Computing with Directed Graphs in Magma

Don Taylor

9 July 2002

Last changed: 1 August 2002

This document is an exercise in "literate MAGMA programming". That is, the source file combines documentation and executable code. The file can be processed with LaTeX to produce printable output and the code can be extracted and loaded into MAGMA.

The primary aim is to develop a MAGMA implementation of Tarjan's algorithm for computing the strong components of directed graphs. This algorithm is described in the book "The Design and Analysis of Computer Algorithms" by A. V. Aho, J. E. Hopcroft and J. D. Ullman (Addison-Wesley 1974). An account of a similar algorithm together with a proof of correctness was given by Edsger Dijkstra in Chapter 25 of his book "A Discipline of Programming" (Prentice-Hall, 1976).

1 Directed graphs

A *directed graph* is a pair $D = (N, A)$ where N is a set and A is a subset of $N \times N$. The elements of N are the *nodes* of D and the elements of A are its *arcs*.

Even though MAGMA has internal support for directed graphs as a data type, we shall take a fairly minimalist approach and (for the time being) assume that N is an initial segment of the positive integers and that the arcs are specified by a list *arcs*, where *arcs*$[i]$ is the set $\{ j \mid (i, j) \in A \}$.

Nodes u and v are said to be *strongly connected* if they belong to a directed circuit; that is, if there is a directed path from u to v and a directed path from v to u. Being strongly connected is an equivalence relation and the equivalence classes are called the *strong components* of the directed graph.

Tarjan's algorithm is a clever variant of the depth-first search algorithm and so we shall describe that first.

2 Depth-first search

Depth-first search is the name given to an algorithm that computes a directed spanning forest of a directed graph D and ranks the nodes in the order visited. Informally, the algorithm is a process that "visits" each node exactly once and optionally carries out some computation there during the visit. The forest is a disjoint union of directed trees with the property that at each node except one (the *root*) there is exactly one incoming arc.

If we are visiting node v and if there is an arc from v to an unvisited node w, then we visit w and continue as far as possible from there before exploring any other arcs leaving v.

In the following MAGMA code, *n* is the number of nodes and *arcs* is the collection of arcs, as described in the previous section.

```
1    depthFirstSearch := function( arcs )
2        context := recformat< forest, ρ, counter >;
```

The current state is a record *S* of type *context* and a reference to *S* is passed to each (recursive) call of the local procedure nodeVisitor. In this procedure, *v* is the current node. The *counter* field of *S* keeps track of the number of nodes processed so far and its value becomes the rank $\rho(v)$ of the current node. A node of rank 0 has not been visited.

```
3            nodeVisitor := procedure( v, ~S )
4                S`counter +:= 1;
5                S`ρ[v] := S`counter;
6                for w in arcs[v] do
7                    if S`ρ[w] eq 0 then
8                        INCLUDE( ~S`forest[v], w );
9                        $$( w, ~S );
10                   end if;
11               end for;
12           end procedure;
13           n := #arcs;
14           candidate := 1;
15           state := rec< context |
16               forest := [{INTEGERS()|} : i in [1..n]],
17               ρ := [0 : i in [1..n]],
18               counter := 0 >;
```

The call to nodeVisitor within the **repeat** loop occurs once for each tree in the forest. *candidate* is the next available node that has not been visited.

```
19           repeat
20               nodeVisitor( candidate, ~state );
21               while candidate le n and state`ρ[candidate] gt 0 do
22                   candidate +:= 1;
23               end while;
24           until candidate gt n;
25           return state`forest, state`ρ;
26       end function;
```

3 Strong components

After carrying out a depth-first search, each arc (v, w) of the directed graph has one of five types and the ranks of v and w are related as follows:

loop	$v = w$	$\rho(v) = \rho(w)$
tree	(v, w) is in the forest	$\rho(v) < \rho(w)$
forward	v is an ancestor of w	$\rho(v) < \rho(w)$
back	v is a descendant of w	$\rho(v) > \rho(w)$
cross	v and w are unrelated	$\rho(v) > \rho(w)$

Lemma 1. *The ranks of the descendants of node v are consecutive positive integers, beginning with the rank of v.*

Proof. The depth-first search process visits all descendants of v before back-tracking to visit other nodes. □

Lemma 2. *If K is a strong component of D and if r is the node of least rank in K, then every node of K is a descendant of r.* We shall call r the *root* of the strong component.

Proof. Let x be a node in K and consider a directed path P in K from r to x. By Lemma 1 and the properties of the rank function given above, the path P cannot leave the set of descendants of r and therefore x is a descendant of r. □

Lemma 3. *The ancestors of the node v are the nodes whose visit has not been completed when v is reached in the depth-first search process.*

Proof. Each node v, other than a root, has a unique immediate ancestor and the visit of this node cannot be completed until the process back-tracks from v. □

While constructing the strong components we effectively maintain a partition of the nodes into three sets S, T and U. Initially the sets S and T are empty and U contains all the nodes. When the algorithm terminates, S will contain all the nodes (partitioned into strong components) and T and U will be empty. In an intermediate state, S is a union of strong components, T is a partially processed tree and U is the set of unprocessed nodes. There will be no arcs from any node of S to a node not in S. The tree T will be constructed by carrying out a depth-first search and S will be obtained from T by pruning a strong component from T and moving it to S as soon as it has been found.

In the code that follows, the sets S, T and U can be obtained from the record representing the current state: S is the union of the elements of the ***components*** field, T is the ***stack*** field (equivalently, the nodes for which ***stacked*** is true), and the nodes in U are those for which the rank is 0.

For each node v, let $\Sigma(v)$ be the set of nodes w such that there is directed path from v to the root of the strong component of v and w is either v or the first node of the path that is not a descendant of v. Let $\sigma(v) = \min\{\, \rho(w) \mid w \in \Sigma \,\}$.

Lemma 4. *We have $\sigma(v) \leq \rho(v)$ and equality holds if and only if v is the root of its strong component.*

Proof. Let r be the root of the strong component containing v. If $w \in \Sigma(v)$, then by Lemma 1 we have $\rho(w) < \rho(v)$ unless $w = v$. Thus $\sigma(v) \leq \rho(v)$ and if $v \neq r$ we have $\sigma(v) < \rho(v)$. Conversely, if $\sigma(v) < \rho(v)$, there is a path from v to r through w such that $\sigma(v) = \rho(w)$ and therefore $\rho(r) \leq \rho(w) < \rho(v)$. In particular, $v \neq r$. □

It follows from this lemma that $\sigma(v)$ is the minimum of $\rho(v)$ and the values $\sigma(w)$, where $w \in T$ and w is either a direct descendant of v or the target of a back arc or a cross arc from v.

```
1    strongComponents := function( arcs )
```

The global context will be a record with the following fields.

```
2        context := recformat< components, ρ, σ, stack, stacked, counter >;
```

components is the list of strong components obtained so far;
counter is the number of nodes processed so far;
ρ is the list of "depth-first" ranks of the nodes;
σ is the "sub-rank" function defined above;
stacked is true if the node is on the stack.

```
3       nodeVisitor := procedure( v, ~S )
4           S`counter +:= 1;
5           S`ρ[v] := S`counter;
6           S`σ[v] := S`counter;
7           APPEND( ~S`stack, v );
8           S`stacked[v] := true;
9           for w in arcs[v] do
10              if S`ρ[w] eq 0 then
11                  $$( w, ~S );
12                  S`σ[v] := MIN( S`σ[v], S`σ[w] );
13              elif S`ρ[w] lt S`ρ[v] and S`stacked[w] then
14                  S`σ[v] := MIN( S`ρ[w], S`σ[v] );
15              end if;
16          end for;
```

At this point we have searched all the immediate descendants of *v* and so we check to see if *v* is the root of its strong component. If it is, we remove the entire strong component from the stack and append it to *components*.

```
17          if S`σ[v] eq S`ρ[v] then
18              comp := {};
19              repeat
20                  x := S`stack[#S`stack];
21                  PRUNE( ~S`stack );
22                  INCLUDE( ~comp, x );
23                  S`stacked[x] := false;
24              until x eq v;
25              APPEND( ~S`components, comp );
26          end if;
27      end procedure;
28      n := #arcs;
29      candidate := 1;
30      state := rec< context |
31          components := [],
32          ρ := [0 : i in [1..n]],
33          σ := [0 : i in [1..n]],
34          stacked := [false : i in [1..n]],
35          stack := [],
36          counter := 0 >;
```

The main loop runs through each tree in the forest.

```
37          repeat
38              nodeVisitor( candidate, ~state );
39              while candidate le n and state`ρ[candidate] gt 0 do
40                  candidate +:= 1;
41              end while;
42          until candidate gt n;
43          return state`components;
44      end function;
```

4 Examples

The first example is from page 193 of Aho, Hopcroft and Ullman.

```
E := [[2,4,5],
    [3,4],
    [1],
    [3],
    [4],
    [7,8],
    [5],
    [4,6,7]];
```

Here is an example with 20 nodes and 9 strong components. It was generated by having MAGMA choose random subsets of size at most 5 from the set $\{1..20\}$.

```
D := [{},
    {},
    { 2, 11, 12, 16, 19 },
    { 2, 10, 13, 17, 18 },
    { 13, 18 },
    { 6, 16, 17 },
    { 1, 5, 8, 16, 18 },
    { 1, 2 },
    { 13 },
    {},
    { 2, 7, 9, 19 },
    { 13 },
    { 1, 2, 20 },
    {},
    { 8, 16, 20 },
    { 12 },
    { 2, 8, 9, 14, 18 },
    { 4, 5, 9, 11 },
    { 13, 19 },
    { 5 }
];
```

5 The acyclic quotient of a directed graph

This is a straightforward extension of the strong components algorithm. It produces a new directed graph whose nodes are the strong components of the original graph. There is a single arc from component S_1 to component S_2 in the new graph whenever there is a node $v_1 \in S_1$ and a node $v_2 \in S_2$ such that (v_1, v_2) is an arc.

It is clear that the new graph cannot have non-trivial directed circuits.

```
1     acyclicQuotient := function( arcs )
2         context := recformat< ρ, σ, stack, stacked, counter,
3             compndx, comparcs, compno, newarcs >;
```

The field *compndx* is the sequence of the component numbers of the original nodes.

```
4         nodeVisitor := procedure( v, ~S )
5             S`counter +:= 1;
6             S`ρ[v] := S`counter;
7             S`σ[v] := S`counter;
8             APPEND( ~S`stack, v );
9             S`stacked[v] := true;
10            for w in arcs[v] do
11                if S`ρ[w] eq 0 then
12                    $$( w, ~S );
13                    S`σ[v] := MIN( S`σ[v], S`σ[w] );
14                elif S`stacked[w] then
15                    if S`ρ[w] lt S`ρ[v] then
16                        S`σ[v] := MIN( S`ρ[w], S`σ[v] );
17                    end if;
18                else
```

When we find an arc pointing back from *v* to a node *w* in a strong component already found, we add the component number of *w* to the set of new arcs arising from *v*. Later these will be consolidated into the arcs of the graph on the strong components themselves.

```
19                    INCLUDE( ~S`newarcs[v], S`compndx[w] );
20                end if;
21            end for;
```

If we find a new strong component, we remove it from the stack.

```
22            if S`σ[v] eq S`ρ[v] then
23                ndx := S`compno + 1;
24                arcset := {INTEGERS()|};
25                repeat
26                    x := S`stack[#S`stack];
27                    PRUNE( ~S`stack );
28                    S`compndx[x] := ndx;
29                    arcset join:= S`newarcs[x];
30                    S`stacked[x] := false;
31                until x eq v;
32                S`compno := ndx;
33                APPEND( ~S`comparcs, arcset );
34            end if;
```

```
35          end procedure;
36          n := #arcs;
37          candidate := 1;
38          state := rec< context |
39              compndx := [0 : i in [1..n]],
40              comparcs := [],
41              newarcs := [{INTEGERS()|} : i in [1..n]],
42              compno := 0,
43              ρ := [0 : i in [1..n]],
44              σ := [0 : i in [1..n]],
45              stacked := [false : i in [1..n]],
46              stack := [],
47              counter := 0 >;
48          repeat
49              nodeVisitor( candidate, ~state );
50              while candidate le n and state`ρ[candidate] gt 0 do
51                  candidate +:= 1;
52              end while;
53          until candidate gt n;
54          return state`comparcs, state`compndx;
55      end function;
```

6 The transitive closure of the acyclic quotient

Given a directed graph D we first form the acyclic quotient Q and we add additional arcs so that if there are arcs (u, v) and (v, w) then there is an arc (u, w).

```
1       transitiveQuotient := function( arcs )
2           context := recformat< ρ, σ, stack, stacked, counter,
3               compndx, transarcs, compno, newarcs >;
```

When the algorithm terminates, the field *transarcs* will be a list of the arcs of the transitive closure of the acyclic quotient.

The nodeVisitor procedure is the same as the one used in acyclicQuotient.

```
4           nodeVisitor := procedure( v, ~S )
5               S`counter +:= 1;
6               S`ρ[v] := S`counter;
7               S`σ[v] := S`counter;
8               APPEND( ~S`stack, v );
9               S`stacked[v] := true;
10              for w in arcs[v] do
11                  if S`ρ[w] eq 0 then
12                      $$( w, ~S );
13                      S`σ[v] := MIN( S`σ[v], S`σ[w] );
14                  elif S`stacked[w] then
15                      if S`ρ[w] lt S`ρ[v] then
16                          S`σ[v] := MIN( S`ρ[w], S`σ[v] );
17                      end if;
```

```
18                  else
19                      INCLUDE( ~S`newarcs[v], S`compndx[w] );
20                  end if;
21              end for;
```

If we find a new strong component, we remove it from the stack and consolidate the arcs.

```
22              if S`σ[v] eq S`ρ[v] then
23                  ndx := S`compno + 1;
24                  arcset := {INTEGERS()|};
```

At this stage *newarcs* holds the arcs from an original node to a known strong component and *transarcs* holds all the arcs in the transitive closure of those components. First we take the union of the sets in *newarcs* to get the immediate descendants of *v*. We then take the union of their descendants to get the transitive closure.

```
25                  repeat
26                      x := S`stack[#S`stack];
27                      PRUNE( ~S`stack );
28                      S`compndx[x] := ndx;
29                      arcset join:= S`newarcs[x];
30                      S`stacked[x] := false;
31                  until x eq v;
32                  S`compno := ndx;
33                  APPEND( ~S`transarcs,
34                      &join{ S`transarcs[w] : w in arcset } join arcset );
35              end if;
36          end procedure;
```

This completes the nodeVisitor procedure

```
37          n := #arcs;
38          candidate := 1;
39          state := rec< context |
40              compndx := [0 : i in [1..n]],
41              transarcs := [],
42              newarcs := [{INTEGERS()|} : i in [1..n]],
43              compno := 0,
44              ρ := [0 : i in [1..n]],
45              σ := [0 : i in [1..n]],
46              stacked := [false : i in [1..n]],
47              stack := [],
48              counter := 0 >;
49          repeat
50              nodeVisitor( candidate, ~state );
51              while candidate le n and state`ρ[candidate] gt 0 do
52                  candidate +:= 1;
53              end while;
54          until candidate gt n;
55          return state`transarcs, state`compndx;
56      end function;
```

7 Addendum

We could probably get a speed improvement in all the algorithms by representing the stack by a sequence together with a pointer to the top element.