A PVS Graph Theory Library

Ricky W. Butler
Langley Research Center, Hampton, Virginia

Jon A Sjogren
Air Force Office of Scientific Research, Washington, DC

February 1998
Abstract

This paper documents the NASA Langley PVS graph theory library. The library provides fundamental definitions for graphs, subgraphs, walks, paths, subgraphs generated by walks, trees, cycles, degree, separating sets, and four notions of connectedness. Theorems provided include Ramsey’s and Menger’s and the equivalence of all four notions of connectedness.

Contents

1 Introduction .......................... 2
2 Definition of a Graph .................. 2
3 Graph Operations ...................... 5
4 Graph Degree .......................... 6
5 Subgraphs ................................ 7
6 Walks and Paths ....................... 8
7 Connected Graphs ..................... 13
8 Circuits ................................ 14
9 Trees .................................. 15
10 Ramsey’s Theorem ..................... 15
11 Menger’s Theorem ..................... 16
12 PVS Theories ......................... 18
13 Concluding Remarks ................... 20
A APPENDIX: Other Supporting Theories 20
A.1 Graph Inductions .................... 20
A.2 Subgraphs Generated From Walks .... 21
A.3 Maximum Subgraphs .................. 21
A.4 Minimum Walks ...................... 22
A.5 Abstract Min and Max Theories .... 23
1 Introduction

This paper documents the NASA Langley PVS graph theory library. The library develops the fundamental concepts and properties of finite graphs.

2 Definition of a Graph

The standard mathematical definition of a graph is that it is an ordered pair of sets \((V,E)\) such that \(E\) is a subset of the ordered pairs of \(V\). Typically \(V\) and \(E\) are assumed to be finite though sometimes infinite graphs are treated as well. The NASA library is restricted to finite graphs only. The set \(V\) is called the vertices of the graph and the set \(E\) is called the edges of the graph.

Although PVS directly supports ordered pairs, we have chosen the PVS record structure to define a graph. The advantage of the record structure is that it provides names for the vertex and edge sets rather than \text{proj} \_1 and \text{proj} \_2. For efficiency reasons, it is preferable to define a graph in PVS in two steps. We begin with the definition of a pregraph:

\[
\text{pregraph: TYPE} = \langle \text{vert : finite_set}[T], \text{edges: finite_set}[\text{doubleton}[T]] \rangle
\]

A pregraph is a structured type with two components: \text{vert} and \text{edges}. The \text{vert} component is a finite set over an arbitrary type \(T\). This represents the vertices of the graph. The \text{edges} component is a finite set of doubletons (i.e. sets with exactly two members) of \(T\). Thus, an edge is defined by designating its two end vertices. The type \text{finite_set} is defined in the PVS finite sets library. It is a subtype of the type \text{set} which is defined in the PVS prelude as follows:

\[
\text{sets [T: TYPE]: THEORY}
\begin{align*}
\text{set: TYPE} & = [T \rightarrow \text{bool}] \\
\text{x, y: VAR T} \\
\text{a, b, c: VAR set} \\
\text{p: VAR [T \rightarrow \text{bool}]} \\
\text{member(x, a): bool} & = a(x) \\
\text{emptyset: set} & = x | \text{false} \\
\text{subset?(a, b): bool} & = (\text{FORALL} x: \text{member}(x, a) \Rightarrow \text{member}(x, b)) \\
\text{union(a, b): set} & = x | \text{member}(x, a) \text{ OR member}(x, b) \\
\text{intersection(a, b): set} & = x | \text{member}(x, a) \text{ AND member}(x, b)
\end{align*}
\]

\text{END sets}

A set is just a boolean-valued function of the element type, i.e., a function from \(T\) into \text{bool}. In PVS this is written as \([T \rightarrow \text{bool}]\). If \(x\) is a member of a set \(S\), the expression \(S(x)\) evaluates to \text{true}, otherwise it evaluates to \text{false}.

Finite sets are defined as follows:
S: VAR set[T]

is_finite(S): bool = (EXISTS (N: nat, f: [(S) -> below[N]]): injective?(f))

finite_set: TYPE = { S | is_finite(S) } CONTAINING emptyset[T]

Thus finite sets are sets which can be mapped onto 0...N for some N. The cardinality function card is defined as follows:

inj_set(S): (nonempty?[nat]) =
    { n | EXISTS (f : [(S)->below[n]]): injective?(f) }

card(S): nat = min(inj_set(S))

All of the standard properties about card have been proved and are available:

card_union : THEOREM card(union(A,B)) = card(A) + card(B) -
    card(intersection(A,B))

card_add : THEOREM card(add(x,S)) = card(S) +
    IF S(x) THEN 0 ELSE 1 ENDIF

card_remove : THEOREM card(remove(x,S)) = card(S) -
    IF S(x) THEN 1 ELSE 0 ENDIF

card_subset : THEOREM subset?(A,B) IMPLIES card(A) <= card(B)

card_emptyset : THEOREM card(emptyset[T]) = 0

card_singleton: THEOREM card(singleton(x)) = 1

Now we are ready to define a graph as follows:

graph: TYPE = {g: pregraph | (FORALL (e: doubleton[T]): edges(g)(e) IMPLIES
    (FORALL (x: T): e(x) IMPLIES vert(g)(x))) }

A graph is a pregraph where the edges set contains doubleton sets with elements restricted to the vert set. The doubleton type is defined as follows:

doubletons[T: TYPE]: THEORY
BEGIN

x,y,z: VAR T

dbl(x,y): set[T] = {t: T | t = x OR t = y}

3
S: VAR set[T]

doubleton?(S): bool = (EXISTS x,y: x /= y AND S = dbl(x,y))

doubleton: TYPE = {S | EXISTS x,y: x /= y AND S = dbl(x,y)}
END doubletons

For example, suppose the base type T is defined as follows:

T: TYPE = {a,b,c,d,e,f,g}

Then the following pregraph is also a graph:

(# vert := {a,b,c},
 edges := { {a,b}, {b,c} } #)

whereas

(# vert := {a,b,c},
 edges := { {a,b}, {b,d}, {a,g} } #)

is a pregraph but is not a graph.\footnote{PVS does not allow \{ .. \} as set constructors. These must be constructed in PVS using LAMBDA expressions or through use of the functions add, emptyset, etc.}

The size of a graph is defined as follows:

size(G): nat = card[T](vert(G))

A singleton graph with one vertex x (i.e. size is 1) can be constructed using the following function:

singleton_graph(v): graph = (# vert := singleton[T](v),
 edges := emptyset[doubleton[T]] #)

For convenience we define a number of predicates:

edge?(G)(x,y): bool = x /= y AND edges(G)(dbl[T](x,y))

empty?(G): bool = empty?(vert(G))

singleton?(G): bool = (size(G) = 1)

isolated?(G): bool = empty?(edges(G))

The net result is that we have the following:
vertices of graph $G$ (a finite set of $T$)
edges of a graph $G$ (a finite set of doubletons taken from $\text{vert}(G)$)
true IFF there is an edge between vertices $x$ and $y$
true IFF the graph $G$ has no vertices
true IFF graph $G$ has only 1 vertex
true IFF graph $G$ has no edges

The following useful lemmas are provided:

$x, y, v$: VAR $T$
e: VAR doubleton[$T$]
G: var graph

definitions and lemmas are located in the $\text{graphs}$ theory.

3 Graph Operations

The theory $\text{graph.ops}$ defines the following operations on a graph:

$\text{union}(G_1, G_2)$ creates a graph that is a union of $G_1$ and $G_2$
$\text{del_vert}(G, v)$ removes vertex $v$ and all adjacent edges to $v$ from the graph $G$
$\text{del_edge}(e, G)$ creates subgraph with edge $e$ removed from $G$

These operations are defined as follows:

$\text{union}(G_1, G_2): \text{graph}[T] = ($
  (# $\text{vert} := \text{union}(\text{vert}(G_1), \text{vert}(G_2)),$
   $\text{edges} := \text{union}(\text{edges}(G_1), \text{edges}(G_2))$ #)

$\text{del_vert}(G: \text{graph}[T], v: T): \text{graph}[T] =$
  (# $\text{vert} := \text{remove}[T](v, \text{vert}(G)),$
   $\text{edges} := e \mid \text{edges}(G)(e) \text{ AND NOT } e(v)$ #)

$\text{del_edge}(G, e): \text{graph}[T] = G$ WITH [$\text{edges} := \text{remove}(e, \text{edges}(G))$]

The following is a partial list of the properties that have been proved:
del_vert_still_in  : LEMMA FORALL (x: (vert(G))):
           x /= v IMPLIES vert(del_vert(G, v))(x)

size_del_vert   : LEMMA FORALL (v: (vert(G))):
           size(del_vert(G, v)) = size(G) - 1

edge_in_del_vert : LEMMA (edges(G)(e) AND NOT e(v)) IMPLIES
           edges(del_vert(G, v))(e)

del_vert_comm    : LEMMA del_vert(del_vert(G, x), v) =
           del_vert(del_vert(G, v), x)

del_edge_lem3    : LEMMA edges(G)(e2) AND e2 /= e IMPLIES
           edges(del_edge(G, e))(e2)

vert_del_edge    : LEMMA vert(del_edge(G, e)) = vert(G)

del_vert_edge_comm : LEMMA del_vert(del_edge(G, e), v) =
           del_edge(del_vert(G, v), e)

4 Graph Degree

The theory graph_deg develops the concept of degree of a vertex. The following functions
are defined:

incident_edges(v, G) returns set of edges attached to vertex v in graph G
deg(v, G) number of edges attached to vertex v in graph G

Formally they are specified as follows:

v: VAR T
G,GS: VAR graph[T]

incident_edges(v, G) : finite_set[doubleton[T]]
           = {e: doubleton[T] | edges(G)(e) AND e(v) }

deg(v, G): nat = card(incident_edges(v, G))

The following useful properties are proved

deg_del_edge       : LEMMA e = dbl(x,y) AND edges(G)(e) IMPLIES
           deg(y, G) = deg(y, del_edge(G, e)) + 1

deg_edge_exists   : LEMMA deg(v, G) > 0 IMPLIES
           (EXISTS e: e(v) AND edges(G)(e))
deg_to_card : LEMMA \( \text{deg}(v, G) > 0 \) IMPLIES \( \text{size}(G) \geq 2 \)

del_vert_deg_0 : LEMMA \( \text{deg}(v, G) = 0 \) IMPLIES \( \text{edges}(\text{del_vert}(G, v)) = \text{edges}(G) \)

deg_del_vert : LEMMA \( x \neq v \) AND \( \text{edges}(G)(\text{dbl}[T](x, v)) \) IMPLIES \( \text{deg}(v, \text{del_vert}(G, x)) = \text{deg}(v, G) - 1 \)

del_vert_not_incident: LEMMA \( x \neq v \) AND NOT \( \text{edges}(G)(\text{dbl}[T](x, v)) \) IMPLIES \( \text{deg}(x, \text{del_vert}(G, v)) = \text{deg}(x, G) \)

singleton_deg: LEMMA \( \text{singleton?}(G) \) IMPLIES \( \text{deg}(v, G) = 0 \)

5 Subgraphs

The subgraph relation is defined as a predicate named \( \text{subgraph}? \):

\[ G_1, G_2: \text{VAR \ graph}[T] \]

\[ \text{subgraph}? (G_1, G_2): \text{bool} = \text{subset?}(\text{vert}(G_1), \text{vert}(G_2)) \text{ AND} \]

\[ \text{subset?}(\text{edges}(G_1), \text{edges}(G_2)) \]

The subgraph type is defined using this predicate:

\[ \text{Subgraph}(G: \text{graph}[T]): \text{TYPE} = \{ S: \text{graph}[T] \mid \text{subgraph?}(S, G) \} \]

The subgraph generated by a vertex set is defined as follows:

\[ i: \text{VAR \ T} \]

\[ e: \text{VAR \ doubleton}[T] \]

\[ \text{subgraph}(G, V): \text{Subgraph}(G) = \]

\[ (G \text{ WITH } \text{vert} := \{ i \mid \text{vert}(G)(i) \text{ AND } V(i) \}, \]

\[ \text{edges} := \{ e \mid \text{edges}(G)(e) \text{ AND} \]

\[ \text{(FORALL } x: T): e(x) \text{ IMPLIES } V(x) \} ] \]

The following properties have been proved:

\( \text{finite_vert_subset} : \text{LEMMA \ is_finite}(\text{LAMBDA } i: T): \text{vert}(G)(i) \text{ AND } V(i)) \)

\( \text{subgraph_vert_sub} : \text{LEMMA} \text{ subgraph?}(V, \text{vert}(G)) \text{ IMPLIES} \)

\[ \text{ vert}(\text{subgraph}(G, V)) = V \]
subgraph_lem : LEMMA subgraph?(subgraph(G,V),G)

SS: VAR graph[T]
subgraph_smaller : LEMMA subgraph?(SS, G) IMPLIES
    size(SS) <= size(G)

These definitions and lemmas are located in the subgraphs theory.

6 Walks and Paths

Walks are defined using finite sequences which are defined in the seq_def theory:

seq_def[T: TYPE]: THEORY
BEGIN
    finite_seq: TYPE = [# l: nat, seq: [below[l] -> T] #]
END

We begin by defining a prewalk as follows:

prewalk: TYPE = {w: finite_seq[T] | l(w) > 0}

where, as before, T is the base type of vertices. A prewalk is a finite sequence of vertices. Thus, if we make the declaration:

w: VAR prewalk

l(w) is the length of the prewalk and seq(w)(i) is the ith element in the sequence. Prewalks are constrained to be greater than 1 in length. We have used the PVS conversion mechanism, so that w(i) can be written instead of seq(w)(i). A walk is then defined as follows:

s,ps,ww: VAR prewalk

verts_in?(G,s): bool = (FORALL (i: below(l(s))): vert(G)(seq(s)(i)))

walk?(G,ps): bool = verts_in?(G,ps) AND
    (FORALL n: n < l(ps) - 1 IMPLIES
        edge?(G)(ps(n),ps(n+1)))

Seq(G) : TYPE = {w: prewalk | verts_in?(G,w)}
Walk(G): TYPE = {w: prewalk | walk?(G,w)}
A walk is just a prewalk where all of the vertices are in the graph and there is an edge between each consecutive element of the sequence. The dependent type \( \text{Walk}(G) \) defines the domain (or type) of all walks in a graph \( G \). The dependent type \( \text{Seq}(G) \) defines the domain (or type) of all prewalks in a particular graph \( G \).

The predicates \( \text{from?} \) and \( \text{walk_from?} \) identify sequences and walks from one particular vertex to another.

\[
\text{from?}(ps,u,v): \text{bool} = \text{seq}(ps)(0) = u \text{ AND } \text{seq}(ps)(l(ps) - 1) = v
\]

\[
\text{walk_from?}(G,ps,u,v): \text{bool} = \\
\text{seq}(ps)(0) = u \text{ AND } \text{seq}(ps)(l(ps) - 1) = v \text{ AND } \text{walk?}(G,ps)
\]

The function \( \text{verts_of} \) returns the set of vertices that are in a walk:

\[
\text{verts_of}(ww: \text{prewalk}): \text{finite_set}[T] = \\
\{ t: T \mid (\exists (i: \text{below}(l(ww))): \text{ww}(i) = t) \}
\]

Similarly, the function \( \text{edges_of} \) returns the set of edges that are in a walk:

\[
\text{edges_of}(ww): \text{finite_set}[\text{doubleton}[T]] = \{ e: \text{doubleton}[T] \mid \\
(\exists (i: \text{below}(l(ww)-1)): e = \text{dbl}(\text{ww}(i),\text{ww}(i+1))) \}
\]

Below are listed some of the proved properties about walks:

\[
G,GG: \text{VAR graph}[T] \\
x,u,v: \text{VAR } T \\
i,j,n: \text{VAR nat} \\
ps: \text{VAR prewalk}
\]

\[
\text{verts_in?_concat}: \text{LEMMA FORALL (s1,s2: Seq(G)): verts_in?(G,s1 o s2)}
\]

\[
\text{verts_in?_caret}: \text{LEMMA FORALL (j: below(l(ps))): i <= j IMPLIES} \\
\text{verts_in?}(G,ps) \text{ IMPLIES verts_in?}(G,ps^*(i,j))
\]

\[
\text{vert_seq_lem}: \text{LEMMA FORALL (w: Seq(G)): n < l(w) IMPLIES vert}(G)(w(n))
\]

\[
\text{verts_of_subset}: \text{LEMMA FORALL (w: Seq(G)): subset?(verts_of(w),vert(G))}
\]

\[
\text{edges_of_subset}: \text{LEMMA walk?}(G,ps) \text{ IMPLIES subset?}(\text{edges_of}(ps),\text{edges}(G))
\]

\[
\text{walk_verts_in}: \text{LEMMA walk?}(G,ps) \text{ IMPLIES verts_in?}(G,ps)
\]

\[
\text{walk_from_vert}: \text{LEMMA FORALL (w: prewalk,v1,v2:T):}
\]
walk_from?(G,w,v1,v2) IMPLIES
vert(G)(v1) AND vert(G)(v2)

walk_edge_in : LEMMA walk?(G,ps) AND
subset?(edges_of(ps),edges(GG)) AND
subset?(verts_of(ps),vert(GG))
IMPLIES walk?(GG,ps)

The walks theory also proves some useful operators for walks:

- gen_seq1(G,u) create a prewalk of length 1 consisting of a single vertex u
- gen_seq2(G,u,v) create a prewalk of length 2 from u to v
- trunc(p) return a prewalk equal to p except the last vertex has been removed
- addl(p,v) return a prewalk equal to p except the vertex v has been added
- rev(p) return a finite sequence that is the reverse of p
- o^(m,n) concatenates two finite sequences
- returns a finite sequence from the m .. n elements of a sequence. For example if p = v0 -> v1 -> v2 -> v3 -> v4, then p^(1,2) = v1 -> v2.

These are defined formally as follows:

\[
\text{gen_seq1}(G, (u: (\text{vert}(G)))): \text{Seq}(G) = \\
(\# 1 := 1, \text{seq} := (\text{LAMBDA} (i: \text{below}(1)): u) \#)
\]

\[
\text{gen_seq2}(G, (u,v: (\text{vert}(G)))): \text{Seq}(G) = \\
(\# 1 := 2, \\
\text{seq} := (\text{LAMBDA} (i: \text{below}(2)): \\
\quad \text{IF } i = 0 \text{ THEN } u \text{ ELSE } v \text{ ENDIF}) \#)
\]

\[
\text{Longprewalk}: \text{TYPE} = \{ps: \text{prewalk} \mid l(ps) \geq 2\}
\]

\[
\text{trunc}(p: \text{Longprewalk }): \text{prewalk} = p^{(0,l(p)-2)}
\]

\[
\text{addl}(ww,x): \text{prewalk} = (# 1 := l(ww) + 1, \\
\quad \text{seq} := (\text{LAMBDA} (ii: \text{below}(l(ww) + 1)): \\
\quad \quad \text{IF } ii < l(ww) \text{ THEN } \text{seq}(ww)(ii) \text{ ELSE } x \text{ ENDIF}) \#)
\]

\[
\text{fs, fs1, fs2, fs3: VAR finite_seq}
\]

\[
m, n: \text{VAR nat}
\]

\[
\text{o}(fs1, fs2): \text{finite_seq} = \\
\text{LET } l1 = l(fs1), \\
lsum = l1 + l(fs2) \\
\text{IN} (# 1 := lsum, \\
\text{seq} := (\text{LAMBDA} (n: \text{below}[lsum])):
\]

10
IF n < l
    THEN seq(fs1)(n)
    ELSE seq(fs2)(n-l)
ENDIF #);

emptyarr(x: below[0]): T
emptyseq: fin_seq(0) = (# 1 := 0, seq := emptyarr #) ;

p: VAR [nat, nat] ;

^(fs: finite_seq, (p: [nat, below(l(fs))])): 
    fin_seq(IF proj_1(p) > proj_2(p) THEN 0
              ELSE proj_2(p)-proj_1(p)+1 ENDIF) =
    LET (m, n) = p
    IN IF m > n
        THEN emptyseq
        ELSE (# 1 := n-m+1,
              seq := (LAMBDA (x: below[n-m+1]): seq(fs)(x + m)) #)
        ENDIF ;

rev(fs): finite_seq = (# 1 := l(fs),
                 seq := (LAMBDA (i: below(l(fs))): seq(fs)(l(fs)-1-i)) #)

The following is a partial list of the proven properties about walks:

gen_seq1_is_walk: LEMMA vert(G)(x) IMPLIES walk?(G,gen_seq1(G,x))

edge_to_walk : LEMMA u /= v AND edges(G)(edg[T](u, v)) IMPLIES
               walk?(G,gen_seq2(G,u,v))

walk?-addl : LEMMA walk?(G,ww) AND vert(G)(x)
            AND edge?(G)(seq(ww)(l(ww)-1),x)
            IMPLIES walk?(G,add1(ww,x))

walk?-rev : LEMMA walk?(G,ps) IMPLIES walk?(G,rev(ps))

walk?-caret : LEMMA i <= j AND j < l(ps) AND walk?(G,ps)
              IMPLIES walk?(G,ps^((i,j))

yt: VAR T
p1,p2: VAR prewalk
walk_merge: LEMMA walk_from?(G, p1, v, yt) AND
walk_from?(G, p2, u, yt)
IMPLIES
(EXISTS (p: prewalk): walk_from?(G, p, u, v))

A path is a walk that does not encounter the same vertex more than once. The predicate path? identifies paths:

ps: VAR prewalk

path?(G,ps): bool = walk?(G,ps) AND (FORALL (i,j: below(l(ps))):
    i /= j IMPLIES ps(i) /= ps(j))

Similarly the predicate path_from? identifies paths from vertex s to t:

path_from?(G,ps,s,t): bool = path?(G,ps) AND from?(ps,s,t)

Corresponding dependent types are defined:

Path(G): TYPE = {p: prewalk | path?(G,p)}

Path_from(G,s,t): TYPE = {p: prewalk | path_from?(G,p,s,t) }

The following is a partial list of proven properties:

G: VAR graph[T]
x,y,s,t: VAR T
i,j: VAR nat
p,ps: VAR prewalk

path?_caret : LEMMA i <= j AND j < l(ps) AND path?(G,ps)
    IMPLIES path?(G,ps^(i,j))

path_from?_caret: LEMMA i <= j AND j < l(ps) AND path_from?(G, ps, s, t)
    IMPLIES path_from?(G, ps^(i,j),seq(ps)(i),seq(ps)(j))

path?_rev : LEMMA path?(G,ps) IMPLIES path?(G,rev(ps))

path?_gen_seq2 : LEMMA vert(G)(x) AND vert(G)(y) AND
    edge?(G)(x,y) IMPLIES path?(G,gen_seq2(G,x,y))

path?_add1 : LEMMA path?(G,p) AND vert(G)(x)
    AND edge?(G)(seq(p)(l(p)-1),x)
AND NOT verts_of(p)(x)
IMPLIES path?(G,addl(p,x))

path_trunc1 : LEMMA path?(G,p) AND l(p) > 1 IMPLIES
path_from?(G,trunc1(p),seq(p)(0),seq(p)(l(p)-2))

These definitions and lemmas about paths are located in the paths theory.

7 Connected Graphs

The library provides four different definitions for connectedness of a graph and provides proofs that they are all equivalent. These are named connected, path_connected, piece_connected, and completed:

G,G1,G2,H1,H2: VAR graph[T]

connected?(G): RECURSIVE bool = singleton?(G) OR
(EXISTS (v: (vert(G))): deg(v,G) > 0
AND connected?(del_vert(G,v)))
MEASURE size(G)

path_connected?(G): bool = NOT empty?(G) AND
(FORALL (x,y: (vert(G))):
(EXISTS (w: Walk(G)): seq(w)(0) = x AND
seq(w)(l(w)-1) = y))

piece_connected?(G): bool = NOT empty?(G) AND
(FORALL H1,H2: G = union(H1,H2) AND
NOT empty?(H1) AND NOT empty?(H2)
IMPLIES NOT empty?(intersection(vert(H1),
vert(H2))))

completed?(G): bool = IF isolated?(G) THEN singleton?(G)
ELSIF (EXISTS (v: (vert(G))): deg(v,G) = 1) THEN
(EXISTS (x: (vert(G))): deg(x,G) = 1 AND
connected?(del_vert(G,x)))
ELSE
(EXISTS (e: (edges(G))):
connected?(del_edge(G,e)))
ENDIF

These definitions are located in the graph_conn_defs theory. The following lemmas about equivalence are located in the theory graph_connected:

graph_connected[T: TYPE]: THEORY
8 Circuits

A slightly non-traditional definition of circuit is used. A circuit is a walk that starts and ends in the same place (i.e. a pre_circuit) and is cyclically reduced (i.e. cyclically_reduced?).

reducible?(G: graph[T], w: Seq(G)): bool = (EXISTS (k: posnat): k < 
1(w) - 1 AND w(k-1) = w(k+1))

reduced?(G: graph[T], w: Seq(G)): bool = NOT reducible?(G,w)

cyclically_reduced?(G: graph[T], w: Seq(G)): bool = l(w) > 2 AND 
reduced?(G,w) AND w(1) /= w(l(w)-2)

pre_circuit?(G: graph[T], w: prewalk): bool = walk?(G,w) AND 
   w(0) = w(l(w)-l)

circuit?(G: graph[T], w: Seq(G)): bool = walk?(G,w) AND 
   cyclically_reduced?(G,w) AND 
   pre_circuit?(G,w)

The following properties are proved in the circuit_deg theory:

cir_deg_G : LEMMA (EXISTS (a,b: (vert(G))): vert(G)(z) AND 
a /= b AND edge?(G)(a,z) AND edge?(G)(b,z) ) IMPLIES 
deg(z,G) >= 2

circuit_deg : LEMMA FORALL (w: Walk(G),i: below(l(w))): circuit?(G,w) 
IMPLIES deg(w(i),G_from(G,w)) >= 2
9 Trees

Trees are defined recursively as follows:

\[ G: \text{VAR Graph[T]} \]

\[
\text{tree?(G): RECURSIVE bool = card[T](\text{vert}(G)) = 1 \text{ OR } } \\
(\exists (v: \text{vert}(G)): \deg(v, G) = 1 \text{ AND } \\
\text{tree?(del_vert[T](G,v))})
\]

\[ \text{MEASURE size(G)} \]

and the Tree type is defined as follows:

\[ \text{Tree: TYPE = \{G: graph[T] | tree?(G)\}} \]

The fundamental property that trees have no circuits is proved in \text{tree_circ} theory.

\[ \text{tree_no_circuits: THEOREM (FORALL (w: Walk(G)): tree?(G) => NOT circuit?(G,w))} \]

10 Ramsey’s Theorem

This work builds upon a verification of this theorem by Natarajan Shankar and the paper entitled “The Boyer-Moore Prover and Nuprl: An Experimental Comparison” by David Basin and Matt Kaufmann\(^2\).

\[ i, j: \text{VAR T} \]
\[ n, p, q, ii: \text{VAR nat} \]
\[ g: \text{VAR graph[T]} \]
\[ G: \text{VAR Graph[T]} \text{ % nonempty} \]
\[ V: \text{VAR finite_set[T]} \]

\[ \text{contains_clique(g, n): bool = } \\
(\exists (C: \text{finite_set[T]}): \\
\text{subset?(C,vert(g)) AND card(C) \geq n AND } \\
(\forall i,j: i \neq j \text{ AND C(i) AND C(j) IMPLIES edge?(g)(i,j)}) \]

\[ \text{contains_indep(g, n): bool = } \\
(\exists (D: \text{finite_set[T]}): \\
\]

subset?(D, vert(g)) AND card(D) >= n AND
(FORALL i, j: i/=j AND D(i) AND D(j) IMPLIES NOT edge?(g)(i, j))

subgraph_clique: LEMMA (FORALL (V: set[T]):
contains_clique(subgraph(g, V), p)
IMPLIES contains_clique(g, p))

subgraph_indep : LEMMA (FORALL (V: set[T]):
contains_indep(subgraph(g, V), p)
IMPLIES contains_indep(g, p))

ramseys_theorem: THEOREM (EXISTS (n: posnat):
(FORALL (G: Graph[T]): size(G) >= n
IMPLIES (contains_clique(G, 11) OR
contains_indep(G, 12)))))

11 Menger’s Theorem

To state menger’s theorem one must first define minimum separating sets. This is fairly complicated in a formal system. We begin with the concept of a separating set:

G: VAR graph[T]
v,s,t: VAR T
e: VAR doubleton[T]
V: VAR finite_set[T]

del_verts(G,V): graph[T] =
(# vert := difference[T](vert(G),V),
edges := {e I edges(G)(e) AND
(FORALL v: V(v) IMPLIES NOT e(v))} #)

separates(G,V,s,t): bool = NOT V(s) AND NOT V(t) AND
NOT (EXISTS (w: prewalk): walk_from?(del_verts(G,V),w,s,t))

In other words V separates s and t when its removal disconnects s and t. To define the minimum separating set, we use an abstract minimum function defined in the abstract_min theory. The net result is that we end up with a function min_sep_set with all of the following desired properties

min_sep_set(G,s,t): finite_set[T] = min[seps(G,s,t),
(LAMBDA (v: seps(G,s,t)): card(v)),
(LAMBDA (v: seps(G,s,t)): true)]
separable?(G,s,t): bool = (s /= t AND NOT edge?(G)(s,t))

min_sep_set_edge: LEMMA NOT separable?(G,s,t) IMPLIES
    min_sep_set(G,s,t) = vert(G)

min_sep_set_card: LEMMA FORALL (s,t: (vert(G))): separates(G,V,s,t)
    IMPLIES card(min_sep_set(G,s,t)) <= card(V)

min_sep_set_seps: LEMMA separable?(G,s,t) IMPLIES
    separates(G,min_sep_set(G,s,t),s,t)

min_sep_set_vert: LEMMA separable?(G,s,t) AND min_sep_set(G,s,t)(v)
    IMPLIES vert(G)(v)

ends_not_in_min_sep_set: LEMMA separable?(G,s,t) AND min_sep_set(G, s, t)(v)
    IMPLIES v /= s AND v /= t

We then define sep_num as follows:

    sep_num(G,s,t): nat = card(min_sep_set(G,s,t))

Next, we define a predicate independent?(w1,w2: prewalk) that defines when two paths are independent:

    independent?(w1,w2: prewalk): bool =
      (FORALL (i,j: nat): i > 0 AND i < l(w1) - 1 AND
       j > 0 AND j < l(w2) - 1 IMPLIES
       seq(w1)(i) /= seq(w2)(j))

The concept of a set of independent paths is defined as follows:

    set_of_paths(G,s,t): TYPE = finite_set[Path_from(G,s,t)]

    ind_path_set?(G,s,t,(pset: set_of_paths(G,s,t))): bool =
      (FORALL (p1,p2: Path_from(G,s,t)):
       pset(p1) AND pset(p2) AND p1 /= p2
       IMPLIES independent?(p1,p2))

In other words, a set of paths is an ind_path_set? if all pairs of paths in the set are independent. We can now state Menger’s theorem in both directions:

    easy_menger: LEMMA FORALL (ips: set_of_paths(G,s,t)):
        separable?(G,s,t) AND
        ind_path_set?(G,s,t,ips) IMPLIES
        card(ips) <= sep_num(G,s,t)
hard_menger: AXIOM separable?(G,s,t) AND sep_num(G,s,t) = K AND vert(G)(s) AND vert(G)(t)
IMPLIES (EXISTS (ips: set_of_paths(G,s,t)):
    card(ips) = K AND ind_path_set?(G,s,t,ips))

The hard direction of menger has only been formally proved for the K = 2 case.

hard_menger: LEMMA separable?(G,s,t) AND sep_num(G,s,t) = 2 AND vert(G)(s) AND vert(G)(t)
IMPLIES (EXISTS (ips: set_of_paths(G,s,t)):
    card(ips) = 2 AND ind_path_set?(G,s,t,ips))

12 PVS Theories

The following is a list of the PVS theories and description:
<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract_min</td>
<td>abstract definition of min</td>
</tr>
<tr>
<td>abstract_max</td>
<td>abstract definition of max</td>
</tr>
<tr>
<td>circuit_deg</td>
<td>degree of circuits</td>
</tr>
<tr>
<td>circuits</td>
<td>theory of circuits</td>
</tr>
<tr>
<td>cycle_deg</td>
<td>degree of cycle</td>
</tr>
<tr>
<td>doubletons</td>
<td>theory of doubletons used for definition of edge</td>
</tr>
<tr>
<td>graphs</td>
<td>fundamental definition of a graph</td>
</tr>
<tr>
<td>graph_completed</td>
<td>unusual definition of connected graph</td>
</tr>
<tr>
<td>graph_conn_defs</td>
<td>defs of piece, path, and structural connectedness</td>
</tr>
<tr>
<td>graph_conn_piece</td>
<td>structural connected supset piece connected</td>
</tr>
<tr>
<td>graph_connected</td>
<td>all connected defns are equivalent</td>
</tr>
<tr>
<td>graph_path_conn</td>
<td>path connected supset structural connected</td>
</tr>
<tr>
<td>graph_piece_path</td>
<td>piece connected supset path connected</td>
</tr>
<tr>
<td>graph_deg</td>
<td>definition of degree</td>
</tr>
<tr>
<td>graph_deg_sum</td>
<td>theorem relating vertex degree and number of edges</td>
</tr>
<tr>
<td>graph_inductions</td>
<td>vertex and edge inductions for graphs</td>
</tr>
<tr>
<td>graph_ops</td>
<td>delete vertex and delete edge operations</td>
</tr>
<tr>
<td>h_menger</td>
<td>hard menger</td>
</tr>
<tr>
<td>ind_paths</td>
<td>definition of independent paths</td>
</tr>
<tr>
<td>max_subgraphs</td>
<td>maximal subgraphs with specified property</td>
</tr>
<tr>
<td>max_subtrees</td>
<td>maximal subtrees with specified property</td>
</tr>
<tr>
<td>meng_scaff</td>
<td>scaffolding for hard menger proof</td>
</tr>
<tr>
<td>meng_scaff_defns</td>
<td>scaffolding for hard menger proof</td>
</tr>
<tr>
<td>meng_scaff_prelude</td>
<td>scaffolding for hard menger proof</td>
</tr>
<tr>
<td>menger</td>
<td>menger’s theorem</td>
</tr>
<tr>
<td>min_walk_reduced</td>
<td>theorem that minimum walk is reduced</td>
</tr>
<tr>
<td>min_walks</td>
<td>minimum walk satisfying a property</td>
</tr>
<tr>
<td>path_lemns</td>
<td>some useful lemmas about paths</td>
</tr>
<tr>
<td>path_ops</td>
<td>deleting vertex and edge operations</td>
</tr>
<tr>
<td>paths</td>
<td>fundamental definition and properties about paths</td>
</tr>
<tr>
<td>ramsey_new</td>
<td>Ramsey’s theorem</td>
</tr>
<tr>
<td>reduce_walks</td>
<td>operation to reduce a walk</td>
</tr>
<tr>
<td>sep_set_lemns</td>
<td>properties of separating sets</td>
</tr>
<tr>
<td>sep_sets</td>
<td>definition of separating sets</td>
</tr>
<tr>
<td>subgraphs</td>
<td>generation of subgraphs from vertex sets</td>
</tr>
<tr>
<td>subgraphs_from_walk</td>
<td>generation of subgraphs from walks</td>
</tr>
<tr>
<td>subtrees</td>
<td>subtrees of a graph</td>
</tr>
<tr>
<td>tree_circ</td>
<td>theorem that tree has no circuits</td>
</tr>
<tr>
<td>tree_paths</td>
<td>theorem that tree has only one path between vertices</td>
</tr>
<tr>
<td>trees</td>
<td>fundamental definition of trees</td>
</tr>
<tr>
<td>walk_inductions</td>
<td>induction on length of a walk</td>
</tr>
<tr>
<td>walks</td>
<td>fundamental definition and properties of walks</td>
</tr>
</tbody>
</table>

The PVS specifications are available at:

13 Concluding Remarks

This paper gives a brief overview of the NASA Langley PVS Graph Theory Library. The library provides definitions and lemmas for graph operations such as deleting a vertex or edge, provides definitions for vertex degree, subgraphs, minimal subgraphs, walks and paths, notions of connectedness, circuit and trees. Both Ramsey's Theorem and Menger's Theorem are provided.

A APPENDIX: Other Supporting Theories

A.1 Graph Inductions

The graph theory library provides two basic means of performing induction on a graph: induction on the number of vertices and induction on the number of edges.

\[ \text{G, GG: VAR graph[T]} \]
\[ \text{P: VAR pred[graph[T]]} \]

\[ \text{graph_induction_vert : THEOREM (FORALL G:} \]
\[ \quad \text{(FORALL GG: size(GG) < size(G) IMPLIES P(GG))} \]
\[ \quad \text{IMPLIES P(G))} \]
\[ \quad \text{IMPLIES (FORALL G: P(G))} \]

\[ \text{graph_induction_edge : THEOREM (FORALL G:} \]
\[ \quad \text{(FORALL GG: num_edges(GG) < num_edges(G) IMPLIES P(GG))} \]
\[ \quad \text{IMPLIES P(G))} \]
\[ \quad \text{IMPLIES (FORALL G: P(G))} \]

These theorems can be invoked using the PVS strategy INDUCT. For example

\[ \text{(INDUCT "G" 1 "graph_induction_vert")} \]

invokes vertex induction on formula 1. They are available in theory graph_inductions. These induction theorems were proved by rewriting with the following lemmas

\[ \text{size_prep : LEMMA (FORALL G : P(G)) IFF} \]
\[ \quad \text{(FORALL n, G : size(G) = n IMPLIES P(G))} \]

\[ \text{num_edges_prep : LEMMA (FORALL G : P(G)) IFF} \]
\[ \quad \text{(FORALL n, G : num_edges(G) = n IMPLIES P(G))} \]

which converts the theorem into formulas that are universally quantified over the naturals. The resulting formulas were then easily proved using PVS's built-in theorem for strong induction:
NAT_induction: LEMMA
(FORALL j: (FORALL k: k < j IMPLIES p(k)) IMPLIES p(j))
IMPLIES (FORALL i: p(i))

A.2 Subgraphs Generated From Walks

The graph theory library provides a function $G_{from}$ that constructs a subgraph of a graph $G$ that contains the vertices and edges of a walk $w$:

$G_{from}(G_{graph}[T], w: Walk(G)): Subgraph(G) = (\# vert := verts_of(w),
edges := edges_of(w) \#)$

The following properties of $G_{from}$ have been proved:

vert$_{G_{from}}$ : LEMMA FORALL (w: Walk(G), i: below(l(w))):
vert(G$_{from}(G, w))(w(i))$

edge$_{G_{from}}$ : LEMMA FORALL (w: Walk(G), i: below(l(w)-1)):
edge?(G$_{from}(G, w))(w(i), w(i+1))$

vert$_{G_{from} \_not}$ : LEMMA FORALL (w: Walk(G)):
subset?(vert(G$_{from}(G, w)), vert(GG)) AND
NOT verts_of(w)(v)
IMPLIES
subset?(vert(G$_{from}(G, w)), remove[T](v, vert(GG)))

del$_{vert \_subgraph}$: LEMMA FORALL (w: Walk(G), v: (vert(GG))):
subgraph?(G$_{from}(G, w), GG) AND
NOT verts_of(w)(v) IMPLIES
subgraph?(G$_{from}(G, w), del_{vert}(GG, v))$

This lemmas are available in the theory subgraphs_from_walk.

A.3 Maximum Subgraphs

Given a graph $G$ we say that a subgraph $S$ is maximal with respect to a particular property $P$ if it is the largest subgraph that satisfies the property. Formally we write:

maximal?(G: graph[T], S: Subgraph(G), P: Gpred(G)): bool = P(S) AND
(FORALL (SS: Subgraph(G)): P(SS) IMPLIES
size(SS) <= size(S))
We can define a function that returns the maximum subgraph under the assumption that there exists at least one subgraph that satisfies the predicate. Therefore this function is only defined on a subtype of \( P \), namely \( G_{pred} \):

\[
G: \text{VAR} \ \text{graph}[T] \\
G_{pred}(G): \text{TYPE} = P: \text{pred}[\text{graph}[T]] \mid (\text{EXISTS} \ (S: \text{graph}[T]): \\
\text{subgraph}(S, G) \ \text{AND} \ P(S))
\]

We now define \( \text{max}_{-}\text{subgraph} \) as follows:

\[
\text{max}_{-}\text{subgraph}(G: \text{graph}[T], P: G_{pred}(G)): S: \text{Subgraph}(G) \mid \text{maximal}(G, S, P)
\]

The following useful properties of \( \text{max}_{-}\text{subgraph} \) have been proved:

\[
\text{max}_{-}\text{subgraph}_{-}\text{def} : \text{LEMMA} \ \text{FORALL} \ (P: G_{pred}(G)):
\quad \text{maximal}(G, \text{max}_{-}\text{subgraph}(G, P), P)
\]

\[
\text{max}_{-}\text{subgraph}_{-}\text{in} : \text{LEMMA} \ \text{FORALL} \ (P: G_{pred}(G)) : P(\text{max}_{-}\text{subgraph}(G, P))
\]

\[
\text{max}_{-}\text{subgraph}_{-}\text{is}_{-}\text{max} : \text{LEMMA} \ \text{FORALL} \ (P: G_{pred}(G)):
\quad \text{FORALL} \ (SS: \text{Subgraph}(G)) : P(SS) \ \text{IMPLIED}\n\quad \text{size}(SS) \leq \text{size}(\text{max}_{-}\text{subgraph}(G, P))
\]

These definitions and lemmas are located in the theory \( \text{max}_{-}\text{subgraphs} \).

A similar theory for subtrees is available in the theory \( \text{max}_{-}\text{subtrees} \).

### A.4 Minimum Walks

Given that a walk \( w \) from vertex \( x \) to vertex \( y \) exists, we sometimes need to find the shortest walk from \( x \) to \( y \). The theory \( \text{min}_{-}\text{walks} \) provides a function \( \text{min}_{-}\text{walk}_{-}\text{from} \) that returns a walk that is minimal. It is defined formally as follows:

\[
v_1, v_2, x, y: \text{VAR} \ T \\
G: \text{VAR} \ \text{graph}[T] \\
\text{gr}_{-}\text{walk}(v_1, v_2): \text{TYPE} = G: \text{graph}[T] \mid \text{vert}(G)(v_1) \ \text{AND} \ \text{vert}(G)(v_2) \ \text{AND} \\
\text{(EXISTS} \ (w: \text{Seq}(G)):\n\quad \text{walk}_{-}\text{from}(G,w,v_1,v_2))
\]

\[
\text{min}_{-}\text{walk}_{-}\text{from}(x,y,(G_w: \text{gr}_{-}\text{walk}(x,y))): \text{Walk}(G_w) = \\
\quad \text{min}[\text{Seq}(G_w), \text{LAMBDA} \ (w: \text{Seq}(G_w)) : \text{1}(w)), \\
\quad \text{(LAMBDA} \ (w: \text{Seq}(G_w)) : \text{walk}_{-}\text{from}(G_w,w,x,y))]
\]

The following properties of \( \text{min}_{-}\text{walk}_{-}\text{from} \) have been established:
is_min(G,(w: Seq(G)),x,y): bool = walk?(G,w) AND
    (FORALL (ww: Seq(G)): walk_from?(G,ww,x,y) IMPLIES
    l(w) <= l(ww))

min_walk_def: LEMMA FORALL (Gw: gr_walk(x,y)):
    walk_from?(Gw,min_walk_from(x,y,Gw),x,y) AND
    is_min(Gw, min_walk_from(x,y,Gw),x,y)

min_walk_in : LEMMA FORALL (Gw: gr_walk(x,y)):
    walk_from?(Gw,min_walk_from(x,y,Gw),x,y)

min_walk_is_min: LEMMA FORALL (Gw: gr_walk(x,y), ww: Seq(Gw)):
    walk_from?(Gw,ww,x,y) IMPLIES
    l(min_walk_from(x,y,Gw)) <= l(ww)

reduced?(G: graph[T], w: Seq(G)): bool =
    (FORALL (k: nat): k > 0 AND k < l(w) - 1 IMPLIES w(k-1) /= w(k+1))

x,y: VAR T

min_walk_is_reduced: LEMMA FORALL (Gw: gr_walk(x,y)):
    reduced?(Gw,min_walk_from(x,y,Gw))

These lemmas are available in the theories min_walks and min_walk_reduced.

### A.5 Abstract Min and Max Theories

The need for a function that returns the smallest or largest object that satisfies a particular predicate arises in many contexts. For example, one may need a minimal walk from s to t or the maximal subgraph that contains a tree. Thus, it is useful to develop abstract min and max theories that can be instantiated in multiple ways to provide different min and max functions. Such a theory must be parameterized by

- **T**: TYPE
- **size**: [T -> nat] (the type of the object for which a min function is needed)
- **P**: pred[T] (the "size" function by which objects are compared)
- **P**: pred[T] (the property that the min function must satisfy)

Formally we have

abstract_min[T: TYPE, size: [T -> nat], P: pred[T]]: THEORY

and

abstract_max[T: TYPE, size: [T -> nat], P: pred[T]]: THEORY
To simplify the following discussion, only the abstract_min theory will be elaborated in detail. The abstract_max theory is conceptually identical.

In order for a minimum function to be defined, it is necessary that at least one object exists that satisfies the property. Thus, the theory contains the following assuming clause

**ASSUMING**

\[
T_{\text{ne}}: \text{ASSUMPTION EXISTS } (t: T): P(t)
\]

**ENDASSUMING**

Users of this theory are required to prove that this assumption holds for their type T (via PVS’s TCC generation mechanism).

A function \( \text{minimal?}(S: T) \) is then defined as follows:

\[
\text{minimal?}(S): \text{bool} = P(S) \text{ AND } (\forall SS: T: P(SS) \text{ IMPLIES size}(S) \leq \text{size}(SS))
\]

Using PVS’s dependent type mechanism, \( \text{min} \) is specified by constraining its return type to be the subset of \( T \) that satisfies \( \text{minimal?} \):

\[
\text{min}: \{S: T | \text{minimal?}(S)\}
\]

If there are multiple instances of objects that are minimal, the theory does not specify which object is selected by \( \text{min} \). It just states that \( \text{min} \) will return one of the minimal ones. This definition causes PVS to generate the following proof obligation (i.e. TCC):

\[
\text{min}\_\text{TCC1}: \text{OBLIGATION } (\exists x: S: T | \text{minimal?}(S)): \text{TRUE});
\]

This was proved using a function \( \text{min}\_f \), defined as follows:

\[
\text{is_one}(n): \text{bool} = (\exists S: T: P(S) \text{ AND size}(S) = n)
\]

\[
\text{min}\_f: \text{nat} = \text{min}[\text{nat}](n: \text{nat} | \text{is_one}(n))
\]

to construct the required \( \text{min} \) function. The \( T_{\text{ne}} \) assumption is sufficient to guarantee that \( \text{min}\_f \) is well-defined.

The following properties have been proved about \( \text{min} \):

\[
\text{min}\_\text{def}: \text{LEMMA } \text{minimal?}(\text{min})
\]

\[
\text{min}\_\text{in} : \text{LEMMA } P(\text{min})
\]

\[
\text{min}\_\text{is_min}: \text{LEMMA } P(SS) \text{ IMPLIES size}(\text{min}) \leq \text{size}(SS)
\]

These properties are sufficient for most applications.
**Title and Subtitle:**
A PVS Graph Theory Library

**Authors:**
Ricky W. Butler and Jon A. Sjogren

**Performing Organization:**
NASA Langley Research Center
Hampton, VA 23681-2199

**Sponsoring Agency:**
National Aeronautics and Space Administration
Washington, DC 20546-0001

**Abstract:**
This paper documents the NASA Langley PVS graph theory library. The library provides fundamental definitions for graphs, subgraphs, walks, paths, subgraphs generated by walks, trees, cycles, degree, separating sets, and four notions of connectedness. Theorems provided include Ramsey's and Menger's and the equivalence of all four notions of connectedness.

**Subject Terms:**
Graphs, Formal Methods, PVS Libraries, Formal Proof

**Security Classification:**
Unclassified

**Distribution Code:**
Unclassified-Unlimited, 59, Nonstandard, NASA CASI (301) 621-0390

**Number of Pages:**
29

**Price Code:**
A03