Algorithm Draft

Clexma

March 17, 2019

According to the work of Haase, which gave a theoretical proof of the decidability and NP lower and upper bound for the reachability problem of one-counter automata.

Given a one-counter automata $\mathcal{A}$ and its corresponding transition system $T(\mathcal{A})$ defined as Haase. The reachability problem of one-counter auotmata can be stated as follows.

1-Counter Automata Reachability

INPUT: A one-coutner automata $\mathcal{A}$ and configurations $C, C' \in C(\mathcal{A})$, where $C(\mathcal{A}) = Q \times \mathbb{N}$.

OUTPUT: Does $C \to_{\mathcal{A}}^{*} C'$?

Due to the counter value and the possible infinity of the configurations of one-counter automata, finding a path in $T(\mathcal{A})$ is not realistic because it may cause exponential blow up. For example....Thus we turn to consider the flow $f : E \to \mathbb{N}$ which assigns each edge in the automata a natural number. We call a flow *path flow* if the assignment of the number exactly corresponds to the number of a path π go through the edge in the automata.

The following lemma gives neccessary conditions for a flow to be a path flow.

Lemma 1 (4.1.6 in Haase) *A flow f is a s-t path flow iff f satisfies the following conditions:*

- *If $s = t$ then*
  1. $\Sigma_{w \in out(v)} f(v, w) = \Sigma_{w \in in(v)} f(w, v)$ *for all* v
  2. $F(f)$ *is a s-t support*
- *If $s \neq t$ then*
  1. $\Sigma_{w \in out(v)} f(v, w) = \Sigma_{w \in out(v)} f(w, v)$ *for all* $v \in V - \{s, t\}$
     $\Sigma_{w \in out(s)} f(s, w) = \Sigma_{w \in out(s)} f(w, s) + 1$
     $\Sigma_{w \in out(t)} = \Sigma_{w \in out(t)} f(w, t) - 1$

2. *F(f) is connected.*

***Proof* 1** *Prove by induction on* $n = \Sigma_{e \in E} f(e)$

- $\Rightarrow$: *By definition direction of path flow the proof is obvious.*
- $\Leftarrow$:

  1. $s = t, F(f) isconnected$, *we have the conclusion that* f *is a s-t path flow.*
     - *case 1 : there is a self loop from* s *to* s.
     - *case 2 : there is no self loop from* s *to* s. *In this case we delete an edge from* v *to* s *such that* $v \neq s$.
       * *subcase 2.1 the resulting graph is still connectedl let the new flow be* f'. *By the induction hypothesis* f' *is a path flow from* s *to* v.
       * *subcase 2.2 the resulting graph is not connected, in this case , there are exactly two connected components such in the resulting graph, we call them* C_1 *and* C_2 *and assume they contain* s *and* v *respectively.*
         · *sub-subcase 2.2.1:* C_2 *contains no edges.*
         · *sub-subcase 2.2.2:* C_1 *and* C_2 *both contain at least one edge.*
  2. $s \neq t$ *is the same idea, we omit the proof here.*

By lemma 4.1.14 in Haase, the problem whether (q', n') is reachable from (q, n) can be solve by enumerate all the type-1, type-3, and type-2 reachability and guess nondeterministically the possible path between them. We now present the algorithm in detail.

Definition 1 (Reachability Certificate) *..*

By the definition of type-3 reachability certificate which require the positive cycle template. We put positive cycle template on the level of SCC of the weighted graph.

Definition 2 (Weighted Graph) *Let* $\mathcal{A}$ *be an one-counter automaton, the weighted graph of* $\mathcal{A}$ *is* $G = (V, E, \mu)$ *where* V *is a finite set of vertices according to the states of* $\mathcal{A}$, $E \subseteq V \times V$ *is a finite set of edges correspond to the edges of* $\mathcal{A}$ *and* $\mu : E \to \mathbb{Z}$ *is weight function correspond to the changes of the counter value of* $\mathcal{A}$.

We call $G' = (V', E', \mu')$ a *subgraph* of a weighted graph $G = (V, E, \mu)$ if $V' \subseteq V$, $E' \subseteq E$ and $\mu' = \mu$.

Definition 3 (Strongly Connected Component(SCC)) *A strongly connected component of a weighted graph G is a subgraph $G' = (V', E', \mu)$ of G and for $\forall v, v' \in V'$ and $v \neq v'$, there exists paths π from v to v' and π' from v' to v.*

In this paper, we have to add some more structure on an SCC to make te algorithm works. We do some classification on the number of type of vertices in an SCC:

Given an SCC $G' = (V', E', \mu)$ which is the subgraph of a weighted graph $G = (V, E, \mu)$, we call a vertice $v' \in V'$

- an *in-port vertex* if there exists $u \in V \wedge u \notin V'$ such that $(u, v) \in E$,
- an *out-port vertex* if there exists $u \in V \wedge u \notin V'$ such that $(v, u) \in E$,
- and an *internal vertex* otherwise.

In order to get a path $\Pi = \pi_1 \cdot \pi_2 \cdot \pi_3$ from s to t in $\mathcal{A}$ corresponding to a run in $T(\mathcal{A})$. Algorithm needs to guess the start node v_1 and end node v_2 of π_2. From lemma 4.1.17, if $|\pi_2| > 0$ there should be a positive cycle template at v_1 and a positive cycle template. Intuitively, a positive cycle template gives a overlook on the possible positive cycles in a SCC. That means if there is a simple positive cycle in an SCC, all other node can find a path to a point on that cycle and by circling the simple positive cycle many times. Then we can get a positive cycle on another point.

In the algorithm we first convert the one-counter automaton into weighted graph and then shrink the graph into a abstract graph defined on SCC. Now we define the shrinked weighted graph.

Definition 4 (Shrinked Weighted Graph) *Given a weighted graph $G = (V, E, \mu)$ we defined its corresponding shrinked weighted graph $SWG = (SCC(V), E', \mu')$, where*

- *$SCC(V) \subseteq V \times \mathbb{Z}^+$ is the set of all the SCCs and vertices in the same SCC have the same integer value.*
- *$E' \subseteq SCC(V) \times SCC(V)$ is a finite set of transitions between the SCCs. We also require for any $v, v' \in V$ and $a, b \in \mathbb{Z}^+$, if $((v, a), (v'b)) \in E'$ then $a \neq b$*
- *$\mu' : E' \to \mathbb{Z}$ is the weight function that assign each edge a weight. The value remain the same as μ in G.*

By the definition of SWG, it is obvious that it is a direct acycly graph. Now we need an algorithm to convert a weighted graph into a SWG by using Tarjan's algorithm to find the SCCs in a weighted graph.

[WIKI]

Now by executing algorithm 1 we preprocess a given WG into a SWG.

Algorithm 1 WG to SWG

```
SCCIndex := 1
function WG2SWG(G)
    Input: A weighted graph G = (V, E, μ)
    Output: A shrinked weighted graph SWG = (SCC(V), E′, μ′)

    index := 1
    S := empty stack
    Initialize an empty shrinked weighted graph G′:
    SCC(V) := V × 0, E := E, μ′ := μ

    for each (v, mark) ∈ SCC(V) do
        if mark == 0 then
            StrongConnect((v, mark))
        end if
    end for
    return SWG
end function

function StrongConnect((v, mark))
    v.index := index
    v.lowlink := index
    index := index + 1
    S.push(v)
    v.onStack := true

                                                ▷ Consider successors of v
    for each (v, w) ∈ E do
        if w.index == 0 then
            StrongConnect(w)
            v.lowlink := min(v.lowlink, w.lowlink)
        else if w.onStack then
            v.lowlink := min(v.lowlink, w.index)
        end if
    end for

    if v.lowlink == v.index then
        repeat
            w := S.pop()
            w.onStack := false
            (w, mark) := (w, SCCIndex)
                        ▷ Add w to current strongly connected component
        until w == v
        SCCIndex := SCCIndex + 1
                    ▷ Output the current strongly connected component
    end if
end function
```

Definition 5 (Abstract Shrinked Weighted Graph) *Given an* $SWG = (SCC(V), E', \mu')$*, we define its corresponding abstract shrinked weighted graph* $ASWG = (S, E'', T, \mu', \eta)$ *where*

- S *is a finite set of abstract state. A element* $s \in S$ *represents an SCC in* SWG*.*
- $\eta : S \to \mathbb{N}^+$ *is a tagging function that maps a state in* S *to its corresponding SCC index computed in algorithm 1,*
- $E'' \subseteq E'$ *is a set of detailed edges of the form* $((v_1, a), (v_2, b))$ *where* $(v_1, a), (v_2, b) \in SCC(V)$ *and* $a \neq b$*,*
- $T \subseteq S \times S$ *is a finite set of abstract edges. Given* $s_1, s_2 \in S$ *and* $(s_1, s_2) \in T$ *iff there exists a edge* $((v_1, t_1), (v_2, t_2)) \in E''$ *such that* $t_1 = \eta(s_1)$ *and* $t_2 = \eta(s_2)$*.*
- $\sigma : S \to \{\{+\}, \{-\}, \{+, -\}, \{0\}\}$ *is a function denoting whether there is a positive cycle(*$\{+\}$*), negative cycle(*$\{-\}$*), both(*$\{+, -\}$*), or none(*$\{0\}$*).*

We call the path $\tau : s_1 \cdots s_n$ on an ASWG a *abstract path* where $s_i \in S$ and $(s_i, s_{i+1}) \in T$

Given a weighted graph (V, E, μ) and its corresponding ASWG $(S < E'', T, \mu, \eta)$. We now define the operator $f_{SCC} : V \to S$ that maps a vertice to the SCC contains it.

Algorithm 2 converts an SWG to an ASWG.

The idea of the algorithm goes as follows. Given an one-counter automaton we first convert it into a weighted graph[Hasse]. Then use algorithm 1 and algorithm 2 we get corresponding SWG and ASWG. Then we need to finish the σ function in ASWG. The idea is using a Bellman-Ford like algorithm to detect whether there is a positive, negative cycle in a graph. Here is the algorithm.

By [Haase] the correctness of deciding whether an SCC contains a positive cycle template is given by the following lemma.

Lemma 2 *Given a strongly connected component* $SCC = (V', E', \mu)$*. There is a simple positive cycle in the SCC iff for any* $v \in V'$ *there is a positive v-cycle template respect to a* $n \in \mathbb{N}$*.*

Proof 2

- $\Leftarrow$*: By the definition of v-cycle template this direction is obvious.*
- $\Rightarrow$*: By the definition of v-cycle template in [Haase] and the definition of the strongly connected component in this paper. Assume there is a positive cycle* π_+ *in the SCC, there exists a path from* v *to any point one the v-cycle* π_+*. Here we only take a* $p \in V'$ *on* π_+ *and a v-p path* $\pi_{v\text{-}p}$*. Let* $n = min(drop(\pi_+) + weight(\pi_{v\text{-}p}), drop(\pi_{v\text{-}p}))$*. Then we have a positive v-cycle template* $\pi = p_1 \cdot p_2 \cdot p_3$ *respect to* n *where* $p_1 = \pi_{v\text{-}p}$ *and* $p_2 = \pi_+$*.*

After computing σ in $ASWG = (S, E'', T, \mu', \eta, \sigma)$ there are four kinds of situations to be considered:

Algorithm 2 SWG to ASWG

```
function SWG2ASWG(SWG)
    Input: SWG = (SCC(V), E', μ')
    Output: ASWG = (S, E'', T, μ', η, σ)

    SCCNum := 0
    for each (v, n) ∈ SCC(V) do
        if n > SCCNum then
            SCCNum := n                ▷ Find the number of the SCCs in SWG.
        end if
    end for
    Initialize ASWG:
    S := {s_i | i ∈ [1, ..., SCCNum]}
    E'', T := ∅
    for each ((v_1, n_1), (v_2, n_2) ∈ E') do
        if n_1 ≠ n_2 then
            Add (v_1, n_1), (v_2, n_2) to E''.
            if (s_{n_1}, s_{n_2}) ∉ T then
                T := T ∪ {(s_{n_1}, s_{n_2})}
            end if
        end if
    end for

    for each s ∈ S do
        σ(s) := TestCycle(S)
    end for
end function
```

Algorithm 3 Compute σ in $ASWG$

```
function TestCycle(SCC)
    Input:  SCC = (V', E', μ)
    Output:  o ∈ {{+}, {−}, {+, −}, {0}}.
    isPos := false
    isNeg := false
                                                    ▷ test positive cycle.
    n := #V'
    for i = 1 to n do
        for each v ∈ V' do
            d_v^i := max({0} ∪ {d_u^{i−1} + μ(u, v) : (u, v) ∈ E'})
        end for
    end for
    if there exists v ∈ V' such that d_v^n > d_v^{n−1} then
        isPos := true
    end if
                                                    ▷ test negative cycle.
    for i = 1 to n do
        for each v ∈ V' do
            D_v^i := min({+∞} ∪ {D_u^{i−1} + μ(u, v) : (u, v) ∈ E'})
        end for
    end for
    if there exists v ∈ V' such that D_v^n < D_v^{n−1} then
        isNeg := true
    end if

    if isPos ∧ isNeg then
        return {+, −}
    else if isPos then
        return {+}
    else if isNeg then
        return {−}
    else
        return {0}
    end if
end function
```

- **Situation 1:** There exists $s_1, s_2 \in S$ (possibly $s_1 = s_2$) such that $+ \in \sigma(s_1)$ and $- \in \sigma(s_2)$
- **Situation 2:** For any $s \in S$, $\sigma(s) = \{+\}$ or $\sigma(s) = \{0\}$
- **Situation 3:** For any $s \in S$, $\sigma(s) = \{-\}$ or $\sigma(s) = \{0\}$
- **Situation 4:** For any $s \in S$, $\sigma(s) = \{0\}$.

Due to the requirement that a type-3 reachability certificate require a positive cycle template and a negative cycle, we only need to find type-3 reachability certificate in **Situation 1**.

How to find all the possible path that may cause a reachability certificate. The diffcult problem is how to settle all the nondeterminism in the algorithm. Recall the definition of strongly connected componenet and its internal, inport and outport vertices before, we will use them to analyse how to generate a QFPA formula for reachability in the different situations above.

Assume the reachability problem ask given an-counter automaton $\mathcal{A}$ and configurations (s, n) an (t, n'), whether there is a run $(s, n) \rightarrow^* (t, n')$

Firstly convert the one-counter automaton to a weighted graph $G = (V, E, \mu)$, an SWG $G' = (SCC(V), E', \mu')$ and an ASWG $G'' = (S, E'', T, \mu', \eta)$.

Let $m = f_{SCC}(s) \in S, n = f_{SCC}(t) \in S$.

- **Situation 1:**

  Basic idea: Enumerate all the possible tuples (p, q) where $p, q \in S$, $+ \in \sigma(p)$ and $- \in \sigma(q)$.

  We first do DFS with a depth bound $|S|$ to find all the abstract path $\tau_1 : m \cdots p$ where $+ \in \sigma(p)$ and put these abstract path into set P_1.

  Likewise, do the same thing in G''^{op} and find all the abstract path τ_3^{op} : $n \cdots q$ where $- \in \sigma(q)$. convert τ_3^{op} to τ_3 and put all these abstract paths into set P_3.

  Then, we need to find whether there is a abstract path between p and q, the idea is also finding path by DFS. For pair (p, q), put all the abstract path $\tau_2 = p \cdots q$ into a set $P_2^{(p,q)}$.

  With set $P_1^p, P_2^{(p,q)}, P_3^q$ where $p, q \in S$ we can now generate an abstract path that can be use to construct the reachability certificate by choosing abstract paths from $P_1^p, P_2^{(p,q)}, P_3^q$ and concatenante them. We summarize the process above in algorithm 4.

  Now we have solved the nondeterminism of choosing possible (p, q), the nodeterminism of choosing the concrete path between SCCs and the possible paths in an SCC from inports to outports.

  By the definition of type-1 reachability certificate in [Haase], which requries a support without positive cycles, weight and the correct sum of the weight along the path. It is obvious that on a path the concrete edges

Algorithm 4 Find Type-2 Abstract Paths

```
function FindType2(WG, s, t)
    Input: A weighted graph WG, starting vertex s and ending vertex t.
    Output: An set of lists {[P_1^p, P_3^q, P_2^(p,q)]}_(p,q) where p, q are states in corresponding ASWG
    SWG := WG2SWG(WG)
    ASWG := SWG2ASWG(SWG)
    PosSet := {s | s ∈ ASWG.S, + ∈ ASWG.σ(s)}
    NegSet := {s | s ∈ ASWG.S, − ∈ ASWG.σ(s)}
    list := []
    for each p ∈ PosSet do
        P_1^p := DfsFindAbsPath(ASWG, s, p)
        for each q ∈ NegSet do
            P_2^(p,q) := DfsFindAbsPath(ASWG, p, q)
            P_3^(p,q) := DfsFindAbsPath(ASWG, q, t)
            list := list ∪ {[P_1^p, P_2^(p,q), P_3^q]}
        end for
    end for
    return list
end function

function DfsFindAbsPath(ASWG, p_1, p_2)
    Input: An ASWG = (S, E'', T, μ, η, σ), starting state p_1 and target state p_2.
    Output: a set of all the abstract paths P.
    current := p_1
    S := empty stack
    S.push(p_1)
    while ∃(current, u) ∈ ASWG.T not visited and current ≠ null do
        S.push(u)
        current := u
        if current == p_2 then
            Output the stack as a path to P
            S.pop()
            current := S.top()
        end if
        if all (current, u) ∈ ASWG.T are visited then
            S.pop()
            current := S.top()
        end if
    end while
end function
```

between SCCs will at most be visited once. Hence we can split the type-1 reachability certificate into SCCs the type-1 abstract path visited.

Given a SCC $G' = (V', E', \mu)$ by the definitions above, there is a set of inports V_{in} and a set of outports V_{out}.

Let $\{(v_i, v_o) \mid v_i \in V_{in}, v_o \in V_{out}\}$ be all the inport-outport tuples. Due to the storng connectivity of SCC, there exists at least one path in G' starting from v_i and ending at v_o. Since for one tuple there are often many reachable paths in the graph, we would like to introduce a dynamic programming techniques which saves time we computing the drop and weight of a path whose length is less than a integer value. We define a table that store the intermediate result of the computation.

Definition 6 (Drop-Weight Table(DWT)) *Given a graph $G = (V, E, \mu)$, we define its drop-weight table $DWT_G : V \times V \times \mathbb{N} \to 2^{\mathbb{Z}\times\mathbb{Z}}$ or simply use a tuple $DWT_G(v_1, v_2, l, P)$ to represent a entry of the table, where*

- *$v_1 \in V$ is the starting node of paths and $v_2 \in V$ is the ending node of paths.*
- *$l \in \mathbb{N}$ denote the maximum length of paths.*
- *$P = (d, w) \mid w \in \mathbb{Z}, d \in \mathbb{Z}$ is a set remembering all the drop-weight pairs.*

Given a integer n, we use $DWT_G(n)$ to denote a drop-weight table of G that the length of paths we consider is at most n.

With the definition of drop-weight table, we now need a algorithm to compute a table.

Algorithm 5 Compute DWT

---

function COMPUTEDWT(G)
    Input: A graph G and an integer $n \in \mathbb{N}$
    Output: A DWT table $DWT_G(n)$
end function
