Verified Graph Algorithms in ACL2

Nathan Guermond
Kestrel Institute
November 5, 2018
Another graph library?

Goal: A unified graph library with common algorithms
Another graph library?

Goal: A unified graph library with common algorithms

- Full specifications
Another graph library?

Goal: A unified graph library with common algorithms

- Full specifications
- Modularity
Another graph library?

Goal: A unified graph library with common algorithms

- Full specifications
- Modularity
- Optimization
Core data structure

A graph is a dependent datastructure with
- (setp vertices)
- (true-listp edges)
- (booleanp directed)
A graph is a dependent datastructure with

- (setp vertices) → (get-vertices gph)
- (true-listp edges) → (get-edges gph)
- (booleanp directed) → (directed-p gph)
A graph is a dependent datastructure with
▶ (setp vertices) \rightarrow (get-vertices gph)
▶ (true-listp edges) \rightarrow (get-edges gph)
▶ (booleanp directed) \rightarrow (directed-p gph)

The dependency is given by the well-formedness constraint
▶ (graph-constraint vertices edges)
Common data structures

- \((\text{path-p } \text{pth } \text{gph})\) satisfies
  1. \((\text{true-listp pth})\) with
  2. \((\text{in (car pth) (neighbours (cadr pth) gph)})\)
  3. \((\text{path-p (cdr pth)})\)

- \((\text{rev-path-p } \text{rev-pth } \text{gph})\) satisfies
  1. \((\text{true-listp pth})\) with
  2. \((\text{in (cadr pth) (inv-neighbours (car pth) gph)})\)
  3. \((\text{rev-path-p (cdr pth)})\)

- \((\text{cycle-p } \text{cyc } \text{gph})\) is a \text{path-p} with equal ends
(find-path src tgt gph)

(defthm path-exists-implies-exists-path-spec
  (implies (and (path-p pth gph)
     (graph-p gph))
   (find-path (get-src pth) (get-tgt pth) gph)))

(defthm exists-path-implies-path-constructible-spec
  (implies (and (graph-p gph)
     (find-path src tgt gph))
   (let ((pth (find-path src tgt gph)))
     (and (path-p pth gph)
       (equal (get-src pth) src)
       (equal (get-tgt pth) tgt))))
Algorithms and specs

▶ (find-path src tgt gph)
Algorithms and specs

- (find-path src tgt gph)
- (reachable-set S gph)

(defthm exists-path-implies-reachable-spec
  (implies (and (graph-p gph)
                 (path-p pth gph))
           (in (get-tgt pth)
                (reachable-set
                 (singleton (get-src pth)) gph))))

(defthm exists-path-from-src-to-reachable-set-spec
  (implies (and (graph-p gph)
                 (in src (get-vertices gph))
                 (in tgt (reachable-set
                          (singleton src) gph)))
           (find-path src tgt gph)))
Algorithms and specs

- (find-path src tgt gph)
- (reachable-set S gph) and (inv-reachable-set S gph)
Algorithms and specs

- `(find-path src tgt gph)`
- `(reachable-set S gph)` and
  `(inv-reachable-set S gph)`
- `(find-simple-cycle gph)` and
  `(find-non-trivial-cycle gph)`
Algorithms and specs

- (find-path src tgt gph)
- (reachable-set S gph) and
  (inv-reachable-set S gph)
- (find-simple-cycle gph) and
  (find-non-trivial-cycle gph)
- (topological-sort gph)
Algorithms and specs

- (find-path src tgt gph)
- (reachable-set S gph) and (inv reachable-set S gph)
- (find-simple-cycle gph) and (find-non-trivial-cycle gph)
- (topological-sort gph)
- (get-strongly-connected-component S gph)
- (collapse-strongly-connected-components gph)

constructed from find-non-trivial-cycle, reachable-set, and inv-reachable-set

A strongly connected component is given by \((\text{Reach cyc}) \cap (\text{InvReach cyc})\)
Algorithms and specs

- (find-path src tgt gph)
- (reachable-set S gph) and
  (inv-reachable-set S gph)
- (find-simple-cycle gph) and
  (find-non-trivial-cycle gph)
- (topological-sort gph)
- (get-strongly-connected-component S gph)
- (collapse-strongly-connected-components gph)
  - constructed from find-non-trivial-cycle, reachable-set, and inv-reachable-set
  - A strongly connected component is given by (Reach cyc) ∩ (InvReach cyc)
Reachable and finite differencing

- Specification is proven by a two step refinement
  - Compute set reachable in $k$ steps
    - $S \cup (\text{Neighs } S) \cup \ldots \cup (\text{Neighs \ldots (Neighs } S))$
Reachable and finite differencing

- Specification is proven by a two step refinement
  - Compute set reachable in $k$ steps
    - $S \cup (\text{Neighs } S) \cup \ldots \cup (\text{Neighs } \ldots (\text{Neighs } S)) \ldots$
  - Compute reachable set by iterative unioning
    - $S \cup (\text{Neighs } S) \cup (\text{Neighs } (\text{Neighs } S)) \ldots$
Reachable and finite differencing

- Specification is proven by a two step refinement
  - Compute set reachable in \( k \) steps
    - \( S \cup (\text{Neighs } S) \cup \ldots \cup (\text{Neighs } (\ldots (\text{Neighs } S))\ldots) \)
  - Compute reachable set by iterative unioning
    - \( S \cup (\text{Neighs } S) \cup (\text{Neighs } (\text{Neighs } S))\ldots \)
  - Compute reachable set by finite difference
    - \( S_0 = S, S_1 = (\text{Neighs } S_0) \)
    - \( D_{i+1} = S_{i+1} - S_i, S_{i+1} = S_i \cup (\text{Neighs } D_i) \)
Applications

- Call-graphs

\[
\text{factorial}(zp) \leftarrow \begin{cases}
\text{integerp} & \text{if}\not<\text{not} \text{zp} \\
* & \text{factorial} \\
+ & \text{factorial}
\end{cases}
\]
Applications

- Call-graphs
- Guard verification

(factorial) + (zp)

> if

> integerp

> not

<

> zp

> *

> +

> factorial
Applications

- Call-graphs
- Guard verification
- Getting ordered guard obligations
Applications

- Call-graphs
- Guard verification
- Getting ordered guard obligations
- Your next project!

factorial

\[ \text{if} \quad \text{integerp} \quad \text{not} \quad < \quad * \quad + \]

\[ \text{zp} \quad \text{factorial} \]
Future work

- Prove specs for topological-sort
Future work

- Prove specs for topological-sort
- Prove specs for collapse-strongly-connected-components
Future work

- Prove specs for **topological-sort**
- Prove specs for **collapse-strongly-connected-components**
- Optimize **find-path** using finite differencing
Future work

- Prove specs for topological-sort
- Prove specs for collapse-strongly-connected-components
- Optimize find-path using finite differencing
- Optimize already specified algorithms, possibly using transformations