Our Beautiful Paper

Anonymous Author(s)

Abstract

We formalize an algorithm for computing strongly connected components in a graph due to Bloemen and prove its correctness in the proof assistant Isabelle/HOL.

Keywords: graph algorithm, strongly connected component, formal verification, interactive theorem proving

1 Introduction

2 Background

2.1 Graphs and strongly connected components

2.2 Isabelle/HOL

3 Formalization of the Algorithm in Isabelle/HOL

We represent graphs with nodes of type v as a locale that fixes a finite set of vertices and a successor function associating to each node its set of successors.

```
locale graph =
  fixes   vertices :: v set
  and     successors :: v ⇒ v set
  assumes finite vertices
  and     ∀v ∈ vertices. successors v ⊆ vertices
```

The data on which the algorithm operates is collected in an "environment" record with the following components:

```
record v env =
  root :: v
  𝒮 :: v ⇒ v set
  visited :: v set
  explored :: v set
  sccs :: v set set
  stack :: v list
  cstack :: v list
  vsuccs :: v ⇒ v set
```

The *root* component stores the node for which the algorithm was originally called. The function $\mathcal{S}$ maps every node v to the set of nodes that have already been determined to be part of the same SCC as v. A core invariant of the algorithm will be that this mapping represents a partition of nodes into sets of equivalence classes: for all nodes v and w, we maintain the relationship

$$v \in \mathcal{S}\ w \ \longleftrightarrow\ \mathcal{S}\ v = \mathcal{S}\ w.$$

The sets *visited* and *explored* represent the sets of nodes that have already been seen, respectively fully explored, by the algorithm, and *sccs* is the set of (maximal) SCCs that the algorithm has found so far. Variable *stack* holds the roots of the (incomplete) SCCs in depth-first order, whereas *cstack* represents the stack of calls of the main function *dfs* of the algorithm. Finally, *vsuccs* remembers the set of outgoing edges (represented by the successor nodes) of each node that have already been followed during the exploration. Note that the variables *root* and *cstack* are not actually used by the algorithm: they are ghost variables used in the correctness proof.

The algorithm will initially be called with an environment of the form *init-env* v that holds v as the root node, initializes $\mathcal{S}$ by assigning the singleton $\{u\}$ to every node u, and has all other component initialized to the empty set or the empty list.

Environments are partially ordered according to the following relation.

```
definition sub-env where
sub-env e e' ≜
    root e' = root e
  ∧ visited e ⊆ visited e'
  ∧ explored e ⊆ explored e'
  ∧ (∀v. vsuccs e v ⊆ vsuccs e' v)
  ∧ (∀v. 𝒮 e v ⊆ 𝒮 e' v)
  ∧ (⋃ {𝒮 e v | v. v ∈ set (stack e)})
     ⊆ (⋃ {𝒮 e' v | v. v ∈ set (stack e')})
```

We follow the definition of the algorithm in [1], but represent it as a pair of mutually recursive functions *dfs* and *dfss* defined as follows.

```
function dfs  :: v ⇒ v env ⇒ v env
and      dfss :: v ⇒ v env ⇒ v env where
  dfs v e =
    let e1 = e⦇ visited := visited ∪ {v},
                stack := v # stack e,
                cstack := v # cstack e ⦈;
        e' = dfss v e1
    in  if v = hd (stack e')
        then e'⦇ sccs := sccs e' ∪ {𝒮 e' v},
```

Conference'17, July 2017, Washington, DC, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
                   explored := explored e' ∪ 𝒮 e' v,
                   stack := tl (stack e'),
                   cstack := tl (cstack e') ⦈
         else e'⦇ cstack := tl (cstack e') ⦈
| dfss v e =
     let vs = successors v - vsuccs e v
     in  if vs = {} then e
         else let w = SOME x. x ∈ vs;
                  e' = (if w ∈ explored e then e
                        else if w ∉ visited e
                             then dfs w e
                             else unite w e);
                  e'' = e'⦇ vsuccs := λx.
                              if x = v
                              then vsuccs e' v ∪ {w}
                              else vsuccs e' x⦈
              in dfss v e''
```

Function *dfs* explores a node *v* by adding it to the set of visited nodes, pushing it on both stacks, and then calling the function *dfss*. If *v* is still at the head of the stack, it is the root node of an SCC, and therefore $\mathcal{S}\ v$ is added to the set of maximal SCCs, all its nodes are marked as fully explored, and *v* is popped from both stacks. Otherwise, *v* is only popped from the call stack. The tail-recursive function *dfss* iterates over all successors *w* of node *v*. If *w* has already been fully explored, there is nothing to do. If *w* is unvisited, the function *dfs* is called for *w*, otherwise the algorithm detected a loop back to a node that has already been visited and whose root node *r* must therefore be on the stack. It unites the SCCs of all nodes contained in the prefix[1] of the stack starting at *r*, using the following auxiliary function.

```
definition unite :: v ⇒ v env ⇒ v env where
unite w e ≜
  let pfx = takeWhile (λx. w ∉ 𝒮 e x) (stack e);
      sfx = dropWhile (λx. w ∉ 𝒮 e x) (stack e);
      cc = ⋃ { 𝒮 e x | x . x ∈ set pfx ∪ {hd sfx} }
  in  e⦇ 𝒮 := λx. if x ∈ cc then cc else 𝒮 e x;
          stack := sfx ⦈
```

In words, *unite* computes the prefix of the stack consisting of the roots of SCCs that do not contain *w*. Since *w* must be represented by some node on the stack, the remaining suffix is non-empty, and $\mathcal{S}$ (*hd sfx*) contains *w*. A new connected component *cc* is computed by taking the union of the sets $\mathcal{S}\ x$ for all nodes *x* in the prefix and the head of the suffix. The mapping $\mathcal{S}$ is updated so that all nodes in *cc* are mapped to *cc*, and the stack is shortened to the suffix of the previous stack, implicitly making the head of that suffix the root of the component *cc*.

A technical complication in the definition of the algorithm is that the functions *dfs* and *dfss* need not terminate when their pre-conditions are violated. We will come back to proving termination in Sect. 5.

[1]Note that the top of the stack is at the head of the list.

4 Proof of Partial Correctness

4.1 Main invariant

We define well-formed environments to be those satisfying the following conditions, and we will prove this predicate to hold throughout the execution of the algorithm.

```
definition wf-env where
wf-env e ≜
    (∀n ∈ visited e. reachable (root e) n)
  ∧ distinct (stack e)
  ∧ distinct (cstack e)
  ∧ (∀n m. n ≼ m in stack e ⟶ n ≼ m in cstack e)
  ∧ (∀n m. n ≼ m in stack e ⟶ reachable m n)
  ∧ explored e ⊆ visited e
  ∧ set (cstack e) ⊆ visited e
  ∧ (∀n ∈ explored e. ∀m.
        reachable n m ⟶ m ∈ explored e)
  ∧ (∀n. vsuccs e n ⊆ successors n ∩ visited e)
  ∧ (∀n. n ∉ visited e ⟶ vsuccs e n = {})
  ∧ (∀n ∈ explored e. vsuccs e n = successors n)
  ∧ (∀n ∈ visited e - set (cstack e).
        vsuccs e n = successors n)
  ∧ (∀n m. m ∈ 𝒮 e n ⟷ 𝒮 e n = 𝒮 e m)
  ∧ (∀n. n ∉ visited e ⟶ 𝒮 e n = {n})
  ∧ (∀n ∈ set (stack e). ∀m ∈ set (stack e).
        n ≠ m ⟶ 𝒮 e n ∩ 𝒮 e m = {})
  ∧ ⋃ {𝒮 e n | n. n ∈ set (stack e)}
     = visited e - explored e
  ∧ (∀n in set (stack e). ∀m in 𝒮 e n.
        m ∈ set (cstack e) ⟶ m ≼ n in cstack e)
  ∧ (∀n m. n ≼ m in stack e ∧ n ≠ m ⟶
        (∀u ∈ 𝒮 e n. ¬ reachable-avoiding u m
                                         (unvisited e n)))
  ∧ (∀n. is-subscc (𝒮 e n))
  ∧ (∀S ∈ sccs e. is-scc S)
  ∧ ⋃ (sccs e) = explored e
```

In words, every visited node is reachable from the root node, the two stacks do not contain duplicate elements, and the stack is a subsequence of the call stack. Nodes on the stack are reachable from all stack nodes below them. All explored nodes, as well as all nodes on the call stack, are visited. All nodes reachable from fully explored nodes are themselves fully explored. The visited successors of a node are both successors and visited, and no outgoing edge of an unvisited node has been followed. All outgoing edges of fully explored nodes, as well as of visited nodes that are no longer on the call stack, have been followed. The mapping $\mathcal{S}$ represents an equivalence relation, as discussed above, and $\mathcal{S}\ n$ is the singleton $\{n\}$ for an unvisited node n. The nodes on the stack are roots of (non-maximal) SCCs, hence their equivalence classes are disjoint. A node has been visited but not yet fully explored iff its root node is on the stack. Moreover, root nodes of SCCs are the oldest nodes in depth-first order. No node u belonging to the SCC of a node n on the stack can reach a node m strictly below n on the stack without going through some edge from some node in n's equivalence class that has not yet been followed. In other words, the current SCCs are maximal with respect to the current knowledge of the algorithm. The set of not yet followed edges for nodes in the equivalence class of n are defined as

```
definition unvisited where
unvisited e n ≜
  {(a,b) | a b. a ∈ 𝒮 e n
               ∧ b ∈ successors a - vsuccs e a}
```

Finally, every equivalence class is a (not necessarily maximal) SCC, elements of *sccs* are maximal SCCs, and their union is the set of explored nodes.

We prove a number of simple consequences of this predicate. For example, $n \in \mathcal{S}\ n$ holds for every node. If node w is reachable from a visited node v, but not through any edge that has not yet been followed, then w must be visited. Also, edges towards fully explored nodes do not contribute to reachability of unexplored nodes avoiding certain edges:

```
lemma avoiding explored:
assumes wf-env e and reachable-avoiding x y E
    and y ∉ explored e and w ∈ explored e
shows   reachable-avoiding x y (E ∪ {(v,w)})
```

4.2 Pre- and post-conditions of functions *dfs* and *dfss*

5 Proof of Termination

6 Conclusion

References

[1] V. Bloemen. *Strong Connectivity and Shortest Paths for Checking Models.* PhD thesis, University of Twente, Enschede, The Netherlands, 2019.