NASA/TM-1998-206923

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

The NASA STI Program Office ... in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA Scientific and Technical Information (STI) Program Office plays a key part in helping NASA maintain this important role.

The NASA STI Program Office is operated by Langley Research Center, the lead center for NASA's scientific and technical information. The NASA STI Program Office provides access to the NASA STI Database, the largest collection of aeronautical and space science STI in the world. The Program Office is also NASA's institutional mechanism for disseminating the results of its research and development activities. These results are published by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counter-part of peer reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services that help round out the STI Program Office's diverse offerings include creating custom thesauri, building customized databases, organizing and publishing research results ... even providing videos.

For more information about the NASA STI Program Office, see the following:

- Access the NASA STI Program Home Page at ***http://www.sti.nasa.gov***

- E-mail your question via the Internet to help@sti.nasa.gov

- Fax your question to the NASA Access Help Desk at (301) 621-0134

- Phone the NASA Access Help Desk at (301) 621-0390

- Write to:
  NASA Access Help Desk
  NASA Center for AeroSpace Information
  800 Elkridge Landing Road
  Linthicum Heights, MD 21090-2934

NASA/TM-1998-206923

A PVS Graph Theory Library

Ricky W. Butler
Langley Research Center, Hampton, Virginia

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

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

February 1998

Available from the following:

NASA Center for AeroSpace Information (CASI)
800 Elkridge Landing Road
Linthicum Heights, MD 21090-2934
(301) 621-0390

National Technical Information Service (NTIS)
5285 Port Royal Road
Springfield, VA 22161-2171
(703) 487-4650

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

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 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 `proj_1` and `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`:

```
pregraph: TYPE = [# vert : finite_set[T],
                    edges: finite_set[doubleton[T]] #]
```

A `pregraph` is a structured type with two components: `vert` and `edges`. The `vert` component is a finite set over an arbitrary type `T`. This represents the vertices of the graph. The `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 `finite_set` is defined in the PVS finite sets library. It is a subtype of the type `set` which is defined in the PVS prelude as follows:

```
sets [T: TYPE]: THEORY
BEGIN
  set: TYPE = [T -> bool]

  x, y: VAR T
  a, b, c: VAR set
  p: VAR [T -> bool]

  member(x, a): bool = a(x)
  emptyset: set = x | false
  subset?(a, b): bool = (FORALL x: member(x, a) => member(x, b))
  union(a, b): set = x | member(x, a) OR member(x, b)
  intersection(a, b): set = x | member(x, a) AND member(x, b)

END sets
```

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

```
    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 [1].

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:

[1] 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.

vert(G)	vertices of graph G (a finite set of T)
edges(G)	edges of a graph G (a finite set of doubletons taken from vert(G))
edge?(G)(x,y)	true IFF there is an edge between vertices x and y
empty?(G)	true IFF the graph G has no vertices
singleton?(G)	true IFF graph G has only 1 vertex
isolated?(G)	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

edge?_comm           : LEMMA edge?(G)(y, x) IMPLIES edge?(G)(x, y)

edge_has_2_verts     : LEMMA x /= v AND e(x) AND e(v) IMPLIES e = dbl(x,v)

edge_in_card_gt_1    : LEMMA edges(G)(e) IMPLIES card(vert(G)) > 1

not_singleton_2_vert : LEMMA NOT empty?(G) AND NOT singleton?(G)
                                IMPLIES (EXISTS (x,y: T): x /= y AND
                                     vert(G)(x) AND vert(G)(y))
```

These definitions and lemmas are located in the graphs theory.

3 Graph Operations

The theory graph_ops defines the following operations on a graph:

union(G1,G2)	creates a graph that is a union of G1 and G2
del_vert(G,v)	removes vertex v and all adjacent edges to v from the graph G
del_edge(e,G)	creates subgraph with edge e removed from G

These operations are defined as follows:

```
union(G1,G2): graph[T] = (# vert := union(vert(G1),vert(G2)),
                            edges := union(edges(G1),edges(G2)) #)

del_vert(G: graph[T], v: T): graph[T] =
        (# vert := remove[T](v,vert(G)),
           edges := e | edges(G)(e) AND NOT e(v) #)

del_edge(G,e): graph[T] = G WITH [edges := remove(e,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 deg(v,G) > 0 IMPLIES size(G) >= 2

del_vert_deg_0  : LEMMA deg(v,G) = 0 IMPLIES edges(del_vert(G,v)) = edges(G)

deg_del_vert    : LEMMA x /= v AND edges(G)(dbl[T](x, v))
                             IMPLIES deg(v, del_vert(G, x)) =
                                        deg(v, G) - 1

del_vert_not_incident: LEMMA x /= v AND NOT edges(G)(dbl[T](x, v)) IMPLIES
                                    deg(x, del_vert(G, v)) = deg(x, G)

singleton_deg: LEMMA singleton?(G) IMPLIES deg(v, G) = 0
```

5 Subgraphs

The subgraph relation is defined as a predicate named `subgraph?`:

```
G1,G2: VAR graph[T]

subgraph?(G1,G2): bool = subset?(vert(G1),vert(G2)) AND
                         subset?(edges(G1),edges(G2))
```

The subgraph type is defined using this predicate:

```
Subgraph(G: graph[T]): TYPE = { S: graph[T] | subgraph?(S,G) }
```

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

```
i: VAR T
e: VAR doubleton[T]
subgraph(G, V): Subgraph(G) =
    (G WITH [vert :=  {i | vert(G)(i) AND V(i) },
             edges := {e | edges(G)(e) AND
                           (FORALL (x: T): e(x) IMPLIES V(x)) }])
```

The following properties have been proved:

```
finite_vert_subset : LEMMA is_finite(LAMBDA (i:T): vert(G)(i) AND V(i))


subgraph_vert_sub : LEMMA subset?(V,vert(G)) IMPLIES
                                   vert(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 `i`th element in the sequence. Prewalks are contrained 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 Walk(G) defines the domain (or type) of all walks in a graph G. The dependent type Seq(G) defines the domain (or type) of all prewalks in a particular graph G.

The predicates from? and walk_from? identify sequences and walks from one particular vertex to another.

```
from?(ps,u,v): bool = seq(ps)(0) = u AND seq(ps)(l(ps) - 1) = v

walk_from?(G,ps,u,v): bool =
               seq(ps)(0) = u AND seq(ps)(l(ps) - 1) = v AND walk?(G,ps)
```

The function verts_of returns the set of vertices that are in a walk:

```
verts_of(ww: prewalk): finite_set[T] =
            {t: T | (EXISTS (i: below(l(ww))): ww(i) = t)}
```

Similarly, the function edges_of returns the set of edges that are in a walk:

```
edges_of(ww): finite_set[doubleton[T]] = {e: doubleton[T] |
                       EXISTS (i: below(l(ww)-1)): e = dbl(ww(i),ww(i+1))}
```

Below are listed some of the proved properties about walks:

```
G,GG: VAR graph[T]
x,u,v: VAR T
i,j,n: VAR nat
ps: VAR prewalk


verts_in?_concat: LEMMA FORALL (s1,s2: Seq(G)): verts_in?(G,s1 o s2)

verts_in?_caret : LEMMA FORALL (j: below(l(ps))): i <= j IMPLIES
                                verts_in?(G,ps) IMPLIES verts_in?(G,ps^(i,j))

vert_seq_lem    : LEMMA FORALL (w: Seq(G)): n < l(w) IMPLIES vert(G)(w(n))

verts_of_subset : LEMMA FORALL (w: Seq(G)): subset?(verts_of(w),vert(G))


edges_of_subset : LEMMA walk?(G,ps) IMPLIES subset?(edges_of(ps),edges(G))

walk_verts_in   : LEMMA walk?(G,ps) IMPLIES verts_in?(G,ps)


walk_from_vert  : 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`
`trunc1(p)`	return a prewalk equal to `p` except the last vertex has been removed
`add1(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`	concatenates two finite sequences
`^(m,n)`	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:

```
gen_seq1(G, (u: (vert(G)))): Seq(G) =
                     (# l := 1, seq := (LAMBDA (i: below(1)): u) #)

gen_seq2(G, (u,v: (vert(G)))): Seq(G) =
               (# l := 2,
                  seq := (LAMBDA (i: below(2)):
                                    IF i = 0 THEN u ELSE v ENDIF) #)

Longprewalk: TYPE = {ps: prewalk | l(ps) >= 2}

trunc1(p: Longprewalk ): prewalk = p^(0,l(p)-2)

add1(ww,x): prewalk = (# l := l(ww) + 1,
                         seq := (LAMBDA (ii: below(l(ww) + 1)):
                                  IF ii < l(ww) THEN seq(ww)(ii) ELSE x ENDIF)
                       #)

fs, fs1, fs2, fs3: VAR finite_seq
m, n: VAR nat

o(fs1, fs2): finite_seq =
   LET l1 = l(fs1),
       lsum = l1 + l(fs2)
    IN (# l := lsum,
          seq := (LAMBDA (n:below[lsum]):
```

```
                  IF n < l1
                     THEN seq(fs1)(n)
                     ELSE seq(fs2)(n-l1)
                  ENDIF) #);

emptyarr(x: below[0]): T
emptyseq: fin_seq(0) = (# l := 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 (# l := n-m+1,
              seq := (LAMBDA (x: below[n-m+1]): seq(fs)(x + m)) #)
      ENDIF ;

rev(fs): finite_seq = (# l := 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?_add1       : 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,add1(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 are equivalent. These are named `connected`, `path_connected`, `piece_connected`, and `complected`:

```
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))))

complected?(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
```

```
BEGIN

   G: VAR graph[T]

   conn_eq_path : THEOREM connected?(G) = path_connected?(G)

   path_eq_piece: THEOREM path_connected?(G) = piece_connected?(G)

   piece_eq_conn: THEOREM piece_connected?(G) = connected?(G)

   conn_eq_complected: THEOREM connected?(G) = complected?(G)

END graph_connected
```

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 <
       l(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)-1)

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: VAR graph[T]

tree?(G): RECURSIVE bool = card[T](vert(G)) = 1 OR
                      (EXISTS (v: (vert(G))): deg(v,G) = 1 AND
                               tree?(del_vert[T](G,v)))
          MEASURE size(G)
```

and the Tree type is defined as follows:

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

The fundamental property that trees have no circuits is proved in `tree_circ` theory.

```
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: VAR T

n, p, q, ii: VAR nat
g: VAR graph[T]
G: VAR Graph[T]   % nonempty

V: VAR finite_set[T]

contains_clique(g, n): bool =
    (EXISTS (C: finite_set[T]):
      subset?(C,vert(g)) AND card(C) >= n AND
        (FORALL i,j: i/=j AND C(i) AND C(j) IMPLIES edge?(g)(i,j)))

contains_indep(g, n): bool =
    (EXISTS (D: finite_set[T]):
```

[2]CLI Technical Report 58, July 17, 1990.

```
      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, l1) OR
                                      contains_indep(G, l2))))
```

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 | 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?` 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:

`abstract_min`	abstract definition of min
`abstract_max`	abstract definition of max
`circuit_deg`	degree of circuits
`circuits`	theory of circuits
`cycle_deg`	degree of cycle
`doubletons`	theory of doubletons used for definition of edge
`graphs`	fundamental definitiion of a graph
`graph_complected`	unusual definition of connected graph
`graph_conn_defs`	defs of piece, path, and structural connectedness
`graph_conn_piece`	structural connected *supset* piece connected
`graph_connected`	all connected defs are equivalent
`graph_path_conn`	path connected *supset* structural connected
`graph_piece_path`	piece connected *supset* path connected
`graph_deg`	definition of degree
`graph_deg_sum`	theorem relating vertex degree and number of edges
`graph_inductions`	vertex and edge inductions for graphs
`graph_ops`	delete vertex and delete edge operations
`h_menger`	hard menger
`ind_paths`	definition of independent paths
`max_subgraphs`	maximal subgraphs with specified property
`max_subtrees`	maximal subtrees with specified property
`meng_scaff`	scaffolding for hard menger proof
`meng_scaff_defs`	scaffolding for hard menger proof
`meng_scaff_prelude`	scaffolding for hard menger proof
`menger`	menger's theorem
`min_walk_reduced`	theorem that minimum walk is reduced
`min_walks`	minimum walk satisfying a property
`path_lems`	some useful lemmas about paths
`path_ops`	deleting vertex and edge operations
`paths`	fundamental definition and properties about paths
`ramsey_new`	Ramsey's theorem
`reduce_walks`	operation to reduce a walk
`sep_set_lems`	properties of separating sets
`sep_sets`	definition of separating sets
`subgraphs`	generation of subgraphs from vertex sets
`subgraphs_from_walk`	generation of subgraphs from walks
`subtrees`	subtrees of a graph
`tree_circ`	theorem that tree has no circuits
`tree_paths`	theorem that tree has only one path between vertices
`trees`	fundamental definition of trees
`walk_inductions`	induction on length of a walk
`walks`	fundamental definition and properties of walks

The PVS specifications are available at:

`http://atb-www.larc.nasa.gov/ftp/larc/PVS-library/.`

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.

```
G,GG: VAR graph[T]
P: VAR pred[graph[T]]

graph_induction_vert    : THEOREM (FORALL G:
                                (FORALL GG: size(GG) < size(G) IMPLIES P(GG))
                                                IMPLIES P(G))
                                   IMPLIES (FORALL G: P(G))


graph_induction_edge    : THEOREM (FORALL G:
                               (FORALL GG: num_edges(GG) < num_edges(G) IMPLIES P(GG))
                                                IMPLIES P(G))
                                    IMPLIES (FORALL G: P(G))
```

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

```
(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

```
size_prep              : LEMMA (FORALL G : P(G)) IFF
                                     (FORALL n, G : size(G) = n IMPLIES P(G))

num_edges_prep         : LEMMA (FORALL G : P(G)) IFF
                                  (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(Ggraph[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 Gpred:

```
G: VAR graph[T]

Gpred(G): TYPE = P: pred[graph[T]] | (EXISTS (S: graph[T]):
                                         subgraph?(S,G) AND P(S))
```

We now define max_subgraph as follows:

```
max_subgraph(G: graph[T], P: Gpred(G)): S: Subgraph(G) | maximal?(G,S,P)
```

The following useful properties of max_subgraph have been proved:

```
max_subgraph_def    : LEMMA FORALL (P: Gpred(G)):
                                 maximal?(G,max_subgraph(G,P),P)

max_subgraph_in     : LEMMA FORALL (P: Gpred(G)): P(max_subgraph(G,P))

max_subgraph_is_max : LEMMA FORALL (P: Gpred(G)):
                       (FORALL (SS: Subgraph(G)): P(SS) IMPLIES
                                      size(SS) <= size(max_subgraph(G,P)))
```

These definitions and lemmas are located in the theory max_subgraphs.

A similar theory for subtrees is available in the theory max_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 min_walks provides a function min_walk_from that returns a walk that is minimal. It is defined formally as follows:

```
v1,v2,x,y: VAR T
G: VAR graph[T]

gr_walk(v1,v2): TYPE = G: graph[T] | vert(G)(v1) AND vert(G)(v2) AND
                                       (EXISTS (w: Seq(G)):
                                         walk_from?(G,w,v1,v2))

min_walk_from(x,y,(Gw:gr_walk(x,y))): Walk(Gw) =
                     min[Seq(Gw),(LAMBDA (w: Seq(Gw)): l(w)),
                            (LAMBDA (w: Seq(Gw)): walk_from?(Gw,w,x,y))]
```

The following properties of min_walk_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`	the type of the object for which a `min` function is needed
`size:[T -> nat]`	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_ne: ASSUMPTION EXISTS (t: T): P(t)

ENDASSUMING
```

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

A function `minimal?(S: T)` is then defined as follows:

```
minimal?(S): bool = P(S) AND
                    (FORALL (SS: T): P(SS) IMPLIES size(S) <= size(SS))
```

Using PVS's dependent type mechanism, `min` is specified by constraining it's return type to be the subset of `T` that satisfies `minimal?`:

```
min: {S: T | minimal?(S)}
```

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

```
min_TCC1: OBLIGATION (EXISTS (x: S: T | minimal?(S)): TRUE);
```

This was proved using a function `min_f`, defined as follows:

```
is_one(n): bool = (EXISTS (S: T): P(S) AND size(S) = n)

min_f: nat = min[nat](n: nat | is_one(n))
```

to construct the required `min` function. The `T_ne` assumption is sufficient to guarantee that `min_f` is well-defined.

The following properties have been proved about `min`:

```
min_def: LEMMA minimal?(min)

min_in : LEMMA P(min)

min_is_min: LEMMA P(SS) IMPLIES size(min) <= size(SS)
```

These properties are sufficient for most applications.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	February 1998	Technical Memorandum

4. TITLE AND SUBTITLE
A PVS Graph Theory Library

5. FUNDING NUMBERS
WU 519-50-11-01

6. AUTHOR(S)
Ricky W. Butler and Jon A. Sjogren

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681-2199

8. PERFORMING ORGANIZATION REPORT NUMBER
L-17692

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
NASA/TM-1998-206923

11. SUPPLEMENTARY NOTES
Butler, Ricky W.: NASA Langley Research Ctr.
Sjogren, Jon A.: Air Force Office of Scientific Research; Washington, DC

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Unclassified-Unlimited
Subject Category: 59
Distribution: Nonstandard
Availability: NASA CASI (301) 621-0390

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)
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.

14. SUBJECT TERMS
Graphs, Formal Methods, PVS Libraries, Formal Proof

15. NUMBER OF PAGES
29

16. PRICE CODE
A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		