Proving Functional Correctness of Tarjan's Algorithm and its C Implementation in Coq and VST

Litao Zhou
Shanghai Jiao Tong University
Shanghai, China
ltzhou@sjtu.edu.cn

Abstract

Tarjan's algorithm computes the strongly connected components of a finite graph using depth-first search. We formalize a small step description of the algorithm in Coq and verify an imperative implementation written in C programming language based on the small step description in VST.

CCS Concepts: • **Theory of computation → Separation logic;Program verification**; • **Program verification**;

Keywords: Strongly Connected Components, Separation logic, Coq, CompCert, VST

1 Background

1.1 Strongly Connected Components

The strongly connected components on a directed graph are defined as follows.

Definition 1.1 (Strong Connectivity)**.** Let G be a directed graph. If for each pair of vertices v, w in G, v is reachable to w and w is reachable to v. (i.e. exists path $p_1 : v \longrightarrow^* w$ and $p_2 : w \longrightarrow^* v$). Then G is said to be strongly connected.

Definition 1.2 (Strongly Connected Components)**.** Let G be a directed graph. Define an equivalence relation for two vertices if they are mutually reachable to each other. Let the distinct equivalence classes under this relation to be $\mathcal{V}_i, 1 \leq i \leq n$. Let $G_i = (\mathcal{V}_i, \mathcal{E}_i)$, where $\mathcal{E}_i = \{(v, w) \in \mathcal{E} | v, w \in V_i\}$, then

a) Each G_i is strongly connected
b) No G_i is a proper subgraph of a strongly connected components of G

The goal of a strongly connected component algorithm is to find a list of vertex sets $\{G_i = (V_i, E_i) | E_i = \{(u, v) \in E | u, v \in V_i\}\}$, where $\bigcup_i V_i = V$, satisfying the following two propositions.

Proposition 1.3 (SCC Connected)**.** *Each component G_i is strongly connected.*

Proposition 1.4 (SCC Maximal)**.** *No component G_i is a proper subgraph of a strongly connected components of G. i.e., for any $v \in V$, if v is mutually reachable to any $v' \in V_i$, then v should also be a member of V_i.*

In this report we are going to formally verify that the list of sets computed by Tarjan's algorithm[3] satisfies the two properties above.

1.2 Tarjan's Algorithm, a Step's View

```
struct vertex {
  struct edge *vedge;
  int dfn, low, instack, scc;
};
struct edge {
  struct vertex *etail;
  struct edge *next;
};
struct vertexstk {
  struct vertex * vitem;
  struct vertexstk * next;
};
int idx, sccidx;
struct vertexstk *stk;
void tarjan_visit(struct vertex * x){
  x -> dfn = x -> low = ++idx;
  stk_push(x);
  struct edge *pe = x->vedge;
  while (pe != NULL){
    struct vertex *y = pe -> etail;
    if (y -> dfn == 0){
      tarjan_visit(y);
      if (y -> low < x -> low)
        x -> low = y -> low;
    }
    else if (y -> instack == 1 &&
             y -> dfn < x -> low)
      x -> low = y -> dfn;
    pe = pe -> next;
  }
  if (x -> low == x -> dfn)
    stk_pop_top_until(x);
}
```

Figure 1. Tarjan's algorithm implementation

Figure 1 is the imperative Tarjan's algorithm that we verified. Our implementation performs depth first search on the adjacency table representation of a graph. Each vertex can access its linked list of edges from its `vedge` field. For each edge, its source node is determined by the linked list it belongs to, and has a field `etail` pointing to its destination. Note that in an actual program for finding all the strongly connected components, function `tarjan_visit` may be called more than once on top level until all the vertices have been visited. In this report, we will only present the specification and verification for the function `tarjan_visit` itself instead of the whole program.

A global stack `stk` and index `idx` are maintained throughout the algorithm. `idx` is initialized as 0, and will be incremented every time a new vertex is encountered. `stk_push(x)` will push the vertex `x` onto the top of the stack. `stk_pop_top_until(x)` will repeatedly pop the stack until `x` is popped. It will also update the `scc` field of the popped vertex to a new number i, showing that they belong to the same component $\mathcal{V}_i$.

To get a more intuitive understanding of what the algorithm does, we can view Tarjan's algorithm from a step's perspective. When a vertex v is being visited (i.e. is the argument the current function call), one of the following steps may be taken.

- If there is an unvisited edge e from v to an unvisited vertex u, u's Dfn and Low will be updated by an incremented Idx. u will be pushed onto the Stack. u will be passed into the `tarjan_visit` call and its neighbors will be explored next. (Forward Step)
- If there is an unvisited edge e connecting v to a visited vertex u, then $\mathrm{Low}(v)$ will be updated to $\mathrm{Dfn}(u)$, as long as u is still in the Stack and $\mathrm{Dfn}(u)$ is smaller than the old value of $\mathrm{Low}(v)$. (Forward Visited Step)
- If all edges starting from v have been visited, we should check whether $\mathrm{Low}(v) = \mathrm{Dfn}(v)$. If the result is true, we should repeatedly pop the stack until v is not in the stack and the popped vertices become a strongly connected component. Otherwise, nothing needs to be done with the stack. For both cases, we should next take a backward step from v for the DFS process to continue. (Backward Step)

Every step can be viewed as a relation on two consecutive states. Starting from an initial state, we can repeatedly use the rules above to describe the execution of Tarjan's algorithm.

2 An Overview of the Proof

One would naturally expect the formal proof of the graph algorithm to be based on an abstract graph, and follow the small step specification given in Section 1.2. Meanwhile, we also want our proof to be ultimately able to verify a complete imperative program, which treats graph as a concrete data structure (e.g. adjacency table, in our example) and manipulates the graph data by performing instructions.

To achieve this goal, the general framework of our proof is divided into two phases.

First, we define a small-step like description for the algorithm. The description is a relation of states, parameterized by a Graph object. This definition closely corresponds to our informal description in Section 1.2 and therefore has an intuitive implication (Section 3.2). We can use the reflexive transitive closure of this small-step relation to describe the fact that one state can be transformed to another after executing a few steps in the algorithm. We can perform induction on the reflexive transitive closure relation and prove many useful dynamic invariants of the algorithm (Section 3.3). We also need to define a static DFS tree object, in order to provide some important insights into the vertex visiting orders in depth-first search (Section 3.4). By composing the dynamic invariants of the small step description and the static property of the DFS tree, we can prove Proposition 1.3 and 1.4 hold at end-state (Section 3.5).

Next, with the model level theorems ready, to verify a imperative program, we prove that the small step description is a simulation of the imperative program. The challenge here is that separation logic requires disjoint assertions on memory blocks. We have designed proper predicates and efficient tactics to streamline the verification process in VST (Section 4.1 and 4.2). We will show that with small step description, we can provide straightforward specifications and invariants for the program (Section 4.3). The functional correctness of the imperative program can then be easily established by linking the two phases together (Section 4.4).

3 Proving Correctness of Small Step Description

3.1 States of the Algorithm

We define the state of the algorithm as a record type with the following fields. Without special declaration, we will refer to the graph object that the algorithm operates on as G and the initial vertex as v_0. Every graph object includes a set of Vertices and a set of Edges, together with a source/destination relation defined.

- $\mathrm{FuncStack}$: $\mathrm{list\ Vertex}$ is a trace of recursive calls.
- Dfn: $\mathrm{Vertex} \to \mathrm{Nat}$ For every visited vertex, starting from 1, Dfn indicates its depth-first search index. $\mathrm{Dfn}(v) = 0$ indicates that v has not been visited.
- Low: $\mathrm{Vertex} \to \mathrm{Nat}$ For every visited vertex v, Low indicates the smallest vertex's Dfn to which v is reachable by passing at most one back edge. $\mathrm{Low}(v) = 0$ indicates that v has not been visited.
- Stack: $\mathrm{list\ Vertex}$ records the vertices that have been reached during the search but have not yet been placed in a component.
- $\mathrm{SccList}$: $\mathrm{list\ (list\ Vertex)}$ collects the strongly connected components on the fly, initialized as an empty list. Every group of vertices represents a component discovered by the algorithm.

- VisEdge: $\text{Edge} \rightarrow \text{EdgeType}$ There are three types of edges, namely unvisited, tree and other. Initially all edges are mapped to unvisited type.
- Idx: Nat counts the number of vertices that have been reached so far. Idx is initialized as 1.

3.2 Transition Rules

We can now rewrite the informal algorithm description in Section 1.2 into a formal small step relation in Figure 2. We use notation $G \vdash s_1 \Rightarrow s_2$ to indicate that state s_1 can be one-step transmitted to s_2 through Tarjan's algorithm on graph object G.

S-ForwardTree describes the transition of a forward step, the step strictly follows our previous description in "Forward Step". $G \Vdash u \overset{e}{\rightsquigarrow} v$ is a relation defined in Ramify-Coq[4] indicating that u has an edge e to v.

$\text{S-ForwardVisited-1}$ and $\text{S-ForwardVisited-2}$ corresponds to the informal description of "Forward Visited Step". Note that when the destination vertex is not in the stack, nothing needs to be done with the Low field.

S-PopStack-1, S-PopStack-2 and S-Backward together capture the description of "Backward Step". Note that we use an extra field stage to enforce that a decision about whether to pop the stack should be made before an actual backward operation is performed.

To describe the execution of the whole algorithm, we can use the reflexive transitive closure of the single step defined above to represent whether one state is reachable to the other. We will use $G \vdash s_1 \Rightarrow^* s_2$ to indicate that state s_1 can be multi-step transformed to s_2 through Tarjan's algorithm on graph object G.

Our main focus is on the multi-step relations that begin with the initial state of the algorithm. The initial state is defined in figure 3.

Our ultimate goal in defining the small-step relation is to prove some useful properties on the final state. It helps to define what an endstate is.

Definition 3.1 (End State). An end-state s of a multi-step relation is a state where no steps can be taken any more, i.e. $\forall s', \neg s \Rightarrow s'$.

For Tarjan's algorithm, the endstate should have a more intuitive semantic. If the multi-step relation begins with a $\text{start\_state}(v_0)$, then the endstate should have the same function stack as the initial state and all out edges of v_0 should be marked as visited. For any state that does not have the above property, it will fall into one of the premises in Figure 2 and we can always construct a next-state for it. Therefore the following theorem holds.

Theorem 3.2 (End State Semantics). *For any state s, if $G \vdash \text{start\_state}(v_0) \Rightarrow^* s$, and s is an end state, then the function stack of s is $[v_0]$ and for any e such that $G \Vdash v_0 \overset{e}{\rightsquigarrow} v'$, e has been visited at s.*

3.3 Invariants over Transitions

We can prove many useful properties of the algorithm by doing n-1 induction on the multi-step relation. These properties can be generally considered as invariants over transitions. To prove an invariant right, we only need to prove that every single step can preserve the invariant. Below are a few important invariants that contribute to the proof of MAXIMAL and CONNECTED property. For ease of delivery we will refer to *current vertex* as the top vertex in FunStack on a certain state.

Note that some invariants where no Tarjan's algorithm specific fields (e.g. Low, Stack) are involved can be proved for any DFS algorithm. To improve modularity and avoid redundant proofs, we design an abstract algorithm description for depth-first search, which has four kinds of small steps:

- Forward Step: push new vertex onto FuncStack, update the Dfn field of the new vertex, and update the VisEdge field of the forward edge to an unvisited vertex.
- Backward Step: if all out-edges have been visited, pop a vertex from FuncStack.
- Forward Visited Step: Update the VisEdge field of the forward edge to a visited vertex.
- Static Step: do nothing

It is not hard to show that any invariants of depth-first search are also invariants of the Tarjan's algorithm, since every small step of Tarjan's algorithm is simply an extension of one of the above four steps in depth-first search. We present the invariants of Tarjan's algorithm as follows.

Lemma 3.3 (Reachable_DFS). *For any state s that can be multi-stepped from $\text{start\_state}(v_0)$, if v_1 and v_2 are on the function stack of s, v_1 is on top of v_2, then v_2 is reachable to v_1 through tree edges.*

Lemma 3.4 (Stack_Order). *For any state s that can be multi-stepped from $\text{start\_state}(v_0)$, if v_1 is on top of v_2 on the function stack of s, then v_1 is also on top of v_2 on the SCC stack of s.*

The proof of the above two lemmas can be obtained through n-1 induction on the multi-step relation. They will lead to the following lemma about the reachable relation on the stack.

Lemma 3.5 (Stack_Reachable). *For any state s that can be multi-stepped from $\text{start\_state}(v_0)$, if v_1 and v_2 are on the SCC Stack of s, v_1 is on top of v_2, and v_2 is in the function stack of s, then v_2 is reachable to v_1 through tree edges.*

Proof. by n-1 induction on the multi-step relation. Clearly, since the initial state only has v_0 in the stack, the lemma holds.

For the induction case, we have $\text{start\_state}(v_0) \Rightarrow^* s \Rightarrow s'$. Induction hypothesis says that on state s, any pair of vertices on the stack will have an offspring relation if the lower

$$\frac{pg \Vdash u \overset{e}{\rightsquigarrow} v, VisEdge(e) = unvisited, Dfn(u) = 0}{\text{graph}(pg) \vdash \left[\begin{array}{l} u :: FuncStack, Dfn, \\ Low, SccList, Stack, \\ VisEdge, Idx, stage1 \end{array}\right] \Rightarrow \left[\begin{array}{l} v :: u :: FuncStack, (v \mapsto Idx + 1; Dfn), \\ (v \mapsto Idx + 1; Low), SccList, v :: Stack, \\ (e \mapsto tree; VisEdge), Idx + 1, stage1 \end{array}\right]} \text{ S-ForwardTree}$$

$$\frac{pg \Vdash u \overset{e}{\rightsquigarrow} v, VisEdge(e) = unvisited, Dfn(u) \neq 0, v \in Stack}{\text{graph}(pg) \vdash \left[\begin{array}{l} u :: FuncStack, Dfn, \\ Low, SccList, Stack, \\ VisEdge, Idx, stage1 \end{array}\right] \Rightarrow \left[\begin{array}{l} u :: FuncStack, Dfn, \\ (u \mapsto \min(Dfn(v), Low(u)); Low), \\ SccList, Stack, (e \mapsto other; VisEdge), \\ Idx, stage1 \end{array}\right]} \text{ S-ForwardVisited-1}$$

$$\frac{pg \Vdash u \overset{e}{\rightsquigarrow} v, VisEdge(e) = unvisited, Dfn(u) \neq 0, v \notin Stack}{\text{graph}(pg) \vdash \left[\begin{array}{l} u :: FuncStack, Dfn, \\ Low, SccList, Stack, \\ VisEdge, Idx, stage1 \end{array}\right] \Rightarrow \left[\begin{array}{l} u :: FuncStack, Dfn, Low, \\ SccList, Stack, (e \mapsto other; VisEdge), \\ Idx, stage1 \end{array}\right]} \text{ S-ForwardVisited-2}$$

$$\frac{Dfn(u) \neq Low(u) \ \wedge \ \forall e\ w, pg \Vdash v \overset{e}{\rightsquigarrow} w \Rightarrow VisEdge(e) \neq unvisited}{\text{graph}(pg) \vdash \left[\begin{array}{l} u :: FuncStack, Dfn, \\ Low, SccList, Stack, \\ VisEdge, Idx, stage1 \end{array}\right] \Rightarrow \left[\begin{array}{l} u :: FuncStack, Dfn, \\ Low, SccList, Stack, \\ VisEdge, Idx, stage2 \end{array}\right]} \text{ S-PopStack-1}$$

$$\frac{Dfn(u) = Low(u) \ \wedge \ \forall e\ w, pg \Vdash v \overset{e}{\rightsquigarrow} w \Rightarrow VisEdge(e) \neq unvisited}{\text{graph}(pg) \vdash \left[\begin{array}{l} u :: FuncStack, Dfn, Low, \\ SccList, pop@[u]@rem, \\ VisEdge, Idx, stage1 \end{array}\right] \Rightarrow \left[\begin{array}{l} u :: FuncStack, Dfn, Low, \\ (pop@[u]) :: SccList, rem, \\ VisEdge, Idx, stage2 \end{array}\right]} \text{ S-PopStack-2}$$

$$\frac{}{\text{graph}(pg) \vdash \left[\begin{array}{l} v :: u :: FuncStack, Dfn, \\ Low, SccList, Stack, \\ VisEdge, Idx, stage2 \end{array}\right] \Rightarrow \left[\begin{array}{l} u :: FuncStack, Dfn, \\ (u \mapsto \min(Low(v), Low(u)); Low), \\ SccList, Stack, VisEdge, Idx, stage1 \end{array}\right]} \text{ S-Backward}$$

Figure 2. Small Step Description for Tarjan's Algorithm

$$\text{start\_state}(v_0) = \left\{ \begin{array}{lll} \text{FuncStack} : [v_0], & \text{Dfn} : \lambda v.\text{if } v = v_0 \text{ then } 1 \text{ else } 0, & \text{Stack} : [v_0]\,, \\ \text{SccList} : [\,]\,, & \text{Low} : \lambda v.\text{if } v = v_0 \text{ then } 1 \text{ else } 0, & \text{Idx} : 1, \\ \text{Stage} : \text{stage1} & \text{VisEdge} : \lambda e.\text{unvisited} & \end{array} \right\}$$

Figure 3. Initial state of multi-step relation

vertex is also in the function stack. We need to prove that state s' also has this property.

Note that the only single steps that may change the stack or the tree edge map are S-ForwardTree and S-PopStack. For the rest of the cases, the preservation can be proved directly.

For S-ForwardTree step, by Lemma 3.3, the pushed vertex is an offspring of all the vertices in the function stack. For S-PopStack, by Lemma 3.4, all the v_2's in the function stack except the current vertex still remain on the SCC stack. Since the SCC stack at s' is a part of the SCC stack of s, applying the induction hypothesis will finish the proof. □

For proving CONNECTED, it suffices to show that every pair of vertices in a component are mutually reachable through the root. Thus we only need to prove: for each popped component, every vertex in this component and its root (i.e. the vertex that triggers the popping action) are mutually reachable.

From Lemma 3.5, we can prove one direction of the Connected property. By instantiating v_2 as the current vertex, all the popped v_1's are before v_2 in the stack, thus reachable from the current vertex.

Theorem 3.6 (Connect_Direction_1). *For any state s that can be multi-stepped from* $\text{start\_state}(v_0)$*, if $s \Rightarrow s'$ is a $S-PopStack-2$ step, v is the current vertex of s, then for any v' in the popped component, v is reachable to v'*

We continue proving the other direction. This part of proof is a little trickier, but we can still use n-1 induction on multi-steps to construct invariants and guide the proof.

Lemma 3.7 (Low_Property). *For any state s that can be multi-stepped from* $\text{start\_state}(v_0)$*, for any v in the SCC stack of s, either* $v.\text{dfn} = v.\text{low}$ *or there exists a v' whose* dfn *is equal to* $v.\text{low}$*, v' can reach v through several tree edges, and v can reach v' through several tree edges and one back edge.*

Lemma 3.8 (Stack_Dfn_Property). *For any state s that can be multi-stepped from* $\text{start\_state}(v_0)$*, if v_1 and v_2 are vertices in the SCC stack, then v_1 is on top of v_2 on the SCC stack of s if and only if* $v_1.\text{dfn} < v_2.\text{dfn}$.

Lemma 3.9 (Low_Dfn_Relation). *For any state s that can be multi-stepped from* $\text{start\_state}(v_0)$*, for any v, we have* $v.\text{low} \leq v.\text{dfn}$ *on state s.*

Lemma 3.10 (Stack_FunStack_Relation). *For any state s that can be multi-stepped from* $\text{start\_state}(v_0)$*, for any v, if v is in the SCC stack of s, and* $v.\text{dfn} = v.\text{low}$ *on s, then v is also in the function stack of s.*

Still, the above four lemmas can be proved following n-1 induction on the multi-step relation. They will contribute to the following lemma.

Lemma 3.11 (Stack_reachable_Cur). *On any state s that can be multi-stepped from* $\text{start\_state}(v_0)$*, for any vertex v on the stack, v is reachable to the current vertex x of state s.*

Proof. We first assert that for any vertex v' in the stack, either we can prove it is reachable to x, or there exists a path to u, where $u.\text{dfn} < v'.\text{dfn}$, and u is still in the stack.

From Lemma 3.10, if $v'.\text{low} = v'.\text{dfn}$, then v' is on the function stack. Lemma 3.3 implies v' is reachable to x. If $v'.\text{low} < v'.\text{dfn}$, then Lemma 3.7 provides such a u as we need in the stack. Lemma 3.9 implies that $v'.\text{low} > v'.\text{dfn}$ is impossible. Therefore the assertion holds.

Note that Lemma 3.8 shows that Dfn decreases monotonically on SCC stack, by induction on the length of SCC stack we can show that v can eventually reach x by repeatedly applying the assertion above. □

Since the popped component is always a sub-list of the stack, and the current vertex x is always in the popped component when a S-PopStack-2 step is taken, the other direction of the Connected property follows directly from Lemma 3.11.

Theorem 3.12 (Connect_Direction_2). *For any state s that can be multi-stepped from* $\text{start\_state}(v_0)$*, if $s \Rightarrow s'$ is a $S-PopStack-2$ step, v is the current vertex of s, then for any v' in the popped component, v' is reachable to v*

Still, by performing n-1 induction on the multi-step, since the SccList grows monotonically, we can prove that at end state, all components in the SccList have the Connected property, formally stated as follows.

Theorem 3.13 (Tarjan_Connected). *If s can be multi-stepped from* $\text{start\_state}(v_0)$*, and s is an end state (i.e. no more steps can be taken from s), then for any component c in* SccList *of s, any pair of $v_1, v_2 \in c$ are mutually reachable.*

The above proof demonstrates the superiority of small step description in constructing readable and intuitive proofs. Every lemma can correspond to an invariant, either specifying the semantics of a field or revealing the relations between several fields throughout the algorithm. All the proofs are carried out in the n-1 induction fashion.

3.4 Property of DFS Tree

The proof of invariants has provided a lot of properties for the final state. However, to prove the Maximal property, we need some further lemmas that can dig deeper into the nature of DFS. It is the visting order of DFS that ensures the Maximal property of the DFS tree. We define a DFS tree as follows.

Definition 3.14 (DFS Tree). The DFS Tree T of a directed graph G is defined as a sub-graph of G with the following properties.

1. T is a tree, i.e.
   a. Has exactly one root node that has no in edges,
   b. Every node except the root in T has only one in edge,
   c. Every node is reachable from the root by tree edges,
2. There exists a total order Ord on the vertex set of T,
3. If $v_1, v_2 \in Ord$, any vertices in the sub-tree of v_1 will not have a direct edge on G to the sub-tree of v_2.

Here we choose to define the DFS tree as a subgraph of the original graph, instead of as a recursive tree structure. In this way, the connectivity on the DFS tree can have a direct correspondence to the connectivity of tree edges on the original graph. To be specific, $T(\{\mathcal{V}, \mathcal{E}\}, s)$ can be defined as $\{(\mathcal{V}, \mathcal{E}') | \mathcal{E}' = \{e \in \mathcal{E} | s.\text{VisEdge}(e) = \text{tree}\}\}$. We can prove that the multi-step relation can always generate a valid DFS tree.

The Ord in the definition can be simply understood as the "less than" relation on Dfn in the small step description. We generalize this concept in the DFS tree so that the property of DFS tree can apply to other small step descriptions that may not necessarily use a natural number to represent the visiting order. We can formally define the relation between two vertices with respect to their visiting orders.

Definition 3.15 (Relation between Vertices)**.** On a DFS Tree T, two vertices v_1 and v_2 can have the following relations

1. v_1 is the **ancestor** of v_2 if v_1 is reachable to v_2 on T. Conversely, v_2 is the **offspring** of v_1
2. Two vertices are **brothers** if there exists a common ancestor of them. In particular v_1 is the **left** brother of v_2 if $(v_1, v_2) \in Ord$. Conversely v_1 is on the **right** brother of v_2 if $(v_2, v_1) \in Ord$.

It is not hard to prove the following lemmas based on the basic properties of the DFS tree.

Proposition 3.16 (DFS Tree Property)**.** *On a DFS Tree T,*

1. *No loop exists on DFS Tree.*
2. *Any two vertices on the DFS Tree have a (nearest) common ancestor.*
3. *The "offspring" relation of two vertices are transitive*
4. *The "left" relation of two vertices are transitive*
5. *If two vertices are not in the offspring or ancestor relation, then they must be brothers*

The last property in Definition 3.15 captures the depth-first essence of the algorithm. A vertex will not be marked as "finished", i.e. in our algorithm, popped out of the function stack, until all its reachable vertices have been explored. We can further develop a useful lemma based on the DFS property.

Lemma 3.17 (Reachable on Subtree)**.** *If v_1 is left of v_2, v_1 and v_2 mutually reachable, then there exists a common ancestor v_p of v_1 and v_2, such that v_1 is reachable to v_p.*

Proof. by induction on the path from v_1 to v_2. The base case clearly holds since $v_1 = v_2$ when the path is empty. For the induction case, assume v_1 is reachable to v_3 and v_3 has an edge to v_2. DFS Tree property implies that we can find a common ancestor v'_p for v_1 and v_3. The induction hypothesis indicates that v_1 is reachable to v'_p if v_1 is a left brother of v_3. We perform case analysis based on the relation of v_2 and v_3.

1. v_3 is the offspring of v_2. If v_3 is already the common ancestor of v_1 and v_2, then v_3 is the v_p we should find. Otherwise, v_1 is a left brother of v_3, the v_p can be obtained from the induction hypothesis.
2. v_2 is the offspring of v_3. Since v_1 is a left brother of v_2, v_1 is also a left brother of v_3. The v_p can also be obtained from the induction hypothesis.
3. v_2 is the left brother of v_3. By the transitivity of the "left brother" relation, v_1 is also a left brother of v_3. The v_p can also be obtained from the induction hypothesis.
4. v_2 is the right brother of v_3, but v_2 has an edge to v_3, contradicting to the property of DFS Tree.

□

Since the nearest common ancestor belongs to the set of common ancestors, we can refine Lemma 3.17 into the following form.

Theorem 3.18 (Reachable to NCA)**.** *If v_1 and v_2 are brothers and mutually reachable on G, then their nearest common ancestor v_p can be reached from v_1 on G.*

3.5 Putting it All Together

To make use of Theorem 3.18 and prove the Maximal property of the end state, we need to prove the following lemmas.

Lemma 3.19 (Tree_Reachable_Stack)**.** *For any state s that can be multi-stepped from* $\mathrm{start\_state}(v_0)$*, if v_1 and v_2 are on the SCC Stack of s, v_1 is reachable to v_2 through tree edges, i.e. v_1 is the offspring of v_2, and v_2 is in the SCC stack of s, then v_1 is also in the SCC stack.*

Lemma 3.20 (One Hop Reachable in Stack)**.** *At any state s that can be multi-stepped from* $\mathrm{start\_state}(v_0)$*, for any two vertices v_1, v_2, if v_2 can reach v_1 through a path that only has one back edge at the end, then if v_1 is in the SCC stack, v_2 must also be in the SCC stack.*

Lemma 3.21 (NCA exists)**.** *At any state s that can be multi-stepped from* $\mathrm{start\_state}(v_0)$*, for any $c \in s.\mathrm{SccList}$, there exists a common ancestor $v \in c$, such that v is the offspring of any vertex in c.*

The above three lemmas are not hard to prove by n-1 induction on the multi-step relation. We will see how they contribute to the following important lemmas leading to the Maximal property.

Lemma 3.22 (Reachable in Stack)**.** *At any state s that can be multi-stepped from* $\mathrm{start\_state}(v_0)$*, for any two vertices v_1 v_2, if v_2 is reachable to v_1 through visited edges, then if v_1 is in the SCC stack, v_2 must also be in the SCC stack.*

Proof. Since any reachable path can be split into several segments that has exactly one back edge on the tail and one (potentially empty) path that is completely composed of tree edges, the proof can be completed by induction on the path from v_2 to v_1.

The induction case can be handled by Lemma 3.20. For the base case, where only tree edges are left, we can show that all ancestors of v_2 are in the stack by applying Lemma 3.19. □

Lemma 3.23 (Reachable Same Scc)**.** *At any state s that can be multi-stepped from* $\mathrm{start\_state}(v_0)$*, for any two vertices v_1 v_2, if v_2 is the offspring of v_1, and v_2 is reachable to v_1 through visited edges, if v_1 is in the SCC stack or a strongly connected component, then v_2 must be simultaneously in the stack or in the same component.*

Proof. by n-1 induction on the multi-step relation. Clearly, on the initial state the lemma holds.

For the induction case, we have $\mathrm{start\_state}(v_0) \Rightarrow^* s \Rightarrow s'$. There are three kinds of single steps that may change the stack. For the rest of the cases, the preservation can be proved directly.

- For $\mathrm{S\text{-}ForwardTree}$ step, a new tree edge is visited and a new vertex is pushed into the stack. Note that new edge won't affect the rechability in the popped components. We only have to consider the case where v_1 is in the SCC stack. We can apply Lemma 3.22 on v_2's reachability to v_1 to prove v_2 is also in the SCC stack.
- For $\mathrm{S\text{-}ForwardVisited}$ step, a new back edge is visited. Similarly, we only have to consider the case where v_1 is in the SCC stack. We can also apply Lemma 3.22 on v_2's reachability to v_1 to prove v_2 is also in the SCC stack.
- For $\mathrm{S\text{-}PopStack}$ step, we can also use Lemma 3.22 to show that all v_1's offsprings that are reachable to v_2 are still on the stack after popping.

□

Theorem 3.24 (Tarjan_Maximal). *If s can be multi-stepped from $\mathrm{start\_state}(v_0)$, and s is an end state (i.e. no more steps can be taken from s), then for any component c in $\mathrm{SccList}$ of s, if a vertex v' is mutually reachable to any vertex in c, then $v' \in c$.*

Proof. From Lemma 3.21 we know that a "root" v_p exists for the component c. From condition we know v_p and v' are mutually reachable. Proposition 3.16 implies that we can perform case analysis on the visiting order of v_p and v'.

- If v' is an offspring or ancestor of v_p, we can get $v' \in c$ by applying Lemma 3.23 directly.
- If v' is left of v_p, then we may find the nearest common ancestor v_r of v' and v_p. From Theorem 3.18, we know that v' should be reachable to v_r. We argue that v_r must be equal to v_p. Otherwise, we can find a father node y, who has a tree edge pointing to v_p. Note that v_p is reachable to y through $v_p \rightsquigarrow v' \rightsquigarrow v_r \rightsquigarrow y$. By Lemma 3.23, y should also be in the component, contradicting to the fact that v_p is the ancestor of all vertices in the component.
- If v' is right of v_p, and the nearest common ancestor is v_r. Similarly, from Theorem 3.18, we know that v_p should be reachable to v_r. We argue that v_r must be equal to v_p. Otherwise, we can find a father node y, who has a tree edge pointing to v_p. Now, that v_p is reachable to y through $v_p \rightsquigarrow v_r \rightsquigarrow y$. By Lemma 3.23, y should also be in the component, contradiction is yielded.

All cases have indicated that $v' \in C$, which finishes the proof of MAXIMAL property.

□

4 Proving the Simulation Relation on C Program

To prove the functional correctness of an imperative implementation of Tarjan's algorithm, we must define a predicate of graph representation on the memory model of the imperative language and relate it to the graph and field operation in the small step description. We present our verification on Verifiable C, the program logic of the Verified Software Toolchain (VST). We will begin with the basic definitions of predicates that describe the memory representation of graphs and then demonstrate how small step description fits into the program verification process.

4.1 Representation predicates for graphs, a simple version

The Tarjan's Algorithm is designed to be of $O(|V|+|E|)$ running time. Therefore, we choose the adjacency table instead of adjacency matrix as the graph representation implementation in our imperative program to be verified.

In our proofs, we directly treat addresses of `struct vertex` and `struct edge` as vertices and edges in the graph.

Our definition of graph predicates basically follows the memory layout of an adjacency table, shown in Figure 4. Starting from the bottom, we use $\text{field\_storage}_{\pi_V}(v, s)$ to indicate that the fields of vertex v are stored on address v, and values of all fields should be captured by state s. We also add a subscription to specify the read/write permission of the representation predicate. We will discuss about its details later.

We use $\text{edge\_storage}_{\pi_G}(G, v, es)$ to indicate that a list of values es stores the out-edges of v in the graph $G = \{\mathcal{V}, \mathcal{E}\}$. We use a pure proposition elis_prop to capture all out-edges of v. A global order $<_E$ on edges is introduced to ensure a deterministic es.

field_storage and edge_storage together provide the full description for the representation of a single row on the adjacency table (i.e. a vertex and the linked list of its out-edges). We combine them and use vertex_storage to represent multiple rows in the adjacency table.

Note that vertex_storage does not enforce the order of vertices in vs to follow a certain order. This is because all the information about vertices are already represented by the graph object G. In fact, the order of vertices will not make a difference in our implementation, since we do not rely on this order to search the graph. The exploration of new vertices are based on the `etail` field of the `edge struct`.

We use proposition vlis_prop to enforce vs to capture all the vertices in graph $G = \{\mathcal{V}, \mathcal{E}\}$.

Above, we combine vlis_prop and vertex_storage and define the representation predicate of a graph graph_rep.

Other parts of the program memory, e.g. the SCC stack, and global variables such as Idx, will be represented by a predicate global_rep(s), where s is the abstract state.

$$\begin{aligned}
\text{field\_storage}_{\pi_V}(v,s) &\triangleq v.\texttt{dfn} \mapsto_{\pi_V} s.\mathrm{Dfn}(v) \;*\, v.\texttt{low} \mapsto_{\pi_V} s.\mathrm{Low}(v) *\\
&\quad v.\texttt{instack} \mapsto_{\pi_V} \text{if } v \in s.\mathrm{Stack} \text{ then } 1 \text{ else } 0 *\\
&\quad v.\texttt{scc} \mapsto_{\pi_V} (i : v \in s.\mathrm{SccList}[i])\\[1em]
\text{edge\_storage}_{\pi_G}(G, pe, e :: es) &\triangleq \exists pe', e.dst \in \mathcal{V} \wedge pe.\texttt{etail} \mapsto_{\pi_G} e.dst\\
&\quad pe.\texttt{next} \mapsto_{\pi_G} pe' \;*\; \text{edge\_storage}_{\pi_G}(G, pe', es)\\
\text{edge\_storage}_{\pi_G}(G, pe, []) &\triangleq \text{emp}\\
\text{elis\_prop (G, v, es)} &\triangleq \text{NoDup } es \;\wedge\; \forall [e_1, \ldots, e_2] \subseteq es, e_1 <_E e_2 \wedge\; \forall e, (e \in \text{es} \Leftrightarrow e.src = v)\\[1em]
\text{vertex\_storage}(G, vs, s)_{\pi_G,\pi_V} &\triangleq \mathop{*}_{v \in vs} \begin{pmatrix} \exists es, \text{elis\_prop}(G, v, es) \;\wedge\; \text{edge\_storage}_{\pi_G}(G, v, es) * \\ v.\texttt{vedge} \mapsto_{\pi_G} \text{hd}(es) * \text{field\_storage}_{\pi_V}(v, s) \end{pmatrix}\\[1em]
\text{vlis\_prop}(G, vs) &\triangleq \text{NoDup } vs \;\wedge\; \forall v, (v \in \text{vs} \Leftrightarrow v \in \mathcal{V})\\[1em]
\text{graph\_rep}_{\pi_G,\pi_V}(G,s) &\triangleq \exists\, vs, \text{vlis\_prop (G, vs)} \wedge \text{vertex\_storage}_{\pi_G,\pi_V}(G, vs, s)\\[1em]
\text{global\_rep}(s) &\triangleq \exists \mathrm{StkPtr}, \texttt{stk} \mapsto \mathrm{StkPtr} \;*\; \mathrm{LL}(s.\mathrm{Stack}, \mathrm{StkPtr}) \;*\\
&\quad \texttt{idx} \mapsto s.\mathrm{Idx} * \texttt{sccidx} \mapsto \text{length}(s.\mathrm{SccList})
\end{aligned}$$

Figure 4. Predicate Definition of Graphs

4.2 Representation predicates with fractional permission

For now, we can already use graph_rep to describe all memory that our program uses for graph representation. In a separation logic proof, we can unfold this predicate and pick one vertex from the big separating conjunction when reasoning about memory load/store on that vertex. However, at certain program points, it may require extra proof efforts.

Consider, for example, in the loop body, a vertex y connected to x is being accessed. The reasoning of loop body will involve the memory of both vertices simultaneously. As a result in separation logic, we need to perform case analysis on whether x and y points to the same `vertex struct`. We consider such proof as verbose and counter-intuitive, since algorithm correctness proof already covers the case where a vertex has an edge pointing to itself.

To address this problem, we can base our verification on permission accounting[2] to bypass the tedious work of folding/unfolding graph_rep. Originally, permission was introduced for concurrent program verification when reasoning about concurrent-read/exclusive-write resources. Our practice has shown that permission accounting also has a place in the verification of complicated single-threaded programs like Tarjan's algorithm. In our example, we actually don't have to care whether an out-edge from v_1 points to itself or not, since we can logically assume that two virtual operators are manipulating the same graph concurrently. Their respective operations do not affect each other. It makes sense to adopt the idea of permission accounting in our verification.

This is why every representation predicate comes with one or two permission extra parameters. They tell whether the assertion grants read permission, write permission, or some other fractional permission. To be specific, we use π_G to refer to a readable permission for edge_storage and the `vedge` field of `vertex struct`, since Tarjan's algorithm does not modify the graph structure. We assign writable permission π_V on field_storage so that vertex fields can change as the program runs.

Any readable permission π can be split into two readable fractional permissions $\pi/2$, and any writable permission can be split into itself and $\perp$, which indicates no permission. Therefore we can prove the following lemma for the predicate graph_rep.

Theorem 4.1 (Split graph_rep). *if π_G is readable permission and π_V is writable permission,*

$$\text{graph\_rep}_{\pi_G,\pi_V}(G,s) \vdash \text{graph\_rep}_{\pi_G/2,\pi_V}(G,s) * \text{graph\_rep}_{\pi_G/2,\perp}(G,s)$$

Now our graph can be split into two "copies". The one with π_V will be used for updating the Vfield and the other with $\perp$ permission will be used for unfolding the adjacency table.

Any updates in predicates with writable permission will overwrite other predicates without writable permission when they are joined together. Therefore for graph_rep we can derive the following rules.

Theorem 4.2 (Join graph_rep). *if π_G is readable permission and π_V is writable permission,*

$$\text{graph\_rep}_{\pi_G/2,\pi_V}(G,s_2) * \text{graph\_rep}_{\pi_G/2,\perp}(G,s_1) \vdash \text{graph\_rep}_{\pi_G,\pi_V}(G,s_2)$$

Theorem 4.2 implies that we do not have to maintain the consistency of abstract states on our two copies of graph at every program point. We can safely leave the unfolded graph_rep$_{\pi_G/2,\perp}$ not updated in the loop invariant, so that we can save the reasoning on merging fragment predicates into a whole graph predicate when verifying the loop body. Then the updates on vertex fields can merely operate on the graph_rep$_{\pi_G/2,\pi_V}$ predicate. When we encounter a recursive call tarjan_visit(y), we can simply apply the frame rule and pass the graph_rep$_{\pi_G/2,\pi_V}$ part into the pre-condition of the callee function's specification, still leaving the graph_rep$_{\pi_G/2,\perp}$ unfolded. The two graph_reps will be joined together at the end of the program where the old s_0 is wiped out.

Based on the theories above, we can eliminate redundant proofs in the folding and unfolding derivation on graph predicates. Our permission accounting based strategy can achieve great simplicity and modularity in verification.

4.3 Function Specifications

The key components in verifying a recursive implementation of Tarjan's algorithm are to provide the function specification and the loop invariant for the verification system. Our solution is presented in figure 5. Throughout the program, we will be using the small step description to describe program states. We don't have to write out the complex algorithm-correctness level invariants on program states explicitly.

The pre-condition of the specification indicates that every recursive call should start with a "Forward" step. Note that the representation predicates here do not correspond to the small step state exactly: the Dfn and Low value of x have not been modified in the memory but are already updated in the small step description. But since the small step description only serves as an auxiliary assertion, and the control flow of our program will guarantee that these fields will always be updated in the next few instructions(Line 16 - 18 in figure 1), it is reasonable that the small step description outpaces the actual memory state at certain program points, as long as the representation predicates provide an accurate assertion on program states.

The loop invariant is also described by the small step description. Since our small step relation does not enforce an order in selecting out-edges, we refine the assertion with a few extra propositions about the $\mathrm{VisEdge}$ field of s. The out-edges list in the adjacency table is split into two folds, visited and unvisited. Every iteration will mark a new visited edge, moving the head of the second half into tail of the first half. The exit of the loop indicates the second half is empty, and all the out-edges are visited. As has been explained in section 4.2, the looping through out-edges on the graph is represented by the unfolded graph_rep$_{\pi_G/2,\perp}$. graph_rep$_{\pi_G/2,\pi_V}$ is folded and consistent with the small step state in the loop invariant.

The assertion derivation at entry of the loop is trivial. The abstract small step state have become consistent with the actual memory state at loop invariant. We can simply take s to be state s_0 in the context. As for the loop body, it has three branches. If the first branch is taken, branch conditions gives $\mathrm{Dfn}(y) = 0$, we can construct a "Forward" step to prove that the current state of the program satisfies the pre-condition of `tarjan_visit(y)`. According to the recursive specification, the post condition of this function call contains a multi-step relation and the assertion that all out-edges of y are visited. We can then construct a "Backward" step, which derives the loop invariant. If the second branch is taken, branch condition gives $\mathrm{Dfn}(y) \neq 0 \wedge\ y \in \mathrm{Stack}\ \wedge \mathrm{Low}(y) < \mathrm{Low}(x)$. We can construct a "Forward Visited" step to derive the loop invariant. Similarly for the last branch, we can also construct a step in order to derive the loop invariant.

From the exit of the loop to the end of function, the exit condition indicates that all out-edges have been visited. A `stk_pop_top_until(x)` function is called on condition that $\mathrm{Low}(x) = \mathrm{Dfn}(x)$, and then the function returns. Therefore a "Backward" step can be constructed. Now we finish the implementation verification that our C program is a simulation of the small step description.

4.4 Finishing the Proof

To summarize, we choose small step description to formulate and prove the algorithm correctness in an intuitive way. As for the implementation proof, we demonstrate the flexibility of using relation instead of function as the formal algorithm description in constructing straightforward recursive invariants. We also show that permission accounting, which is originally designed for verifying concurrent programs, can help streamline the proof in verifying memory manipulations for complex sequential programs like Tarjan's algorithm.

The only work left for us in proving functional correctness of the C program is to show the user-expected specification in figure 6 holds for function `tarjan_visit(x)`.

Note that the user-expected specification is simply a specialization of the specification in figure 5. For the derivation of pre-condition, we can fill the multi-step relation as a reflexive relation. By setting the $\mathrm{VisEdge}$ map to map all edges as unvisited, the extra proposition on edge fields can also be satisfied. As for the derivation of post-condition, we can show that the witness of the multi-step relation s' is an end-state that can not be stepped to any other state. We can apply the MAXIMAL(Theorem 3.24) and CONNECTED(Theorem 3.13) property to finish the proof.

We base our verification on Verifiable C, the program logic of the Verified Software Toolchain (VST). The idea of "specialization" above is implemented in VST as the theory of *subsumption*[1] $\phi\ <:\ \psi$ which allows a function verified

$$\mathsf{GIVEN}\ s_0 = \{\mathrm{FuncStack}, \mathrm{Dfn}, \mathrm{Low}, \mathrm{Stack}, \mathrm{SccList}, \mathrm{VisEdge}, \mathrm{Idx}\},$$

$$\mathsf{PRE} \triangleq \begin{cases} \mathsf{graph\_rep}_{\pi_G,\pi_V}(G, s_0) * \mathsf{global\_rep}(s_0) \\ \wedge G \vdash \mathsf{start\_state}(v_0) \Rightarrow^* \begin{bmatrix} x :: \mathrm{FuncStack}, \mathrm{Dfn}[x \mapsto \mathrm{Idx}+1], \mathrm{Low}[x \mapsto \mathrm{Idx}+1] \\ x :: \mathrm{Stack}, \mathrm{Scc}, \mathrm{VisEdge}, \mathrm{Idx}+1, \mathrm{Stage1} \end{bmatrix} \\ \wedge \forall e, e.src = x \rightarrow \mathrm{VisEdge}(e) = \mathrm{unvisited} \end{cases}$$

$$\mathsf{LOOP\_INV} \triangleq \begin{cases} \mathsf{EX}\ \ (s : \mathsf{state})\ (vs\ es_1\ es_2 : \mathsf{list\ val}), \\ \mathsf{graph\_rep}_{\pi_G/2,\pi_V}(G, s) * \mathsf{global\_rep}(s) * \\ \mathsf{vertex\_storage}_{\pi_G/2,\perp}(G, vs, s_0) * \\ \mathsf{field\_storage}_{\perp}(x, s_0) * \mathsf{edge\_storage}_{\pi_G/2}(G, x, es_1 @ es_2) \\ \wedge\ \mathsf{vlis\_prop}(G, x :: vs)\ \wedge\ \mathsf{elis\_prop}(G, x, es_1 @ es_2) \\ \wedge\ G \vdash \mathsf{start\_state}(v_0) \Rightarrow^* s \wedge\ s.\mathrm{FuncStack} = x :: s_0.\mathrm{FuncStack} \\ \wedge\ \forall e \in es_1, s.\mathrm{VisEdge}(e) \neq \mathrm{unvisited}\ \ \wedge \forall e \in es_2, s.\mathrm{VisEdge}(e) = \mathrm{unvisited} \\ \wedge\ \forall e, e.src \in s_0.\mathrm{FuncStack} \rightarrow s.\mathrm{VisEdge}(e) = s_0.\mathrm{VisEdge}(e) \end{cases}$$

$$\mathsf{POST} \triangleq \begin{cases} \mathsf{EX}\ \ (s : \mathsf{state}), \\ \mathsf{graph\_rep}_{\pi_G,\pi_V}(G, s) * \mathsf{global\_rep}(s) \\ \wedge\ G \vdash \mathsf{start\_state}(v_0) \Rightarrow^* s\ \wedge\ s.\mathrm{FuncStack} = x :: s_0.\mathrm{FuncStack}\ \wedge \\ \forall e, e.src = x \rightarrow s.\mathrm{VisEdge}(e) \neq \mathrm{unvisited}\ \wedge \\ \forall e, e.src \in s_0.\mathrm{FuncStack} \rightarrow s.\mathrm{VisEdge}(e) = s_0.\mathrm{VisEdge}(e) \end{cases}$$

Figure 5. Function specification and loop invariant for C Program

$$\left\{ \begin{array}{l} \mathsf{graph\_rep}_{\pi_G,\pi_V} \\ \qquad (G, \mathrm{start\_state}(x)) * \\ \mathsf{global\_rep}(\mathrm{start\_state}(x)) \end{array} \right\}$$
$$\texttt{tarjan\_visit(x)}$$
$$\left\{ \begin{array}{l} \mathsf{EX}(s : \mathsf{state}), \mathsf{graph\_rep}_{\pi_G,\pi_V}(G, s) \\ * \ \mathsf{global\_rep}(s) \wedge s.\mathrm{SccList}\ \mathsf{is} \\ \textsc{Connected}\ \mathsf{and}\ \textsc{Maximal} \end{array} \right\}$$

Figure 6. User expected function specification of `tarjan_visit(x)`

with respect to ϕ to be used by clients expecting specification ψ.

Above, we link the proofs in the two phases together easily and prove the functional correctness of a C implementation of Tarjan's algorithm.

5 Conclusions

In this work, we first formalize Tarjan's Algorithm in an intuitive way, namely the small-step description. We present a proof paradigm (n1-induction) for proving properties on the multi-step relation. Through composition of lemmas, the functional correctness in small-step description can be proved. This part of proof is independent of the implementation, which makes our small step description portable to various implementations.

The small step description turns out to be a bridge that links the model level proof of the graph algorithm and the mechanized verification on a concrete imperative program together. We introduced some techniques, such as share accounting and specification subsumption that make proofs in VST less tedious and more intuitive. Together, the combination of small step description and half-automated proofs in VST ensures the functional correctness of the C program.

Last Updated on April 19, 2021.

References

[1] Lennart Beringer and Andrew W Appel. 2019. Abstraction and subsumption in modular verification of C programs. In *International Symposium on Formal Methods*. Springer, 573–590.

[2] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. 2005. Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 259–270.

[3] Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1, 2 (1972), 146–160.

[4] Shengyi Wang, Qinxiang Cao, Anshuman Mohan, and Aquinas Hobor. 2019. Certifying graph-manipulating C programs via localizations within data structures. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 171:1–171:30. https://doi.org/10.1145/3360597