would end up back in $s1$ with 60).Along with that, we cannot take the cycle with a value less than 12 (we would end up with a negative counter value in $s2$), and if we are in $s2$ we cannot take the entire cycle with 42(would end up in $s1$ with 60) or 52(would end up with 58 back in $s2$), however here we can take all values above 0, as we can never reach a value below 0 by taking the cycle. We can calculate the increase in counter value of taking the cycle once ,i.e. the weight of the cycle, is 6. With all this information, we can now easily calculate the closures of the bounded chains. We already have the maximum values (all the values where we cannot take the cycle) and the step size (cycle weight) of the different closures, so all we are missing is the minimum. We can calculate the minima using all other values we have, we simply add to the minimum value to take the cycle the residue of the maximum value modulo the weight. For $s1$, we have a closure with maximum value 70, step size 6 and minimum value 16 $(12 + 70 \mod 6)$, and a second closure with maximum value 54, step size 6 and minimum value 12 $(12 + 54 \mod 6)$. State $s2$ also has 2 non-trivial q-residue classes, one closure with maximum value 42, step size 6 and minimum value 0 $(0 + 42 \mod 6)$, and a second closure with maximum value 52, step size 6 and minimum value 4 $(0 + 52 \mod 6)$.

4.4 Implementation Details

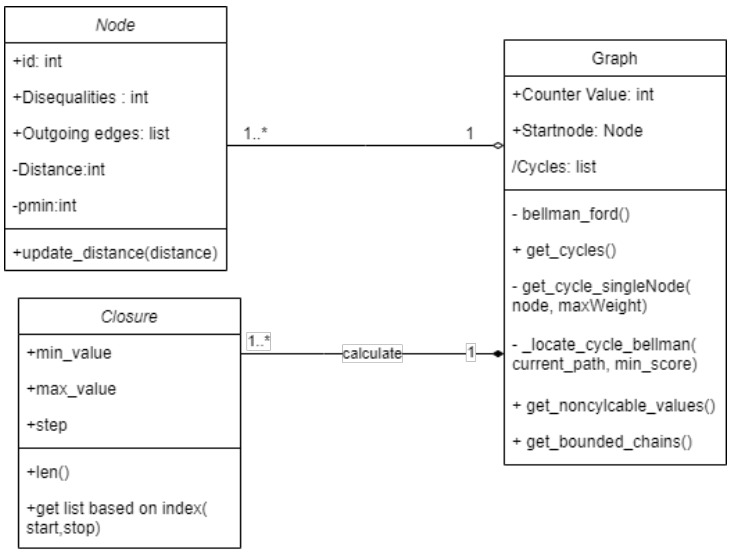

Figure 9: Class diagram of the classes and methods used up until the locating of the bounded chains

The class diagram(section 4.4) has a new object added compared to the previous class diagram(section 3.8). The *closure* object, implemented in Closure.py ,uses a minimum value, a maximum value, and a stepsize. It is extremely similar to the built-in range function in python. While using that may have resulted in better optimisation. The extra functionality and control added by using a self-made class was, at the time, thought to be useful. The ability to use $None$ as maximum value to denote an upward unbounded chain, while in retrospect not necessary to the algorithm, could be used in later parts to represent certain unbounded chains. Along with that, the *Graph* class has a few new methods added that are used to generate the closures.

4.4.1 algorithm for locating bounded chains

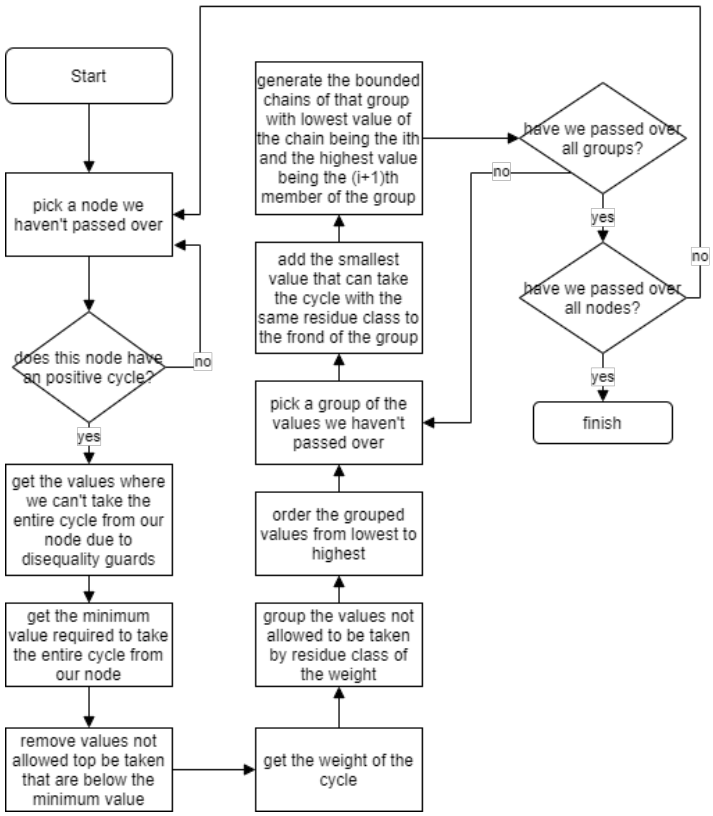

Figure 10: Flowchart of the algorithm for locating bounded chains

We generate the *Closures* in the method get_bounded_chains, which is implemented in graph.py from line 299 until line 327. As can be seen in the flowchart (Section 4.4.1), we start by picking the first node we are locating bounded chains for. Only nodes in a positive cycle can have bounded chains, so we skip any nodes not in a positive cycle. Get the pmin value of that cycle, and loop over all the nodes in the cycle starting from our chosen node to get the values where the disequality guards would block our node form taking the entire cycle. Remove the values lower than our pmin value from the untakeable values list. Group the values of the untakeable list by residue class modulo the weight of the cycle. Sort the values in each of these groups from smallest to largest. Next, we add the sum of the pmin value and the residue class modulo the weight to the front of each group. We create the bounded chains based on each group seperately. Each bounded chain in the group has minvalue: $value_i$, maxvalue: $value_{i+1}$ and a stepsize of weight for $i = 1, ..., n-1$. After having done this for each group

of residue class, we continue gathering the bounded chains for the next node. Until we gathered all bounded chains for all nodes.

5 Checking if the remaining configurations in the non-trivial q-residue classes are unbounded

After locating the bounded chains,we try to check whether any of the configurations in the bounded chains can still reach unboundedness. W check whether or not these configurations can reach a configuration that is already in U_n. But we cannot check each configuration in the bounded chains separately, as this amount is not polynomially bounded. But we can use a trick, when a larger value in the bounded chain can reach U_n, all the lower values in the chain could simply take the cycle until they reach this value, so if a configuration from a bounded chain reaches U_n, we could simply do a downward closure of all other configurations, as all these also reach U_n. Just trying the largest value in the chain is not good enough however, since this configuration could be prevented by the disequality guards from reaching U_n As such we need to try a polynomial amount of the highest values, but enough so that at least one configuration can bypass the disequality guards. We call this polynomial *top*. *top* can be represented as the following polynomial:

$$top(|V|) = 8|V|^7 + 10|V|^6 + 18|V|^5 + 24|V|^4 + 6|V|^3$$

.

Now that we know the amount of configurations that need to be checked is polynomial, we also need to make sure that the checking for each configuration whether they can reach U_n is polynomial. We do this in a rather peculiar manner, namely well divide U_n in a polynomial amount of sets of configurations, and proof that checking reachability for a configuration to that set can be done in polynomial time.

5.1 Bounded coverability with obstacles

We show that within a 1-VASS with disequality guards, we can decide if an initial configuration (s, z_0) can reach an objective of the form

$$O = \left\{ (t, z) \mid z \geq \ell \wedge \bigwedge_{i=1}^{m} (z \not\equiv a_i \pmod{W}) \wedge \bigwedge_{i=1}^{n} (z \neq b_i) \right\}$$

in polynomial time, where ℓ, W and the $(a_i)_{i=1}^m$ and $(b_i)_{i=1}^n$ are non-negative integers.

To begin with, we make sure that when a configuration can reach O using a path π such that $(q, z) \xrightarrow{\pi} (q', z')$ and $(q', z') \in U_n$, there exists a path π', $(q, z) \xrightarrow{\pi'} (q', z'')$ such that $(q', z'') \in U_n$ and π' has a length of at most $L(|V|)$, where $L(|V|)$ is a polynomial. With that in mind, we can already ensure that

we need to do at most $L(|V|)$ passes of the algorithm before we can stop, with $L(|V|)$ being presented as $(8|V|^8+10|V|^7+18|V|^6+24|V|^5+6|V|^4+|V|^2+|V|+3)^2 \cdot |V|+|V|^2+3$, which as a complete expanded form polynomial amounts to:

$$L(|V|) = 64|V|^{17}+160|V|^{16}+388|V|^{15}+744|V|^{14}+900|V|^{13}+984|V|^{12}+808|V|^{11}+324|V|^{10}$$

$$+140|V|^9+144|V|^8+168|V|^7+156|V|^6+37|V|^5+2V|^|4+7|V|^3+7|V|^2+9|V|+3$$

, which means $L(|V|) = \mathcal{O}(|V|^{17})$, an incredibly large but still polynomial amount.

We need to keep the configurations reachable from (q, z) after n passes, which would normally be bounded by E^n configurations with $n \leq L(|V|)$, which unfortunately is not polynomial. As such we need to prune the maximum amount of configurations after each pass to a polynomial amount. First, we delete from the set of configurations (q, z) such that there are $(n + L(|V|))$ configurations (q, z') in the set of configurations with $z' > z$ and $z' \equiv z(\mod W)$. Secondly, we delete from the set of configurations all configurations (q, z) such that there are $(n + L(|V|)) \cdot (m + 1)$ configurations (q, z') with $z' > z$. m being the total amount of a_i in our objective and n being the total amount of b_i given in our objective. Clearly the cardinality after each run is at most $(n+L(|V|))\cdot(m+1)$, and it can be computed from the set of the previous run in polynomial time.

5.2 Dividing U_n into objectives

For each node in a positive cycle, we have 2 different types of objectives: all in the form of

$$O = \left\{ (t, z) \mid z \geq \ell \wedge \bigwedge_{i=1}^{m} (z \not\equiv a_i \pmod{W}) \wedge \bigwedge_{i=1}^{n} (z \neq b_i) \right\}$$

1. A single objective (per node) for all configurations in trivial q-residue classes:
   - ℓ is the minimal value needed to take the cycle
   - W is the weight of the cycle
   - $(a_i)_{i=1}^m$ is a list for the residues of all non-trivial q-residue classes
   - $(b_i)_{i=1}^n$ is an empty list here.
2. As single objective for each non-trivial residue class R_j:
   - ℓ is the smallest value that is in both R_j and U_n, in U_0 this will be the value directly above the bounded chain, but as more values from within the bounded chain get added to U_n this value becomes lower
   - W is still the weight of the cycle from which R_j originates
   - $(a_i)_{i=1}^m$ is a list for the residues of all non-trivial q-residue classes, except the residue from R_j itself.

- $(b_i)_{i=1}^n$ is the list of all values in the bounded chain we skipped over and are thus not yet in U_n, in U_0 this list is empty, but in the case the k highest values in the chain cannot (yet) reach U_n, but the next value can, all k values are placed in the list $(b_i)_{i=1}^n$.

5.3 Algorithm

Algorithm 6 Bounded Coverability with obstacles

```
1: function BNDCOVERWOBSTACLES(q', z', U_n)
2:     for all objective_equationsets(target, O) do
3:         let reachable be a list initially containing (q', z')
4:         for _ in L do
5:             let new_reachable be an empty list
6:             for all Configurations in reachable cf do
7:                 add all configurations reachable from cf using a single edge to
   new_reachable
8:             end for
9:             if Any of new_reachable is in (target, O) then
10:                return True
11:            end if
12:            reachable ← PRUNE(new_reachable)
13:        end for
14:    end for
15:    return False
16: end function
17: procedure UNBOUNDED(graph, q, z)
18:    let U_0 be the unbounded chains          ▷ acquired using bounded chains
19:    let M be an empty list
20:    repeat
21:        for all bounded chains C do
22:            for all (q', z') ∈ top(C) do
23:                canReach ← BNDCOVERWOBSTACLES(q', z', U_n)
24:                if canReach then
25:                    Add (C, q', z') to M
26:                end if
27:            end for
28:        end for
29:        Ũ_{n+1} ← downward closure of a chains C in M from (q', z')
30:        U_{n+1} ← U_n ∪ Ũ_{n+1}; n ← n + 1
31:    until U_n = U_{n−1} return BNDCOVERWOBSTACLES(q, z, U_n)
32: end procedure
```

5.4 Complexity

We need to ensure that there is only a polynomial amount of items to check reachability for, a polynomial amount of objectives, and checking reachability for each objective can be decided in polynomial time. The amount of Items to check reachability for is obviously polynomial, as we check only $top(|V|)$ items per bounded chain, and there are only a polynomial amount of bounded chains. There are at most $\mathcal{O}(|V|)$ bounded chains per node, so there are at most $\mathcal{O}(|V|^2)$ bounded chains in the graph. The amount of objectives is also bounded by the amount of bounded chains, as such bounded by $\mathcal{O}(|V|^2)$. Checking the reachability to one of these objectives is also polynomial, due to our construction of the algorihtm. While checking reachability is normally of length $\mathcal{O}(|E|^n)$ with n the distance reached. We can state that the distance can at most be $L(|V|)$, which is a polynomial distance. however $\mathcal{O}(|E|^{L(|V|)})$ is not polynomial. As such, we chose not be doing this directly. We prune the possible configurations to check reachability from intelligently, until we eventually have at most $(n + L(|V|)) \cdot (m + 1)$ configurations after finishing each step, a polynomial amount. This way, we ensure this method is polynomially bounded.This part is thus bounded by $\mathcal{O}(L(|V|) \cdot (n + L(|V|)) \cdot (m + 1) \cdot |E|)$ The entire algorithm is thus polynomialy bounded by

$$\mathcal{O}(top(|V|) \cdot |V|^4) \cdot L(|V|) \cdot (n + L(|V|)) \cdot (m + 1) \cdot |E|)$$

5.5 Example

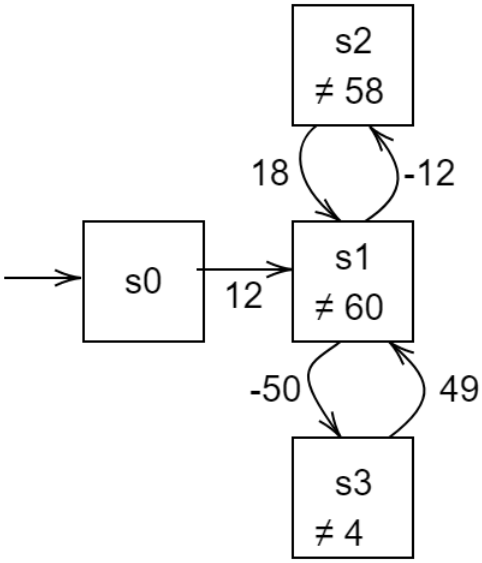

Figure 11: 1-VASS with a positive and a negative cycle

The above figure section 5.5 has 1 positive cycle containing $s1$ and $s2$, with a weight of 6. With this, we can determine that $s1$ has a bounded chain with minimum 0, maximum 54, and stepsize 6, which will be written as $(0, 54, 0)$. It also has the chain $(4, 70, 6)$. $s2$ has the chains $(0, 42, 6)$ and $(4, 52, 6)$. Both $L(|V|)$ and $top(|V|)$ are incredibly large numbers even with this small graph, we never reach these values throughout this example, as such, we continue along the examples without continuously checking whether we have surpassed the immense values each time as we will never exceed them here. We generate the objective-sets using this information. At the initial phase, the objective sets for s1 are:

$O = \{(s1, z) \mid z \geq 12 \wedge (z \not\equiv 0 \pmod 6) \wedge (z \not\equiv 4 \pmod 6)\}$
$O = \{(s1, z) \mid z \geq 76 \wedge (z \not\equiv 0 \pmod 6))\}$
$O = \{(s1, z) \mid z \geq 60 \wedge (z \not\equiv 4 \pmod 6))\}$
and the objective sets for s2 are:
$O = \{(s2, z) \mid z \geq 0 \wedge (z \not\equiv 0 \pmod 6) \wedge (z \not\equiv 4 \pmod 6)\}$
$O = \{(s2, z) \mid z \geq 48 \wedge (z \not\equiv 4 \pmod 6))\}$
$O = \{(s2, z) \mid z \geq 58 \wedge (z \not\equiv 0 \pmod 6))\}$

Now we see whether we can reach any of these objectives from the values of our bounded chains. We start by checking whether $(s1, 54)$ can reach any one of the objectives. In the implementation, we check if it can reach any objective

separately due to the pruning requirement, but in this case we check for all objectives simultaneously to make the example simpler. At $(s1, 54)$ we can go to $(s3, 4)$ and $(s2, 42)$. However, $(s3, 4)$ violates its disequality guard, as such, we remove this as a possibility. $(s2, 42)$ is not part of any objective, so we continue from here to see if we can reach any objectives. $(s2, 42)$ can reach $(s1, 60)$ which once again violates a disequality guard. So this configuration can currently not reach any objective. We should of course try more than one value from the bounded chain, but as we quickly notice, taking the edge to $s3$ with any of the other values in the bounded chains would result in a negative counter value, and taking the edge to s2 would result in taking the cycle once, ending in a configuration we already tried and which failed to reach an objective. $(s1, 70)$ can take the edge to $s3$ to end with $(s3, 20)$ while taking the edge to s2 results in $(s2, 58)$ which violates a disequality guard. from $(s3, 20)$, we reach $(s1, 69)$ which is part of an objective. As such, we update the corresponding objective to include the configuration $(s2, 70)$ along with its downward closure in the bounded chain.

$O = \{(s1, z) \mid z \geq 76 \wedge (z \not\equiv 0 \pmod 6))\} -> \{(s1, z) \mid z \geq \mathbf{4} \wedge (z \not\equiv 0 \pmod 6))\}$

Now, we look whether the next chain can reach any objective. We check $(s2, 42)$ next. The only edge we can taken takes us to $(s1, 60)$, which violates a disequality guard. Next, we try for (s2, 36). From $(s2, 36)$, we can only end up in $(s1, 54)$. Which you can remember from before, could not reach any objective. While the objective has changed since then, any of the values reached previously were not added by the update to the objectives. Continuing down the chain simply fails to reach an objective due to our inability to take the negative cycle at any point in time. Taking the positive cycle would simply result in a previously checked configuration (from which we would thus continue to fail)

The final chain to check in the first iteration starts with the configuration $(s2, 52)$. from here, we can only take the edge to (s1, 64). From there, we can go to $(s3, 14)$ and $(s2, 58)$. $(s2, 58)$ violates a disequality guard, so we discard that configuration. $(s3, 14)$ can reach $(s1, 63)$ which is part of the objectives. as such, once again, the entire chain becomes part of the objective. changing the objective to:

$\{(s2, z) \mid z \geq 58 \wedge (z \not\equiv 0 \pmod 6))\} -> \{(s2, z) \mid z \geq \mathbf{4} \wedge (z \not\equiv 0 \pmod 6))\}$

With this, the first iteration is done, but since the objective has changed during the iteration, a second iteration needs to be done on the remaining configurations in bounded chains.

We once again start with checking $(s1, 54)$. From here we can reach $(s3, 4)$ and $(s2, 42)$. $(s3, 4)$ validated the disequality guard still. and (s2,42) is not part of the objective. from $(s2, 42)$ we can only reach $(s1, 60)$, which is once again the disequality guard violated. The lower values in the bounded chain also react identical to the time we check, meaning we once again do not reach any objective with any value of this chain. Trying $(s2, 42)$ once more still results in reaching $(s1, 60)$ so trying $(s1, 36)$ results in reaching $(s1, 54)$, which we just saw cannot reach any of the objectives. The values in the chain below 36 also act the same with $(s2, 30)$ reaching $(s1, 48)$; $(s2, 24)$ reaching $(s1, 42)$; etc. In this iteration, the objectives did not change, so we have reached the complete

version of U_n

To finish the algorithm, the only thing we need to do now is check whether $(s0, 0)$ can reach any of the objectives. From $(s0, 0)$ we can reach $(s1, 12)$ which unfortunately is not part of any objective, more specifically, it is part of the remaining bounded chains that cannot reach unboundedness. We continue the algorithm here to try anyway as that is what the implementation would do. We can only reach $(s2, 0)$ as going to s3 would result in a negative counter value. and from $(s2, 0)$ we reach $(s1, 18)$. We continue to take the positive cycle to up to $(s1, 54)$ which is a value that can take the edge to $s3$, but would there violate the disequalty guard. Taking the cycle once more results in the configuration $(s1, 60)$, once again violating a disequality guard. Due to that, we can say that $(s0, 0)$ cannot reach unboundedness, and this 1-VASS is not unbounded.

5.6 Implementation details

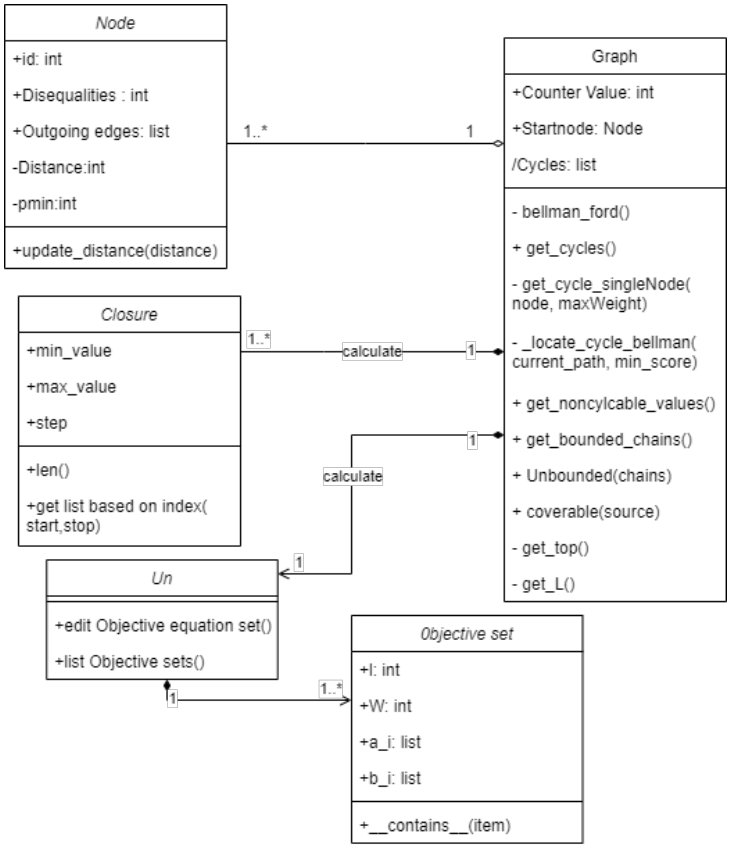

Figure 12: Class diagram of the complete implementation

There are 2 new classes in our class diagram(Section 5.6 compared to the previous diagram(section 4.4). The Un class and the Objective set class. The Un class is simply a collection of objective sets. Objective sets are special manner to store infinite values that work as described previously in section 5.2. New methods are added to the graph class that are used in generating Un and its objective sets.

5.6.1 boundedCoverWObstacles implementation

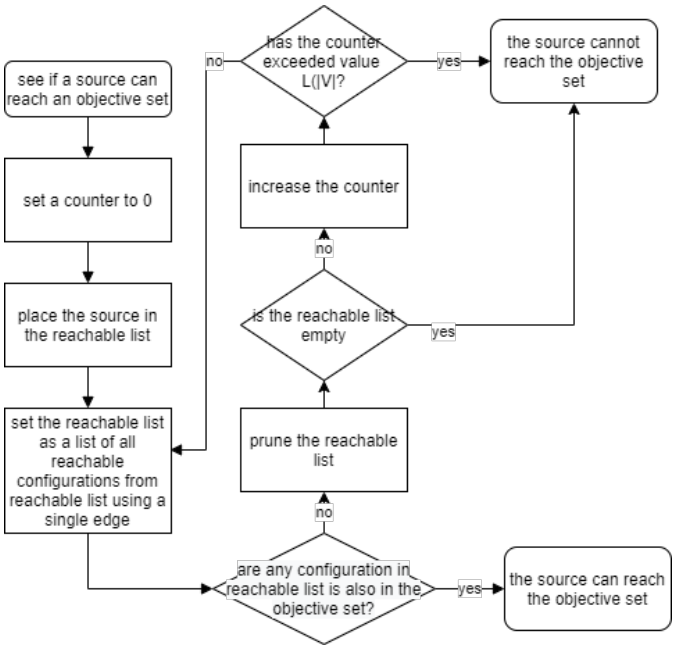

Figure 13: Flowchart of the the boundedCoverWObstacles algorithm

The *bndCoverWObstacles* function is made as a standalone function used within the *unbounded* procedure because the function itself can also be used after U_n has been fully calculated to compute whether any given configuration can achieve unboundedness. The implementation of *bndCoverWObstacles* can be found in Bounded_coverability_with_obstacles.py from line 97 until line 121. As visible in the flowchart (Section 5.6.1), we start by setting our initial source in the reachable list. Next we set a loop to run for at most $L(|V|)$ steps. Within that loop, we gather all configurations reachable (and allowed by disequality guards) following a single edge from any configuration in our reachable list. If any of these configurations is also within our objective set, we can say the source can reach is the objective set and end the algorithm. Otherwise we prune the list of reached configurations. In case that pruned list is empty, we can stop the algorithm and say the source cannot reach the objective set. Else we set the

new configurations as the reachable list and start within the loop again. If we loop $L(|V|)$ times without reaching any configuration in the objective set, we can end the algorithm and conclude the source cannot reach the objective set.

5.6.2 unbounded implementation

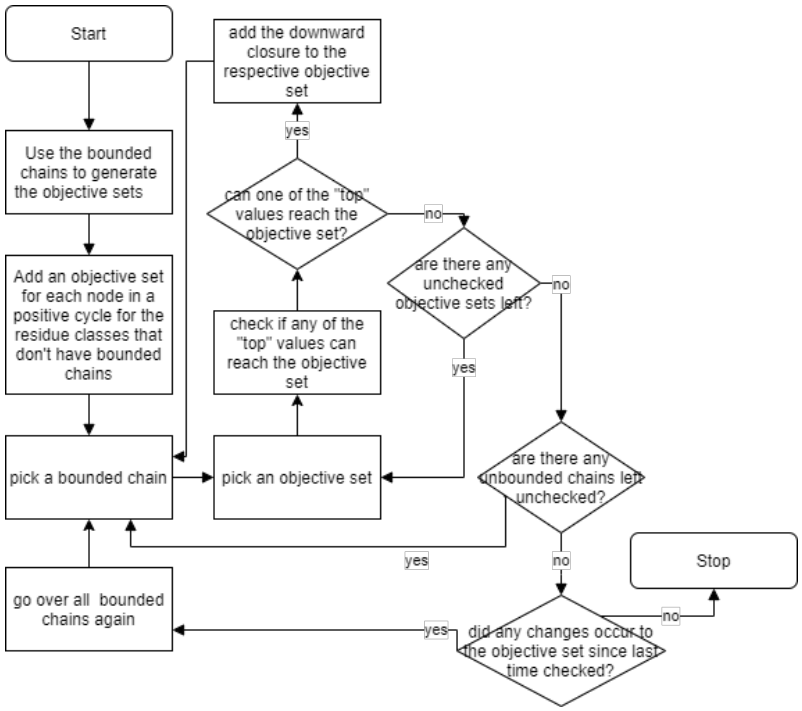

Figure 14: Flowchart of the unbounded algorithm

The *unbounded* procedure is implemented in graph.py from line 361 until line 422. The flowchart (Section 5.6.2) describes that, in order to start our procedure, we turn our *closures* into U_n, a group of *Objective sets*. Next, start a loop over all bounded chains. For each of the $top(|V|)$ highest configurations in the chain (from highest to lowest), we go over all objective sets, and check if the configuration can reach that objective set. If it can reach the objective set, we no longer need to go over the other objective sets, or any smaller configurations in the chain. All smaller configurations can reach the same objective by simply initially taking the cycle a few times. The configuration in the closure that can reach U_n and all smaller configurations (its downward closure) can be added to U_n. These configurations can also be removed from the bounded chains. After looping over all bounded chains, we check if any changes occured to U_n. If U_n did change, we restart the loop over all (remaining) chains. If U_n remained the same, we end the algorithm and have located the complete U_n

5.6.3 generating U_0

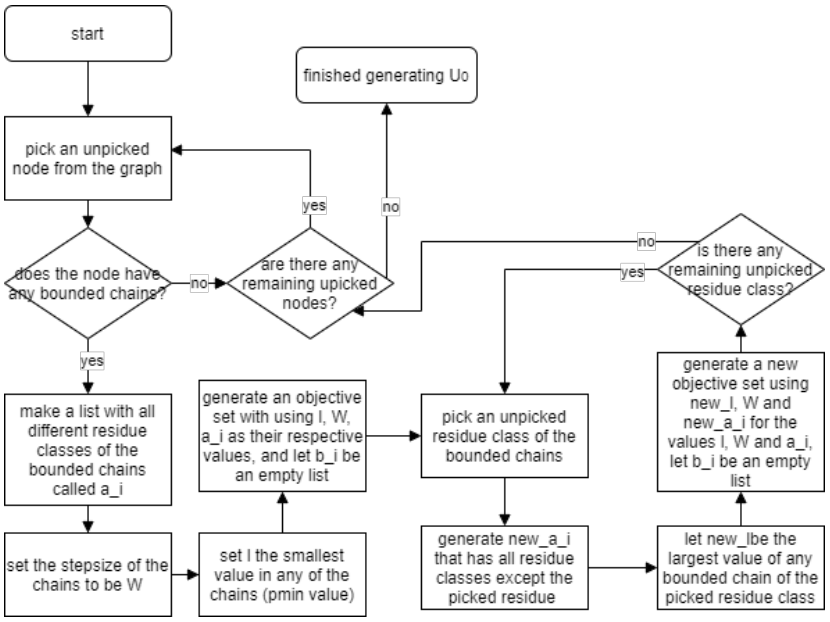

Figure 15: Flowchart turning closures into objective sets

The implementation of the algorithm that turns *closures* into $U_n$can be found in Un_representation.py. As we can see in the flowchart (section 5.6.3), we start by looping over all nodes that have at least 1 bounded chain. Within that loop, we start by making a list of all residue classes of the bounded chains of that node. We use that list as $(a_i)_{i=1}^m$, the stepsize of the chains will be used as W, and the smallest value of any of the chains as ℓ to make our first objective set, the objective set for the trivial q-residue classes. We set $(b_i)_{i=1}^n$ of the objective set as an empty list. Next we loop over all different residue classes of the chosen node's bounded chains. For each residue class, we generate a new objective set with $(b_i)_{i=1}^n$ an empty list and W the stepsize of the chains, $(a_i)_{i=1}^m$ a list of all other residue classes, and ℓ the highest value of any bounded chain of the residue class. After we have done this for every node (with bounded chains) we generated the complete U_0.

A config file is added to determine if we are in a test environment. When we are in a test environment, both the polynomial $top(|V|)$ and the polynomial $L(|V|)$ are replaced with smaller constant values. This is done to ensure the tests run in a timely manner, as while both are polynomial, they are still incredibly large. And with the simple test cases, the large polynomial is not necessary.

6 Conclusion

We have shown that the algorithms proposed in [1] can exist outside of the theoretical space. The algorithms can be implemented and maintain their polynomial time. the classical argument of Rackoff[10] easily generalises to show that control-state reachability remains in EXPSPACE, but to the best of our knowledge, no algorithm or implementation of exist as of yet.

References

[1] Shaull Almagor, Nathann Cohen, Guillermo A. Pérez, Mahsa Shirmohammadi, and James Worrell. Coverability in 1-vass with disequality tests. In Igor Konnov and Laura Kovács, editors, *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference)*, volume 171 of *LIPIcs*, pages 38:1–38:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[2] Benedikt Bollig, Karin Quaas, and Arnaud Sangnier. The Complexity of Flat Freeze LTL. In Roland Meyer and Uwe Nestmann, editors, *28th International Conference on Concurrency Theory (CONCUR 2017)*, volume 85 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 33:1–33:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[3] Daniel Bundala and Joel Ouaknine. On parametric timed automata and one-counter machines. *Information and Computation*, 253:272–303, 2017. GandALF 2014.

[4] Stéphane Demri, Ranko Lazić, and Arnaud Sangnier. Model checking memoryful linear-time logics over one-counter automata. *Theoretical Computer Science*, 411(22):2298–2316, 2010.

[5] John Fearnley and Marcin Jurdziński. Reachability in two-clock timed automata is pspace-complete. *Information and Computation*, 243:26–36, Aug 2015.

[6] Alain Finkel, Stefan Göller, and Christoph Haase. Reachability in register machines with polynomial updates.

[7] Stefan Göller, Christoph Haase, Joël Ouaknine, and James Worrell. Model checking succinct and parametric one-counter automata. In *In Proc. of ICALP (2), volume 6199 of LNCS*, pages 575–586. Springer, 2010.

[8] Christoph Haase, Stephan Kreutzer, Joël Ouaknine, and James Worrell. Reachability in succinct and parametric one-counter automata. In Mario Bravetti and Gianluigi Zavattaro, editors, *CONCUR 2009 - Concurrency Theory*, pages 369–383, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[9] Pascal Lafourcade, Denis Lugiez, and Ralf Treinen. Intruder deduction for ac-like equational theories with homomorphisms. In Jürgen Giesl, editor, *Term Rewriting and Applications*, pages 308–322, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[10] Charles Rackoff. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science*, 6(2):223–231, 1978.