1-VASS with disequality guards

Quentin De Haes

April 26, 2021

This report is for a research project of the university of Antwerp. We will be discussing the fact that deciding unboundedness of 1-vector addition system with states (1-VASS) with disequality guards is polynomial, and mainly provide and discuss it's 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 Defintions

1. **finite state automata (FSA)**: a collection of a finite number of states, connected by various edges. We have an initial state, and from there, we can take an edge that goes from this state to another to end up in the other state.

2. **A 1-vector addition system with states (1-VASS)**: an FSA with a single counter-value initialised on 0, along with that, each edge gets assigned a weight, and taking said edge will add this weight to the counter-value. The counter-value is not allowed to go into negative values, as such, no edge may be taken that would result in the counter-value ending up with a negative value.

3. **disequality guards**: values each state in the 1-VASS can possess which the counter-value may not be at when being in that specific state

4. **unboundedness**: having an infinite amount of different state, counter-value pairs that can be reached from the initial state in a 1-VASS.

5. **cycle**: a enumeration of edges in a 1-VASS where, after taking all enumerated edges, we would end up back in the state where we originally started the cycle

6. **positive cycle**: a cycle, where after taking all edges in the cycle and returning to the original state the counter-value of the 1-VASS has increased in value

7. **pmin-value**: the smallest value we would get as counter-value at any time when taking a positive cycle with an initial counter-value of 0, ignoring the fact negative values are not allowed in a 1-VASS.

8. $|V|$: the total amount of nodes in the 1-VASS, this symbol will be used to calculate the complexity of the various steps in the algorithm

9. $|E|$: the total amount of edges in the 1-VASS, this symbol will be used to calculate the complexity of the various steps in the algorithm

2 Introduction

A Vector addition system with states has some interesting applications, one of these is its similarity to petri-nets. Which means this research could be used to create algorithms to solve non-elementary problems for petri-nets as well. We will be deciding unboundedness for a 1-vector addition system with states (1-VASS) with disequality guards. In order to achieve a polynomial efficiency, we assume there exists a maximum amount of disequaltity guards a single state can possess. We can safely make this assumption as the diseqality guards of a state need to be listed, this means they can't be infinite. In the code each node will have a single disequality guard for simplicity. The proofs that validate the various techniques used to ensure polynomiallity will not be provided within this paper as they have been previously discussed [2, 1] and are not my contribution to the research.

3 Cycle detection

The algorithm requires positive cycles to work, since we could continue taking the cycle infinitely to achieve unboundedness. This means we need to locate these cycles, but we need to make sure this is also done in polynomial time. We don't need to locate all positive cycles, we just need to look for the optimal positive cycle for each node. If a node is not in any positive cycle, it won't have an optimal positve cycle of course. Deciding which the optimal positive cycle is for each node is rather easy, as the optimal cycle would be the cycle with the highest pmin value, or lowest absolute pmin value, which would amount to getting the smallest value required take any positive cycle from the node.

3.1 acquiring the optimal pmin-value for each node

We cannot simply try a plethora of pmin-values as that would not be polynomial, but we could look 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. And since a cycle cannot have a node occurring twice, we know that the largest amount of nodes in a cycle would be all nodes of the 1-VASS thus $|V|$. The pmin-value would thus be bounded by $-W * |V|$ and 0 as 0 is the highest pmin-value we

can get due to starting the calculation of the pmin-value of a cycle in a node with the counter-value initialised at 0. Trying all these bounded pmin-values is still not polynomial due to the fact that W is an exponential value , due to it being a number and adding a bit to the size of a number doubles the amount of representable numbers. We can however negate said exponential complexity by using a logarithmic algorithm to locate 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 no positive cycle incorporating the node exists.

3.2 locating the cycle using a prospective pmin-value

for locating the cycle we are using a variation of the Ford-Fulkerson algorithm

3.2.1 Ford-Fulkerson algorithm

The Ford-Fulkerson 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. Initially, each node is given a distance of $+\infty$, except for the start-node, this one is given a distance of 0. 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 a nodes cost + the edges cost is smaller than the cost of the newly reached node, then we edit the cost of the newly reached node to the smaller value. A single pass over all edges is clearly of efficiency O(E). But because the order of edges we pass over is random, with a single pass, we'd only have all nodes with correct distance in the best case, if all edges are exactly how we'd want them. In the worst case, only a single extra node is given it's correct distance. With this knowledge however, 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, all will be minimal. The total efficiency is thus O(V*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. When 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 Ford-Fulkerson algorithm

The first issue we face is that the Ford-Fulkerson algorithm locates negative cycles, while our algorithm requires positive cycles. This can be fixed easily by

simply negating all weights assigned to the edges, this way 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 Ford-Fulkerson 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 is a 'ripple' caused by a cycle. So with this knowledge can be used to continue updating the cost of nodes until the node for which we're 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, so knowing that a cycle has at most V nodes in it, due to a node not being able to occur twice in the same cycle. 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've located a cycle, we return this successful cycle along with it's 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 Examples

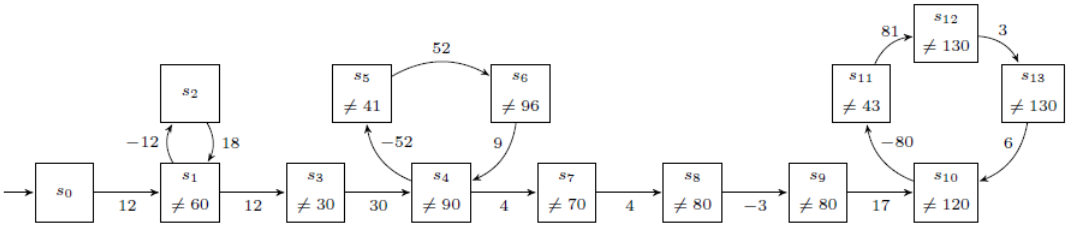

Figure 1: 1-VASS with 3 positive cycles

in the picture above (3.3), 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 as the largest absolute weight of an edge. W is 81 in our case. We try to locate the optimal cycle for s0 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*V, no cycle is found, we say that s0 has no positive cycle. We try the same for s1. After resetting the distance for each node and running the base Ford-Fulkerson algorithm (with negatives of their weights), we try to make s1 the cause for any next update of distances in at most V iterations of updating the value of all nodes, in this case (since s1 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 (s2, score=-12) and (s3, score=12), since the -12 score is still above $-W * V/2 (= 567)$, this update still occurs. And both s2 and s3 set s1 as the node from where they got their optimal pmin. If we now try to look for new update-able nodes, we have found that both (s1, score=6) and (s4, score=42) have been found. For s1, s2 is set as the origin of its optimal score, and for s4, s3 is set as its best origin. Since s1 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 s1. s1's origin is s2, and s2's origin is s1. since we have now found our cycle(backwards), we have it be returned. So now, since this has returned a cycle, we try doing this for $-W * V/2 + W * V/4$ until eventually, we try for a pmin value above -12 and are unable to find a cycle, then we try to do a lower one again, and we find it for pmin -12 as both the minimum and maximum value for pmin. We continue doing this until we have have found the optimal positive cycles for each node.

3.4 solving problems through examples

3.4.1 problem 1

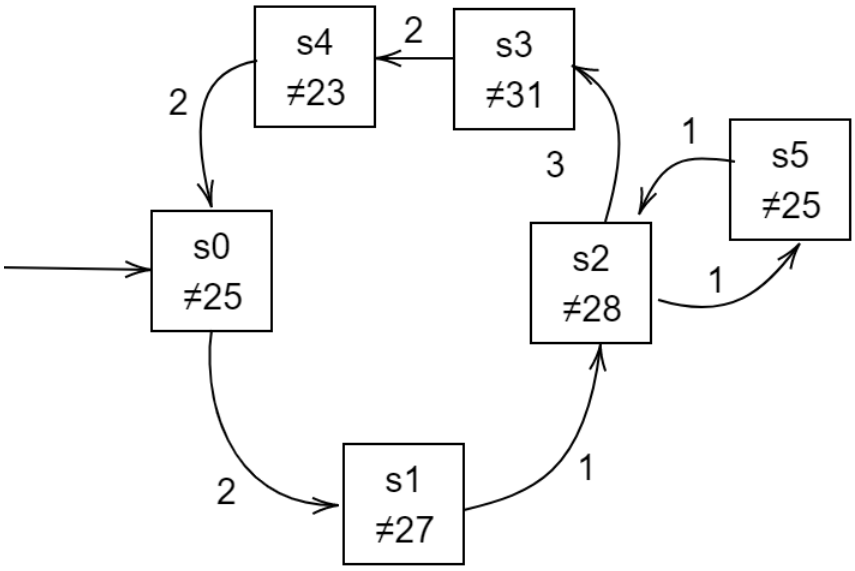

Figure 2: 1-VASS with 2 linked positive cycles

If we try to get the optimal cycle for s0 in this graph, we run into an issue. We ignore the minimal pmin, since all edges have positive weights, so the pmin 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 since we will never reach our original node due to 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, and our current system no longer has 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 don't 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 has once again updated, the cycle has been located. However, with the current 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.

3.4.2 Example 2

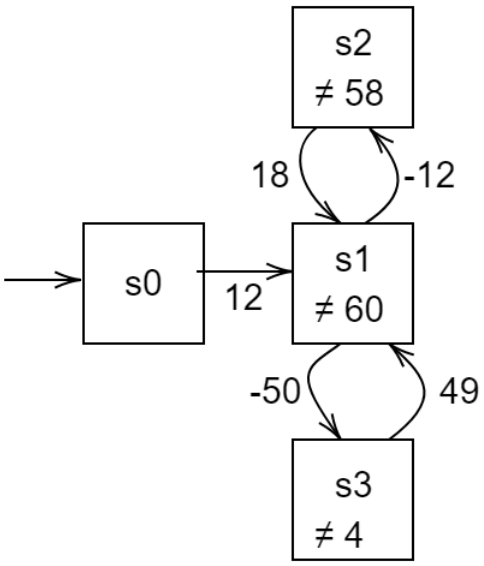

Figure 3: 1-VASS with 1 positive and 1 negative cycle

is the second picture here (3.4.2) we can see 2 cycles, the cycle (s1,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. For s3, however,when try to locate a positive cycle for s3. We start by running the bellman-ford algorithm as well as try to make s3 the last updated node. Something which occurs due to the influence of the neighbouring positive cycle. 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 end up with a lower score for bellman-ford, and

thus not cause an update. We attempt to update s1 once more, through the edge of s2. However, with s1 already in the partial cycle reaching s2, the update 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 including s3.

3.5 complexity of the algorithm

For each of our nodes, we reset the distances on every node to infinity, this takes O(V) Then, the algorithm does the actual Ford-Fulkerson algorithm, Which is O(V*E). Afterwards, if any changes can still occur, we do at most the entire Ford-Fulkerson algorithm again to ensure our given node is the one that causes change, which once again is O(V*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 for O(log(W)) times since we need to find pmin using the binary search algorithm. the total complexity for finding the optimal positive cycle for a node is $(O(V)+(O(V*E)+O(V*E))+O(V*E)))*O(log(W))$, which ends up with O(V*E*log(W)), but we need to do this for each node so our complexity for the entire algorithm is $O(V * E * log(W))$

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

We need to locate the values for each node in a positive cycle where we can't infinitely keep taking the cycle towards unboundedness, we call the set of configurations which can already reach unboundedness U_n. The reason we can't 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 can't 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 can't 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, making it's positive the minimal value needed to take the cycle. 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 passing that node. Since we now have the values where we can't 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 $O(V)$. For each cycle we need to run over each node in the cycle to acquire all necessary values. this is once again $O(V)$ 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 $O(1)$. so in total this method is $O(V) * O(V^2) * O(1) = O(V^3)$

4.2 Examples

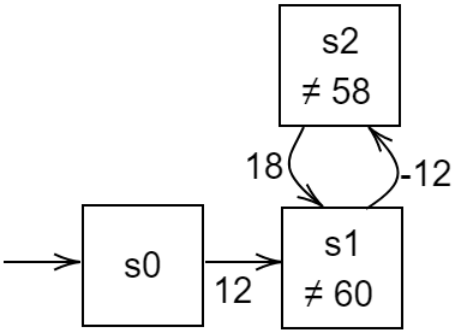

Figure 4: 1-VASS with a single positive cycle

As we can clearly see, the above figure (4.2) 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 can't take the cycle) and the step size (cycle weight) of the different closures, so all we're 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).

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, The way to do this, is to check whether or not these configurations can reach a configuration already in U_n. But we can't check each configuration in the bounded chains separately, as this amount isn't 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: $8V^7 + 10V^6 + 18V^5 + 24V^4 + 6V^3$.

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, reachability from an initial configuration (s, z_0) to an objective of the form

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

is polynomially bounded, 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 = $L(\|V\|)$ a polynomial. So, with that in mind, we can already ensure that we need to do at most L passes of the algorithm before we can stop, with L being presented as $(8V^8 + 10V^7 + 18V^6 + 24V^5 + 6V^4 + V^2 + V + 3)^2 * V + V^2 + 3$, which as a complete expanded form polynomial amounts to: $64V^{17} + 160V^{16} + 388V^{15} + 744V^{14} + 900V^{13} + 984V^{12} + 808V^{11} + 324V^{10} + 140V^9 + 144V^8 + 168V^7 + 156V^6 + 37V^5 + 2V^4 + 7V^3 + 7V^2 + 9V + 3$, which means $L = O(V^{17})$, an incredibly large but still polynomial amount.

Now we need to keep the configurations reachable from (q, z) after n passes, which would normally be bounded by E^n configurations with $n \leq L$, 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) 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)(m + 1) configurations (q, z') with $z' > z$. with L being the polynomial 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)(m + 1), and it can be computed from the set of the previous run in polynomial time.

5.2 Dividing U_n into objectives

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

$$O = \{(t, z) \mid z \geq \ell \wedge \bigwedge_{i=1}^{m} (z \not\equiv a_i \pmod{W}) \wedge \bigwedge_{i=1}^{n} (z \neq b_i)\} \tag{2}$$

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 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 Complexity

The main complexity part in this part of the algorithm is ensuring 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 O(V) bounded chains per node, so there are at most $O(V^2)$ bounded chains in the graph. The amount of objectives is also bounded by the amount of bounded chains, as such bounded by $O(V^2)$. 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 $O(E^n)$ with n the distance reached. We can state that the distance can at most be $L(V)$, which is a polynomial distance. however $O(E^L)$ 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))*(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 $O(L(V)*(n+L(V))*(m+1)*E)$ The entire algorithm is thus polynomialy bounded by $O(top(V)*V^4)*L(V)*(n+L(V))*(m+1)*E)$

5.4 Example

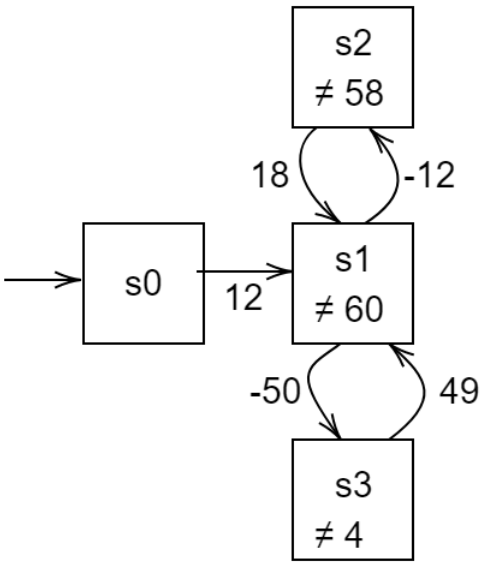

Figure 5: 1-VASS with a positive and a negative cycle

The above 1-VASS has 1 positive cycle containing s1 and s2, with a weight of 6. With this, we can determine that s1 has the bounded chains (0,54,6) and (4,70,6) and 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. Now 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 it's 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 it's 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. As such, continuing down the chain simply fails to reach an objective due to our inability to take the negative cycle at any point in time. And 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, and 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.

6 Conclusion

After dividing the complete algorithm into various different parts, and ensuring each part is in fact polynomial, we can ensure that deciding whether or not a 1-vector addition system with states and disequality guards is unbounded can in fact be done in polynomial time.

References

[1] Guillermo A. Pérez Mahsa Shirmohammadi James Worrell Shaull Almagor, Nathann Cohen. Coverability in Succinct One-Counter Nets with Disequality Tests. 2019.

[2] Guillermo A. Pérez Mahsa Shirmohammadi James Worrell Shaull Almagor, Nathann Cohen. Coverability in 1-VASS with Disequality Tests. *arXiv:1902.06576v2*, 2020.