Graph Proofs Using the Larch Prover

Craig Struble

June 29, 1996

Abstract

Theorem provers and proof checkers are important applications of symbolic computation. Through a combination of term-rewriting and proof tactics, the computer is capable of proving or checking the proofs of some complex theorems. In order to prove a statement is true (or false), an axiomatization of facts for a given theory is needed. In this paper we explore one proof assistant, the Larch Prover, and its ability to represent an axiomatization of graph theory.

1 Proving Graph Theorems

Graphs exist in many aspects of the world. They are used to model relationships between objects in situations, e.g, people, processes, hypertext documents, etc. By using graph theory many of these relationships and problems related to these relationships can be solved. However graph theory differs from many other kinds of mathematical theories, such as algebra, number theory. Algebraic and number theory axiom systems are well defined, and proofs for these theories are typically concise. In graph theory, the axiom system is not a well defined set of relations. Rather, graph theory is rich in definitions and relationships. Also, the proofs given for graph theory often rely on diagrams, intuition, or brute force in order to prove theorems, often resulting in a proof that is not elegant, e.g., the first proof of the four-color theorem [1].

Because of this, we conjecture that coming up with a good axiomatization of graph theory is difficult to accomplish, and that many graph theorems proven are equally as difficult if not impossible to prove with any kind of theorem proving or checking system. To test this idea, the Larch Prover [3] is used as a test bed to find the difficult areas of axiomatizing graph theory, as well as a system for learning about the concepts taught in the symbolic computation class.

2 Larch Prover

Larch Prover is a multi-sorted first order logic theorem proving system. The system supports term-rewriting, Knuth-Bendix completion[4], critical-pair computation, and numerous proof techniques including induction and rule instantiation. We applied much of the knowledge from the symbolic computation class while exploring Larch Prover and graph axiom systems.

Throughout this paper, the following Larch Prover related terminology is used.

sort An abstract type used in defining parameter and variable types. For example, *Bool*, *nat*, *set* are the names for the sorts representing boolean values, natural numbers, and sets.

operator A function in term-rewriting terminology. This maps parameter sorts to some result sort. Operators can be overloaded in Larch Prover.

Larch Prover has the capability of overloading the operators defined. We use this capability to simplify the notation in the axiom systems when it is appropriate and the context is clear.

3 Graph Axiom Systems

Two approaches were considered for the axiomatization of graph theory. The first was to think of both the vertices and edges in the graph as sets, and to build on set theory to prove graph theorems. The second approach was to define vertices as being labeled using the natural numbers, where the order of the graph bounds the highest label used by a vertex, and edges are defined by an operator between two natural numbers. This approach relies more heavily on natural number theory.

The Larch prover provides both a set theory and natural number theory axiomatization. These were extended to included missing operators needed by the axiomatizations.

3.1 Set Theory Approach

For the set theory approach, we created 2 operators, $E : Graph \rightarrow Graph$, and $V : Graph \rightarrow Graph$. These return the set of edges and the set of vertices for a given graph. Unfortunately, Larch does not support having sets of a particular kind of object, so both sets contain the sort *Elem* and then some conversion operators are written to convert between an *Elem* and a *Vertex* or *Edge*. This proved to be quite tedious.

We added some operators and facts to the set axioms to fill out the necessary axioms needed to prove graph theorems. In particular the size of a set is needed and an operator *size* was written to compute the size. Also, the set axioms used left out the fact that $x \cup x = x$, so this rule was also added.

The Larch Prover description of the set theory approach is in figure 1.

```
%
% Graph Proofs in Larch - An axiomitization of graph theory using the
%   Larch Prover
%
% Copyright (c) 1996, Craig Struble
%

%
% We want natural number theory
%
execute nat

%
% Load up the set theory axioms included with Larch Prover
%
execute set2
```

```
%
% Add the size operator for sets
% Add the fact that unioning a set with itself is just the set.
%
declare variables
        t : Set
        x : Elem
        ..

declare operators
        size: Set -> Nat
        ..

assert
        size({}) = 0;
        size({x}) = 1;
        size({x} \union t) = size(t) + 1;
        t \union t = t;
        ..

%
% Start creating graph theory axioms
%
set name graphAxioms

declare sorts Graph, Vertex, Edge
declare variables
        g, g': Graph
        u,v: Vertex
        e: Edge
        ..

declare operators
        V: Graph -> Set                 % Vertex set of a graph
        E: Graph -> Set                 % Edge set of a graph
        asVertex : Elem -> Vertex       % Turn an element into a vertex
        asElem : Vertex -> Elem
        asEdge : Elem -> Edge           % Turn an element into an edge
        asElem : Edge -> Elem
        adjacent : Vertex, Vertex -> Bool       % are two vertexs adjacent
        connect : Vertex, Vertex -> Edge        % connect two vertexs
        degree : Vertex -> Nat          % degree of a vertex
        incident : Vertex, Edge -> Bool % Is an edge incident to the vertex?
        inSet : Vertex -> Set           % The set of edges incident to a vertex
```

```
        degSum : Set -> Nat              % sum of all the degrees
        aGraph : -> Graph                % Give me a particular graph
        ..

assert
        V(g) ~= {};
        ~(asElem(connect(u,u)) \in E(g));
        commutative adjacent;
        commutative connect;
                asElem(e) \in E(g) <=> \E u \E v (e = connect(u,v));
        incident(u,e) <=> asElem(e) \in inSet(u);
        asElem(connect(u,v)) \in E(g) => asElem(u) \in V(g)
                /\ asElem(v) \in V(g);
        (asElem(connect(u,v)) \in E(g)) <=> adjacent(u,v);
        asElem(connect(u,v)) \in E(g) <=> asElem(connect(u,v)) \in inSet(u);
        when V(g) = V(g'), E(g) = E(g') yield g = g';
        degree(u) = size(inSet(u));
        degSum({}) = 0;
        degSum({x} \union t) = degree(asVertex(x)) + degSum(t);
        ..

%
% Show that we have undirected graphs
%
prove asElem(connect(u,v)) \in E(g) <=> asElem(connect(v,u)) \in E(g)

prove asElem(connect(u,v)) \in inSet(u) <=> asElem(connect(v,u)) \in inSet(u)

%
% Prove an edge is incident if and only if two vertices are adjacent
%
prove incident(u,connect(u,v)) <=> adjacent(u,v)
```

Figure 1: Set Approach in Larch

3.2 Preliminary: Extending the Natural Numbers

Of all of the attempts to axiomatize theories in Larch, extending Larch's axiomatization of the natural numbers to include the relational operators was the most successful. By extending the natural numbers, and making it easier to prove some properties in the natural number approach of graph representation, the most application of the symbolic computation class concepts was made.

The Larch Prover description of the relational operators for the natural numbers is in figure 2.

```
%
% Add relational operators to the natural number axioms.
% Assumes that the natural number axioms have been executed already
```

```
%
set name nat
declare variables
        w,x,y,z : Nat
        ..

declare operators
        __ < __ : Nat, Nat -> Bool
        __ <= __ : Nat, Nat -> Bool
        __ > __ : Nat, Nat -> Bool
        __ >= __ : Nat, Nat -> Bool
        ..

register status multiset >, <, <=, >=

assert
        0 < s(x);                      % 0 is the smallest element
        s(x) < s(y) <=> x < y;
        (x <= y) <=> (x < y \/ x = y);
        (x >= y) <=> (x > y \/ x = y);
        (x <= y) <=> ~(x > y);
        (x > y) <=> (y < x);           % How do the operators relate?
        ~(0 < 0);
        ..

complete

%
% These rules causes the completion to go on forever, so put it after the
% completion.
%
assert
        (x <= y) => ((z * x) <= (z * y));
        (x < y) => (x + z) < (y + z);
        ((x < y) /\ (w < z)) => ((x + w) < (y + z));
        x < y => s(x) <= y;
        ..

%
% The transitive property
%
prove x < y /\ y < z => x < z  by induction on x
        resume by induction on y
                resume by induction on z
        resume by induction on y
```

```
                resume by induction on z
qed

%
% Another transitive property
%
prove x <= y /\ y <= z => x <= z  by induction on x
        resume by induction on y
                resume by induction on z
        resume by induction on y
                resume by induction on z
qed

%
% Prove if x is less than and equal to y and x is greater than and equal to y
% that x and y are equal
%
prove (x <= y) /\ (x >= y) => (x = y) by induction on x
        resume by induction on y
        resume by induction on y
qed

%
% No natural number is less than zero
%
prove ~(x < 0) by induction on x
qed

%
% No natural number is less than itself
%
prove ~(x < x) by induction on x
qed
```

Figure 2: Extending the Natural Numbers in Larch

3.3 Natural Number Theory Approach

The natural number theory approach first added a number of axioms to the natural numbers to define the relational operators $<, >, \leq, \geq$. It was not necessary to define $=$ as Larch Prover defines it for every sort defined. From these relationships the operator $\in$: $nat \rightarrow Graph$ was related to the order of the graph.

The Larch Prover description of the natural number theory approach is shown in figure 3.

```
%
% In this attempt, we will assume all nodes are labelled by natural
```

```
% numbers. The order of the graph will be the largest label in the graph.
%

%
% Give us the natural numbers
%
execute nat

% Add relational operators
%
execute natextras

%
% Ok, now let's do a little bit of graph stuff
%
set name graph
declare sort Graph
declare sort Edge
declare variables
        g,h : Graph
        a,b : Edge
        ..

declare operators
        order : Graph -> Nat            % Number of nodes in a graph
        __ \in __ : Nat, Graph -> Bool  % Is the vertex in the graph?
        __ \in __ : Edge, Graph -> Bool % Is the edge in the graph?
        insert : Edge, Graph -> Graph   %
        __ . __ : Nat, Nat -> Edge      % The two vertices are connected
        deg : Nat -> Nat                % The degree of the vertex
        degSum : Graph -> Nat           % The sum of the degrees of vertices
        sum : Nat, Graph -> Nat         % A helper sum function
        size : Graph -> Nat             % The number of edges in the graph
        ..

assert
        commutative .;
        order(g) > 0;              % The order of a graph is greater than 0
        x \in g <=> x < order(g);  % A vertex is in G if its label is less than
                                   % the order
        ~(x \in g) => ~(x.y \in g); % If a vertex isn't in the graph, neither
                                    % can the edge
        a \in g <=> ((a = x.y) /\ (x.y \in g));
        ~(x.x \in g);              % There are no loops.
        ..
```

```
critical-pairs graph with nat

%
% Prove the vertex labelled 0 is always in a graph
%
prove 0 \in g
qed

%
% Prove undirected graphs by showing if an edge is in the graph, so
% is the edge written in reverse.
%
prove x.y \in g => y.x \in g
qed

%
% Prove that the sum of the degrees of the vertices is less than or
% equal to the order of the graph times the order of the graph
%
set name graphSize

assert
        sum(0, g) = deg(0);                  % Base case
        ~(x \in g) => deg(x) = 0;            % Any natural not a vertex
                                             % has degree of 0
        sum(s(x), g) = sum(x, g) + deg(x);   % sum is current plus sum of
                                             % previous
        degSum(g) = sum(order(g), g);        % Total sum is sum from order(g)
        deg(x) < order(g);                   % Degree must be less than order.
        ..


%
% Prove that the sum of the degrees of x vertices is < (x + 1) * order(g)
%
prove sum(x,g) < (s(x) * order(g)) by induction on x
        instantiate x by deg(xc), y by order(g), w by sum(xc, g), z by order(g)* s(xc) in nat.23
qed

%
% Prove that the sum of the degrees of x vertices plus the degree of
% the (non-existant) vertex labeled order(g) is just the sum
%
prove sum(x,g) + deg(order(g)) = sum(x,g)
```

```
        instantiate x by order(g) in graphSize.2
qed
```

Figure 3: Natural Number Approach in Larch

4 Proof Attempts

Some very simple proofs for graph theory succeeded. The theorem that a graph was undirected is trivial to prove because the axioms basically enforce that so Larch proved that by normalization. For the natural number theory approach, the theorem that a vertex labeled 0 is always in a graph was also proven trivially once enough of the axioms and proofs for the relations on the natural numbers was proved.

More complex proofs failed though. For the set theoretic approach, trying to prove that the number of edges in a graph is equal to half the sum of the degrees of the vertices failed. For the natural number approach, trying to prove that the sum of the degrees of the vertices was $< order(g)^2$ where g is a graph also failed, but more progress toward the result was made.

5 Why Proofs Failed

Some of the reasons proofs failed are the result of limitations of Larch Prover. Others reasons are based on our lack of experience with Larch Prover. After attempting to come up with a general axiomatization for graph theory and then viewing Rob Ellis's presentation [2] in the symbolic computation class, it seems that general axiomatization for graph theory is not the right approach to apply Larch to graph theorems. Larch is better used to assert well known theorems and axioms, and then use them to prove a specific conjecture.

The key observation while using Larch Prover is that it is a proof assistant and not an automated theorem prover. It is useful for performing tedious tasks such as setting up an inductive proof, or applying theorems with substitutions.

6 Suggestions to Improve Larch Prover

Larch has a number of limitations that affected how the axioms were presented.

It is difficult to express numeric relationships between elements. While the size of a sort may be computed, trying to prove something by induction on the size is not possible because you cannot induct on an operator. Allowing some way of inducting on the value returned by an operator would be a useful enhancement to Larch.

Also, you cannot instantiate axioms for a particular sort. For example, with the set theoretic approach, it would have been much easier to say that the sets contained particular types of elements instead of needing operators to convert between the sorts. Adding the ability to instantiate an axiom system for a particular sort would allow for an object oriented style development of axiom systems. An axiom system could be a template of axioms, and then specifically instantiated for a sort in the appropriate situation.

7 Summary

Even though the proposed goals were not achieved, learning about Larch Prover as a proof assistant, term-rewriting system, and symbolic computation system was very beneficial. Continued use of Larch might provide more insight to a correct axiomatization or approach to applying Larch to graph theorems.

References

[1] K. Appel and W. Haken, *The solution of the four-color map problem*, Scientific American, (1977), pp. 108–121.

[2] R. Ellis, *Using the larch prover to derive relationships for cubical complexes.* Class Presentation, June 1996.

[3] S. J. Garland and J. V. Guttag, *Larch Prover: Online Help*, MIT Laboratory for Computer Science.

[4] D. E. Knuth and P. B. Bendix, *Simple word problems in universal algebras*, in Computational Problems in Abstract Algebra, J. Leech, ed., Pergamon Press, 1970, pp. 263–297.