15-414: Bug Catching: Automated Program Verification

Lecture Notes on Certificates: Bipartite Graphs

Matt Fredrikson

Carnegie Mellon University
Lecture 14
Tuesday, October 20, 2026

These supplementary notes develop certificates for both positive and negative answers to a graph problem. The main lecture introduces fuel with sorting. Here we apply the same technique to breadth-first search. The complete Why3 example is bipartite.mlw.

1 Certificates for Bipartite Graphs

An undirected graph $G = (V, E)$ is *bipartite* if we can partition V into two sets L and R such that every edge has one endpoint in each set. We can think of the sets as two colors assigned to vertices. Adjacent vertices must receive different colors.

A partition is already a useful certificate for a positive answer. The checker can verify that the sets contain only vertices of G, are disjoint, cover V, and place the endpoints of every edge on opposite sides. Here is the logical property we want to establish.

```
predicate proper_bipartition (g: graph) (l r: Set.set) =
  (forall v . Set.mem v g.dom ->
    (Set.mem v l <-> not (Set.mem v r))) /\
  forall u v . G.edge g u v -> (Set.mem u l <-> Set.mem v r)

predicate bipartite (g:graph) =
  exists l r:Set.set. proper_bipartition g l r
```

The first conjunct says that each vertex in the graph lies on exactly one side. The second says that edges cross sides. We define `bipartite g` by existentially quantifying the two sets. Membership outside the graph's domain is irrelevant to this logical property, although the checker rejects sets containing such extra vertices.

What can we use as a certificate for a negative answer? Consider following a path in a graph with a proper bipartition. Each edge changes sides, so after an even number of edges we are on the starting side, and after an odd number we are on the opposite side. A closed walk with an odd number of edges is therefore impossible in a bipartite graph. Such a walk certifies that the graph is not bipartite.

We represent a path by an array of vertices, including both endpoints. A closed walk has the same first and last entries. If the array has n entries, it describes $n-1$ edges. Thus we check for an odd number of edges by checking that n is even and at least 2. This also allows a self-loop, represented by two copies of the same vertex. The certificate type distinguishes the two possible answers.

```
type path   = array vertex
type bipartition = (Set.set, Set.set)

type cert =
| Bipart bipartition
| OddCycle path
```

We use the constructor name `OddCycle`, but the checker does not require the intermediate vertices to be distinct. A closed odd walk is enough for the argument above. Avoiding a distinctness check also simplifies reconstruction of the certificate.

2 Verifying the Graph Checker

To make the parity argument precise, we first require that consecutive entries in the array describe edges in the graph. We then prove the following lemma by induction on the index i.

```
predicate consecutive_edges (g:graph) (p:path) =
  length p >= 1 /\
  (forall i:int. 0 <= i < length p - 1 -> G.edge g p[i] p[i+1])

let rec lemma parity_along_path
  (g:graph) (l r:Set.set) (p:path) (i:int)
  requires { proper_bipartition g l r }
  requires { consecutive_edges g p }
  requires { length p >= 1 }
  requires { 0 <= i < length p }
  ensures  { (mod i 2 = 0) -> (Set.mem p[i] l <-> Set.mem p[0] l) }
  ensures  { (mod i 2 = 1) -> (Set.mem p[i] l <-> Set.mem p[0] r) }
  variant  { i }
=
  if i = 0 then ()
  else parity_along_path g l r p (i - 1);
```

The first postcondition concerns even indices and the second concerns odd indices. At index zero the even case is immediate. For the inductive step, the recursive call establishes the property at $i - 1$. The edge from `p[i-1]` to `p[i]` changes sides, and incrementing the index changes its parity. These facts establish the two postconditions at i.

Now suppose the path is closed and has an odd number of edges. If a proper bipartition existed, the lemma at the final index would put the final vertex on the opposite side from the initial vertex. But they are the same vertex, and the two sides are disjoint. This contradiction shows that no proper bipartition exists.

For a positive certificate, the interesting check is that neighbors of each side belong to the other side. We implement one direction at a time, with the following specification and loop.

```
predicate side_ok (g:graph) (side other:Set.set) =
  forall u v:vertex. Set.mem u side -> G.edge g u v -> Set.mem v other

let check_side_ok (g:graph) (side other:Set.set) : bool
  requires { Set.subset side g.dom /\ Set.subset other g.dom }
  ensures { result <-> side_ok g side other }
=
  let ref ok = true in
  for u in side with Set.Iter do
    invariant { ok <->
      forall v . Set.mem v it.Set.Iter.visited ->
        Set.subset (G.neighbors g v) other }
    variant { Set.cardinal side - Set.cardinal it.Set.Iter.visited }
    ok <- ok && Set.subset (G.neighbors g u) other
  done;
  ok
```

The loop uses Why3's iterator notation. The iterator records the set of vertices already visited. Our invariant says that `ok` is true exactly when the neighbor condition holds for all those vertices. Initially the visited set is empty. Each iteration adds one vertex and conjoins its neighbor test with the accumulated result. At the end, every vertex in `side` has been visited, so the invariant establishes the postcondition. The variant counts the vertices still to visit. The accompanying file supplies the set iterator interface and uses the standard library contracts for finite sets and maps.

The complete checker combines these tests. For `Bipart`, it checks membership in the domain, disjointness, coverage, and the neighbor condition in both directions. For `OddCycle`, it checks the length, closure, each consecutive edge, and parity. If a check fails, it raises an exception. Its contract is

```
let check_bp (g:graph) (c:cert) : bool
  ensures { result <-> bipartite g }
  raises { InvalidCertificate }
```

The GCD checker reports acceptance with a Boolean. The Boolean result here is the answer to the original problem. A normal return of `false` means the graph is not bipartite. It does not mean that the certificate was rejected. Rejection is reported by the exception.

3 Search and Termination

We extend breadth-first search to produce either kind of certificate. Choose a root in each connected component and place it on the left. When discovering a neighbor of a colored vertex, place it on the opposite side, enqueue it, and record its parent. If the search finishes without a conflict, the two sets form a bipartition.

If an edge connects two vertices u and v on the same side, the parent map provides paths from their common root to each endpoint. Follow the path from the root to u, the edge from u to v, and then the parent path from v back to the root. The two root paths have lengths of the same parity because their endpoints have the same color. Adding the conflicting edge gives an odd length. The walk may repeat vertices or edges, which is permitted by our checker.

The graph representation stores directed adjacency entries. For this search, we represent each undirected edge by entries in both directions. This lets us follow a parent edge back toward the root. The checker still examines every edge in a proposed walk, so a reconstruction error cannot cause an incorrect answer about the stored graph to be accepted.

We want to verify enough about this search to ensure that its operations are safe. For example, before reading a vertex's adjacency set we must know that the vertex belongs to the graph's domain. We maintain this fact for queued vertices, along with domain membership for the two color sets and the parent-map keys. We collect these requirements in `state_ok`. We do not assert that the colors form a proper bipartition or that the parent map describes the search tree.

Termination requires a little more care. Counting undiscovered vertices is not a suitable variant for the search loop: a vertex may be removed from the queue without discovering any new vertices. A proof based on the queue would also need to account for new entries and show that vertices are not enqueued repeatedly.

Instead, we pass an explicit nonnegative integer called *fuel*. Before expanding a vertex, the search checks whether fuel is zero. If so, it stops. Otherwise, it can expand the vertex and recurse with one less unit of fuel.

The variant is just `fuel`. Its decrease does not depend on the coloring, queue, or parent map being correct. The call to `scan_neighbors` terminates because it iterates over a finite set, and its contract preserves `state_ok`. We similarly bound the number of parent pointers followed while reconstructing a walk, so even a cycle in a faulty parent map cannot make reconstruction diverge.

```
let rec bfs_loop (g:graph) (s:bfs_state) (fuel:int)
  : (bfs_state, option (vertex, vertex))
  requires { fuel >= 0 }
  requires { state_ok g s }
  ensures  { let st, _ = result in state_ok g st }
  variant  { fuel }
= if fuel = 0 then (s, None) else
  match s with
  | State l r q pm ->
    match Q.deq q with
    | None -> (s, None)
    | Some (u, q1) ->
        let (l', r') =
          if Set.mem u l || Set.mem u r then (l, r)
          else (Set.add u l, r) in
        let s' = State l' r' q1 pm in
        let (s'', co) = scan_neighbors g u s' in
        match co with
        | Some (a,b) -> (s'', Some (a,b))
        | None -> bfs_loop g s'' (fuel - 1)
        end
    end
  end
```

In a correct implementation, each vertex is enqueued at most once. Starting the search with $|V| + 1$ units of fuel is therefore sufficient for each component. We use this observation to choose a reasonable budget, but we do not need to prove it to establish termination of the implementation above.

If fuel runs out early, the search may produce incomplete sets or an invalid walk. We still pass its proposed certificate to the checker. An invalid certificate causes an exception. If the checker accepts, its contract establishes the answer even though the search stopped earlier than intended. Fuel gives us termination; certificate checking gives us correctness of normal returns. Neither establishes that the solver will produce an accepted certificate for every input.