eAppendix

```
########################################
# dagR v1.0.1                          #
# Copyright Lutz Philipp Breitling 2010 #
# lutz.breitling@gmail.com             #
########################################
# Suite of DAG functions for R;            #
# To be put into an R package in due time. #
##########################################################################
# This program is free software; you can redistribute it and/or modify #
# it under the terms of the GNU General Public License as published by #
# the Free Software Foundation; either version 2 of the License, or    #
# (at your option) any later version.                                  #
# This program is distributed in the hope that it will be useful, ######
# but WITHOUT ANY WARRANTY; without even the implied warranty of  #
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the   #
# GNU General Public License for more details.                    #
#                                                              #####
# The GNU General Public License version 2 can be found here: #
# http://www.gnu.org/licenses/old-licenses/gpl-2.0.html       #
##########################################################################
# When using these functions in your work, please cite:                  #
# Breitling LP. dagR: a suite of R functions for directed acyclic graphs. #
# Epidemiology (2010)                                                    #
##########################################################################

writeLines("dagR v1.0.1");
writeLines("Copyright Lutz Philipp Breitling 2010");
writeLines("Please find the license information (GPL-2) at the top of the script file.");
writeLines("When using these functions for your work, please cite:");
writeLines("Breitling LP. dagR: a suite of R functions for directed acyclic graphs.");
writeLines("Epidemiology (2010).");
writeLines("For future versions and comments, please contact lutz.breitling@gmail.com.");

dag.init<-function(outcome=NULL, exposure=NULL, covs=c(), arcs=c(), assocs=c(), xgap=0.04, ygap=0.05, len=0.1,
                   y.name=NULL, x.name=NULL, cov.names=c(),...)
{ # covs: 1 for a covariable, 2 for an unknown;
  # arcs: the numbering refers to the covs vector, i.e. it
  #       differs from the later numbering in the DAG objects;
  #       exposure X is 0, outcome Y is -1;

  dag.out<-c();
  # creating first coordinates for covariates
  i1<-0;
  cl<-length(covs);
  node.x<-c(0);
  node.y<-c(0);
  while (i1 < cl)
  {
    i1<-i1+1;
    i1.deg<-i1*pi/(cl+1);
    node.x<-c(node.x, 0.5-cos(i1.deg)*0.7);
    node.y<-c(node.y, sin(i1.deg)*0.7);
  }
  node.x<-c(node.x, 1);
  node.y<-c(node.y, 0);

  arcs[arcs<0]<-cl+1;
  arcs<-arcs+1;
  curve.x<-rep(NA, length(arcs)/2);
  curve.y<-rep(NA, length(arcs)/2);

  # set arc.type to default 0, and to 1 for associations
  arc.type<-rep(0, length(arcs)/2);
  i2<-0;
  while (i2 < length(assocs))
  {
    i2<-i2+1;
    arc.type[assocs[i2]]<-1;
  }

  # use standard names for X and Y if none provided;
  if(is.null(y.name)) y.name<-"outcome";
  if(is.null(x.name)) x.name<-"exposure";

  # if not enough labels for the covs, use standard
  if(length(cov.names)<cl)
  { standard.names<-c("covariable", "unknown");
    while(length(cov.names)<cl)
    { cov.names<-c(cov.names, standard.names[covs[length(cov.names)+1]]);
    }
  }
  node.names<-c(x.name, cov.names, y.name);

  dag.out$cov.types<-c(0, covs, -1);
  dag.out$x<-node.x;
```

```
  dag.out$y<-node.y;
  dag.out$arc<-matrix(arcs, ncol=2, byrow=T);
  dag.out$arc.type<-arc.type;
  dag.out$curve.x<-curve.x;
  dag.out$curve.y<-curve.y;
  dag.out$xgap<-xgap;
  dag.out$ygap<-ygap;
  dag.out$len<-len;
  dag.out$names<-node.names;
  dag.out$adj<-c();

  return(dag.out);
}



dag.draw<-function(dag, legend=T, paths=T, numbering=F, p=F,...)
{ y.low<-(-0.25);
  if( legend==T && (paths==T && is.numeric(dag$pathsN)) )
  { y.low<-y.low-max(c(0.05*(length(dag$x)-4), 0.15+0.065*(dag$pathsN-5)));
  } else
  { if(legend==T)
    { y.low<-y.low-0.05*(length(dag$x)-4);
    } else
    if( paths==T && is.numeric(dag$pathsN) )
    { y.low<-y.low-(0.15+0.065*(dag$pathsN-5));
    }
  }

  y.high<-max(c(1, max(dag$y)+0.25));
  x.left<-min(c(-0.25, min(dag$x)-0.25));
  x.right<-max(c(1.25, max(dag$x)+0.25));
  plot(x=c(x.left, x.right),
       y=c(y.low, y.high), type="n", axes=F, xlab='', ylab='');

  text(dag$x[1], dag$y[1], expression(X));
  text(dag$x[length(dag$x)], dag$y[length(dag$y)], expression(Y));
  garrows(dag$x[1], dag$y[1], dag$x[length(dag$x)], dag$y[length(dag$y)], dag$xgap, dag$ygap, dag$len);
  text(0.5*(dag$x[1]+dag$x[length(dag$x)]), 0.5*(dag$y[1]+dag$y[length(dag$y)])+dag$ygap/2, "?");

# write covariate symbols
  i1<-1;
  i_c<-0; # covariable counter for subscripts
  i_u<-0; # unknown covs counter for...
  nodes<-length(dag$x);
  while (i1 < nodes-1)
  {
    i1<-i1+1;
    if(dag$names[i1]=="unknown" || dag$cov.types[i1]==2)
    { i_u<-i_u+1;
      text(dag$x[i1], dag$y[i1], bquote(U[.(i_u)]));
    } else
    { i_c<-i_c+1;
      if(is.in(i1, dag$adj)==T)
      { text(dag$x[i1], dag$y[i1], bquote(underline(bar(C))[.(i_c)]));
      } else
      { text(dag$x[i1], dag$y[i1], bquote(C[.(i_c)]));
      }
    }
  }

# draw legend
  if(legend==T) dag.legend(dag);

# write paths
  if(paths==T) write.paths(dag);

# drawing arcs or associations
  i2<-1;
  while (i2 <= length(dag$arc[,1]))
  {
    if(dag$arc.type[i2]==0)
    { garrows(dag$x[dag$arc[i2,1]], dag$y[dag$arc[i2,1]], dag$x[dag$arc[i2,2]], dag$y[dag$arc[i2,2]], dag$xgap,
dag$ygap, dag$len);
    } else if(dag$arc.type[i2]==1)
    {
      if(is.na(dag$curve.x[i2])==T)
      { x0<-dag$x[dag$arc[i2,1]]; y0<-dag$y[dag$arc[i2,1]];
        x1<-dag$x[dag$arc[i2,2]]; y1<-dag$y[dag$arc[i2,2]];
        dag$curve.x[i2]<- -(y1-y0)/15+(x0+x1)/2;
        dag$curve.y[i2]<-  (x1-x0)/15+(y0+y1)/2;
      }
      smoothArc(A=c(dag$x[dag$arc[i2,1]], dag$y[dag$arc[i2,1]]),
                B=c(dag$x[dag$arc[i2,2]], dag$y[dag$arc[i2,2]]),
                C=c(dag$curve.x[i2], dag$curve.y[i2]),
```

```
                gap=max(dag$xgap, dag$ygap), p=p);
    }
    if(numbering==T)
    { if(dag$arc.type[i2]!=1)
      { text(0.5*(dag$x[dag$arc[i2,1]]+dag$x[dag$arc[i2,2]]),
             0.5*(dag$y[dag$arc[i2,1]]+dag$y[dag$arc[i2,2]]),
             i2);
      } else
      { text(0.5*(dag$x[dag$arc[i2,1]]+dag$curve.x[i2]),
             0.5*(dag$y[dag$arc[i2,1]]+dag$curve.y[i2]),
             i2);
      }
    }
   i2<-i2+1;
  }
  return(dag);
}



garrows<-function(x0,y0,x1,y1, xgap, ygap, len=0.1)
{ # draws the arrows in the DAG...;
  if((x0<x1)&&(abs(atan((y1-y0)/(x1-x0))*180/pi)<60))
  { xx0<-x0+xgap;
    xx1<-x1-xgap;
  } else if ((x0>x1)&&(abs(atan((y1-y0)/(x1-x0))*180/pi)<60))
  { xx0<-x0-xgap;
    xx1<-x1+xgap;
  } else
  { xx0<-x0;
    xx1<-x1;
  }
  if((y0<y1)&&(abs(atan((y1-y0)/(x1-x0))*180/pi)>30))
  { yy0<-y0+ygap;
    yy1<-y1-ygap;
  } else if ((y0>y1)&&(abs(atan((y1-y0)/(x1-x0))*180/pi)>30))
  { yy0<-y0-ygap;
    yy1<-y1+ygap;
  } else
  { yy0<-y0;
    yy1<-y1;
  }
  arrows(xx0, yy0, xx1, yy1, length=len);
}


dag.legend<-function(dag, lx=-0.15, ly=-0.075)
{ # write legend
  i1<-0;
  i_c<-0; # covariable counter for subscripts
  i_u<-0; # unknown covs counter for...
  nodes<-length(dag$x);
  while (i1 < nodes)
  {
    i1<-i1+1;
    text(lx, ly-0.05*i1, i1);
    if(dag$names[i1]=="unknown" || dag$cov.types[i1]==2)
    { i_u<-i_u+1;
      text(lx+0.1, ly-0.05*i1, bquote(U[.(i_u)]));
    } else
    { if(i1==1)
      { text(lx+0.1, ly-0.05*i1, expression(X));
      } else
      { if(i1==nodes)
        { text(lx+0.1, ly-0.05*i1, expression(Y));
        } else
        { i_c<-i_c+1;
          text(lx+0.1, ly-0.05*i1, bquote(C[.(i_c)]));
        }
      }
    }
    text(lx+0.15, ly-0.05*i1, dag$names[i1], pos=4);
  }
}



write.paths<-function(dag, px=0.5, py=-0.060)
{ # writes the paths in the DAG graph;
 if(is.null(dag$paths)==F)
 {
  #if(dag$pathsN==1){ dag$paths<-matrix(dag$paths, nrow=1); }

  i1<-0;                          # path number
  max.l<-0;                       # maximal path length
```

```
  while (i1 < dag$pathsN)
  {
    i1<-i1+1;
    cur.node<-1;                # current node
    dag.letter(dag, 1, x=px, y=py-i1*0.065);
    i2<-1; # arc counter
    arc.typ<-'';
    while(is.na(dag$paths[i1, i2])==FALSE)
    {
      if( (dag$arc[dag$paths[i1, i2], 1] != cur.node) )
      { cur.node<-dag$arc[dag$paths[i1, i2], 1];
        arc.typ<-'<';
      } else
      { cur.node<-dag$arc[dag$paths[i1, i2] ,2];
        arc.typ<-'>';
      }
      if(dag$arc.type[dag$paths[i1, i2]]==1)
      { arc.typ<-'-';
      }
      text(px+i2*0.1-0.05, py-i1*0.065, arc.typ);
      dag.letter(dag, cur.node, x=px+i2*0.1, y=py-i1*0.065);
      i2<-i2+1;
      if(i2>max.l){max.l<-i2;}
    }
  }
  for(i1 in 1:dag$pathsN)
  { text(px+max.l*0.1-0.05, py-i1*0.065, dag$path.status[i1], pos=4);
  }
 }
}



smoothArc<-function(A=c(4,1), B=c(5,9), C=c(3,5), res=20, gap=0.05, p=F)
{ angBA<-angle(B,A);
  angBC<-angle(B,C);
  beta1<-inAngle(angBA, angBC);

  lenBD<-cos(beta1)*distPoints(B,C); # cos(beta1)=[BD]/[BC];
  D<-anglePoint(B, angBA, lenBD);

  lenCD<-distPoints(C, D);
  lenBC<-distPoints(B, C);

  lenBM<-(lenCD^2+lenBD^2)/(2*lenCD);

  beta<-acos((lenBC/2)/lenBM);

  angBM<-addAngle(angBC, -sign(beta1)*beta);
  M<-anglePoint(B, angBM, lenBM);

  angMC<-angle(M,C);
  angMB<-angle(M,B);
  stepWidth<-inAngle(angMC, angMB);
   stepWidth<-sign(stepWidth)*(abs(stepWidth)-gap/lenBM);
   stepWidth<-stepWidth/res;

  circ1_ang<-seq(from=angMC, by=stepWidth,
                 length.out=res+1);
  points(matrix(anglePoint(M, circ1_ang, lenBM), ncol=2),
         type='l', lty=2);

  angAB<-angle(A,B);
  angAC<-angle(A,C);
  beta1<-inAngle(angAB, angAC);

  lenAD<-cos(beta1)*distPoints(A,C);
  D<-anglePoint(A, angAB, lenAD);

  lenCD<-distPoints(C, D);
  lenAC<-distPoints(A, C);

  lenAM<-(lenCD^2+lenAD^2)/(2*lenCD);

  beta<-acos((lenAC/2)/lenAM);

  angAM<-addAngle(angAC, -sign(beta1)*beta);
  M<-anglePoint(A, angAM, lenAM);

  angMC<-angle(M,C);
  angMA<-angle(M,A);
  stepWidth<-inAngle(angMC, angMA);
   stepWidth<-sign(stepWidth)*(abs(stepWidth)-gap/lenAM);
   stepWidth<-stepWidth/res;
```

```
 circ2_ang<-seq(from=angMC, by=stepWidth,
                length.out=res+1);
  points(matrix(anglePoint(M, circ2_ang, lenAM), ncol=2),
         type='l', lty=2);

  if(p==T) {
    points(C[1], C[2]);
  }
}



dag.letter<-function(dag, letter, x, y)
{ # function to draw the letters in the DAG;
 if(letter==1)
 { text(x, y, expression(X));
 } else
 if(letter==length(dag$x))
 { text(x, y, expression(Y));
 } else
 {
  i_c<-0; # covariable counter for subscripts
  i_u<-0; # unknown covs counter for...
  for(i1 in 2:letter)
  {
    if(dag$names[i1]=="unknown" || dag$cov.types[i1]==2)
    { i_u<-i_u+1;
    } else
    { i_c<-i_c+1;
    }
  }

  if(dag$names[letter]=="unknown" || dag$cov.types[i1]==2)
  {
    text(x, y, bquote(U[.(i_u)]));
  } else
  { if(is.in(letter, dag$adj)==T)
      { text(x, y, bquote(underline(bar(C))[.(i_c)]));
      } else
      { text(x, y, bquote(C[.(i_c)]));
      }
  }
 }
}


angle<-function(A,B)
{ # internally used by smoothArc();
  rv<-atan((B[1]-A[1])/(B[2]-A[2]));
  if(B[2]<A[2]) rv<-rv-sign(rv)*pi;
  if(rv==0 && B[2]<A[2]) rv<-pi;

  rv<-(rv+pi)%%(2*pi);    # this changes the range from (-pi, pi);
                          # to (0, 2pi);

  return(rv);
}

inAngle<-function(a,b)
{ # internally used by smoothArc();
  rv<-b-a;
  if(abs(rv)>pi) rv<-sign(rv)*(-1)*(2*pi-abs(rv));
  return(rv);
}

anglePoint<-function(A, angl, len)
{ # internally used by smoothArc();
  c(   A[1]-sin(angl)*len,
       A[2]-cos(angl)*len);
}

distPoints<-function(A,B)
{ # internally used by smoothArc();
  rv<-sqrt((B[1]-A[1])^2+(B[2]-A[2])^2);
}

addAngle<-function(a,b)
{ # internally used by smoothArc();
  (a+b)%%(2*pi);
}



find.paths<-function(dag)
{ # identifies backdoor paths, open and closed ones;
```

```
finish<-F;
backdoorpaths<-NULL;
pathsN<-0;
zaehler<-0;
cpos<-0;                # cpos = current position
X<-1;                   # exposure node
Y<-length(dag$names);    # outcome node

  # search initial backdoor path
  used<-c(T, rep(F, length(dag$names)-1));
  cpos<-X;
  cpath<-c();                           # cpath = current path
  cycledPath<-c(0);  # Stores, how far I cycled through to move on from node.
  finish2<-F;
  if(length(dag$arc[,1])==0)
  { finish2<-T;
  }
  stepback<-F;
  search1<-1;
  while (finish2==F)
  {
    if(     # path not connected.
            (dag$arc[search1,1]!=cpos && dag$arc[search1,2]!=cpos)
            # path connected but target node already used.
        ||  (dag$arc[search1,1]==cpos && used[dag$arc[search1,2]]==T)
        ||  (dag$arc[search1,2]==cpos && used[dag$arc[search1,1]]==T)
            # path arrived at outcome.
        ||  (cpos==Y)
            # already cycled too far.
        ||  (stepback==T)
      )
    {
      # If i'm at X && starting only && already cycled through all arcs, quit.
      if(cpos==X && length(cpath)==0 && search1==length(dag$arc[,1]))
      { finish2<-T;
      } else
      {         # If i have arrived at Y and need to search next path.
                # OR
                # If i'm anywhere (e.g. returned to X via a loop),
                # and can't move, and checked all paths,
                # go one step back (rechange cpos, rechange used,
                #                   shorten cpath, shorten cycledPath) and
                # search from next arc on, only!
        if( cpos==Y || stepback==T || search1==length(dag$arc[,1]) )
        {
          stepback<-F;
          used[cpos]<-F;
          lastStep<-cpath[length(cpath)];
          if(cpos!=dag$arc[lastStep,1])
          {        cpos<-dag$arc[lastStep,1];
          } else { cpos<-dag$arc[lastStep,2];
          }
          cpath<-cpath[-length(cpath)];
          search1<-cycledPath[length(cycledPath)-1];
          cycledPath<-cycledPath[-length(cycledPath)];
          cycledPath[length(cycledPath)]<-0;
        }
      }
    } else
    {       # Arc is valid.
            # Add it to the current path.
            # Update the current position.
            # Mark new node as used.
            # Reset the search counter, as all arcs are available again.
            # Store it in cycledPath, and elongate cycledPath.
      cpath<-c(cpath, search1);
      if(cpos!=dag$arc[search1,1])
      {        cpos<-dag$arc[search1,1];
      } else { cpos<-dag$arc[search1,2];
      }
      used[cpos]<-T;
      cycledPath[length(cycledPath)]<-search1;
      cycledPath<-c(cycledPath, 0);
      search1<-0;
      if(cpos==Y)  # If it reaches OUTCOME, write it into backdoorpaths!
      { goodpath<-c(cpath, rep(NA, length(dag$arc[,1])-length(cpath)));
        backdoorpaths<-rbind(backdoorpaths, goodpath, deparse.level=0);
        pathsN<-pathsN+1;
      }
    }
    search1<-search1+1;
    if(search1>length(dag$arc[,1]))
    { search1<-search1-1;
      stepback<-T;
    }
```

```
    }

  # delete paths, that start with a directed arc from the exposure
  # (i. e. you do not want to consider exposure effects as backdoor)
  if(is.null(backdoorpaths)==FALSE)
  {
    i<-0;
    del.vec<-rep(T, length(backdoorpaths[,1]));
    while(i < length(backdoorpaths[,1]))
    {
      i<-i+1;
      if( (     dag$arc[backdoorpaths[i,1],1] == 1) &&
          (dag$arc.type[backdoorpaths[i,1]]   != 1) )
      {
        del.vec[i]<-F;
        pathsN<-pathsN-1;
      }
    }
    backdoorpaths<-backdoorpaths[del.vec,];
  }

  dag$pathsN<-pathsN;
  if(dag$pathsN==1){ backdoorpaths<-matrix(backdoorpaths, nrow=1); }

  # some of the later function require an NA as the last value of the paths;
  if(is.null(backdoorpaths)==FALSE)
  { backdoorpaths<-cbind(backdoorpaths, rep(NA, nrow(backdoorpaths)));
  }

  dag$paths<-backdoorpaths;

  return(dag);
}



eval.paths<-function(dag)
{ # checks if paths are open or blocked;
  # if both blocked "by collider" and "by adjustment", collider dominates;
  if(is.null(dag$paths)==TRUE || dag$pathsN==0)
  { path.status<-NULL;
  } else
  { path.status<-rep('open', dag$pathsN); # first, all are open
    for(i in 1:dag$pathsN)
    {
      i2<-1;
      while(is.na(dag$paths[i, i2+1])==F) # continue while another arrow follows
      { if( # collider present if arcs i and i2 point to same node
                (dag$arc[dag$paths[i, i2], 2] == dag$arc[dag$paths[i, i2+1], 2])
            # and neither one is an association
            && (dag$arc.type[dag$paths[i, i2]]   != 1)
            && (dag$arc.type[dag$paths[i, i2+1]] != 1) )
        { path.status[i]<-'blocked by collider';
          i2<-length(dag$paths[i,])-2;
        } else
        { if(   (is.in( as.numeric( dag$arc[dag$paths[i, i2], 1]  ), dag$adj) == TRUE)
             || (is.in( as.numeric( dag$arc[dag$paths[i, i2], 2]  ), dag$adj) == TRUE)
             || (is.in( as.numeric( dag$arc[dag$paths[i, i2+1], 1]), dag$adj) == TRUE)
             || (is.in( as.numeric( dag$arc[dag$paths[i, i2+1], 2]), dag$adj) == TRUE) )
          {
            path.status[i]<-'blocked by adjustment';
          }
        }
        i2<-i2+1;
      }
    }
  }
  dag$path.status<-path.status;
  return(dag);
}


is.in<-function(x,c=NULL)
{ # used internally by eval.paths();
  i<-0;
  rv<-FALSE;
  while(i<length(c))
  {
    i<-i+1;
    if(identical(x,as.numeric(c[i]))==T)
    {
      rv<-TRUE;
      i<-length(c);
    }
  }
```

```
  is.in<-rv;
}



dag.adjust<-function(dag,A=c())
{ # wrapper for dag.adjustment() and/or find.paths() and eval.paths();
  if(length(A)>0)
  { dag<-dag.adjustment(dag,A);
  } else
  { writeLines('The adjustment set is empty. Function dag.adjust does not call function dag.adjustment, but only
find.paths and eval.paths.');
  }
  dag<-find.paths(dag);
  dag<-eval.paths(dag);
  return(dag);
}


dag.adjustment<-function(dag,A)
{ # called internally by dag.adjust();
  # identifies the associations introduced by adjusting for A;

  dag$adj<-A;

  dag$pathsN<-NULL;
  dag$paths<-NULL;

# find all ancestors of A
  ancs<-dag.ancestors(dag, A);

# put in all associations for all ancestors
  # for each ancestor
  for(i in 1:length(ancs))
  {
    # 1. Look up all parents.
    #     For this step, I also have to include the x->y arrow,
    #     as is also done inside dag.ancestors().
    arcs<-rbind(dag$arc, c(1,length(dag$names)));
    types<-c(dag$arc.type, 0);
    parents<-c();
    i2<-0;
    while(i2<length(arcs[,2]))
    { i2<-i2+1;
      if(arcs[i2,2]==ancs[i] && types[i2]!=1)
      { parents<-c(parents, arcs[i2,1]);
      }
    }
    # 2. if >1 parents, introduce associations if none yet present
    if(length(parents)>1)
    { # examine--sort of--a triangular matrix of the parents
      for(i3 in 2:length(parents))
      { for(i4 in 1:(i3-1))
        { # is there already an association?
          if(assoc.exists(dag, parents[i3], parents[i4])==F)
          {
            dag$arc<-rbind(dag$arc, c(parents[i3], parents[i4]));
            dag$arc.type<-c(dag$arc.type, 1);
          }
        }
      }
    }
  }
  return(dag);
}


dag.ancestors<-function(dag,A)
{ # identify ancestors of A;
  ancest<-c();
  rv<-c(A);

  # add the X-Y arrow for determination of ancestors;
  # also keep all other arrows emenating from X;
  arcs<-rbind(dag$arc, c(1,length(dag$names)));
  types<-c(dag$arc.type, 0);

  # search for immediate ancestors (parents)
  for(i0 in 1:length(A))
  {
    i<-0;
    while(i<length(arcs[,2]))
    { i<-i+1;
      # only use directed arrows to identify ancestors;
      if(arcs[i,2]==A[i0] && types[i]!=1)
```

```
      { ancest<-c(ancest, arcs[i,1]);
      }
    }
  }
  if(length(ancest)>0)
  { rv<-c(A,dag.ancestors(dag,ancest));
  }
  return(unique(rv));
}


assoc.exists<-function(dag, a, b)
{ # internally used by dag.adjustment;
  # checks, if an association already exists, i.e. doesn't need to be introduced;
  rv<-FALSE;
  for(i in 1:length(dag$arc[,2]))
  {
    if( ( (dag$arc[i,1]==a && dag$arc[i,2]==b) ||
          (dag$arc[i,2]==a && dag$arc[i,1]==b)
        ) &&
        (dag$arc.type[i]==1) )
    {
      rv<-TRUE;
    }
  }
  return(rv);
}



dag.move<-function(dag)
{ # first mouse click identifies node or smoothArc to be moved;
  # right click stops the moving;
  dag<-dag.draw(dag, p=T);

  nodes<-length(dag$x);
  arcs<-length(dag$curve.x);

  i.move<-NULL;
  while(is.null(i.move)==T || is.na(i.move)==T)
  { i.move<-identify(x=c(dag$x, dag$curve.x),
                     y=c(dag$y, dag$curve.y), n=1, plot=F);
  }

  new.xy<-locator(n=1);
  while(is.null(new.xy)==F)
  {
    if(i.move>nodes)
    { dag$curve.x[i.move-nodes]<-new.xy$x;
      dag$curve.y[i.move-nodes]<-new.xy$y;
    } else
    { dag$x[i.move]<-new.xy$x;
      dag$y[i.move]<-new.xy$y;
    }
    dag<-dag.draw(dag, p=T);
    new.xy<-locator(n=1);
  }
  return(dag);
}



is.acyclic<-function(dag)
{ # Function to check by try() if a DAG appears cyclic.
  # using dag.ancestors, this only evaluates directed arcs;

  acyclic<-rep(TRUE,length(dag$names));
  for(i in 1:length(dag$names))
  {
    error<-try(dag.ancestors(dag, i), silent=T);
    if(regexpr(pattern='infinite recursion', text=error[1])[1]>=0)
    {
      acyclic[i]<-FALSE;
    }
  }

  if(length(acyclic[acyclic==F])==0)
  { overall<-T;
  } else overall<-F;

  rv<-list(overall, acyclic);
  names(rv)<-c('acyclic','nodewise');
  return(rv);
}
```

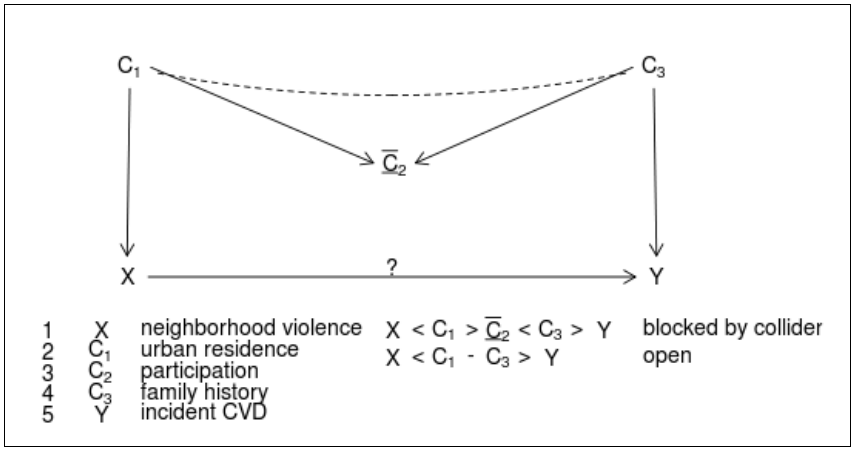
```
brute.search<-function(dag, allow.unknown=F, trace=T, stop=0)
{ # brute search for adjustment sets,
  # i.e. all possible adjustment sets are evaluated;
  # default is that "unknown" covariables are not evaluated;
  # if the input DAG is adjusted, only sets incl. these adjusted covs are evaluated;
  # stop: 0 means no stopping, 1 means stop after 1st sufficient set;
  covsN<-length(dag$names)-2;
  allSets<-c();
  totalPaths<-c();
  openPaths<-c();
  if(covsN>0)
  {
    if(is.null(dag$adj)==T)
    { force<-c();
    } else
    { force<-dag$adj;
    }

    if(allow.unknown==T)
    { allSets<-allCombs(2:(1+covsN), force);
    } else
    { allCovariables<-c(2:(1+covsN));
      theUnknowns<-sapply(X=allCovariables,
                          function(x){ is.unknown(x, dag);})
      theKnowns<-theUnknowns==F;
      knownCovs<-allCovariables[theKnowns];
      allSets<-allCombs(knownCovs, force);
    }

    for(i in 1:dim(allSets)[1])
    { currentSet<-as.vector(na.omit(allSets[i,]));
      dag.temp<-dag.adjust(dag, currentSet);

      totalPaths<-c(totalPaths, dag.temp$pathsN);

      if(dag.temp$pathsN==0)
      { openPaths<-c(openPaths, 0);
      } else
      { openPaths<-c(openPaths,
                     dag.temp$pathsN-sum(dag.temp$path.status!='open'));
      }

      if(trace==T)
      { writeLines(paste(c('set:', currentSet), collapse=' '));
        writeLines(paste(c('paths:', dag.temp$pathsN,
                           '; open:', openPaths[length(openPaths)]),
                         collapse=' '));
      }

      if( openPaths[length(openPaths)]==0 && stop==1 )
      { totalPaths<-c(totalPaths, rep(NA, dim(allSets)[1]-length(openPaths)));
        openPaths<-c(openPaths, rep(NA, dim(allSets)[1]-length(openPaths)));
        break();
      }
    }
  }
  rv<-data.frame(allSets, totalPaths, openPaths);
}


allCombs<-function(x, force=c(), trace=F)
{ # internally used by brute.search;
  # creates eligible adjustment set combinations;
  combs<-NULL;

  # Remove the variables, that are forced to stay in,
  # from the set from which permutations are created.
  # The forced ones will be put in front of each premutation.
  if(length(force)>0)
  { remove.x<-c();
    for(i in 1:length(x))
    { if(trace==T) writeLines(paste(x[i]));
      if(is.in(as.numeric(x[i]), c=force)==T)
      { remove.x<-c(remove.x, i);
        if(trace==T) writeLines('removing');
    } }
    if(length(remove.x)>0)
    { x<-x[-remove.x];
  } }

  if(is.null(x)==F)
  { for(i in 0:length(x))
    { if(length(x)>1)
```

```
      { new.combs<-t(combn(x, i));
      } else
      { if(i==0) { new.combs<-as.matrix(NA);
        } else
        { new.combs<-as.matrix(x);
        }
      }
      new.combs<-cbind(new.combs,
                       matrix(rep(NA, dim(new.combs)[1]*
                                      (length(x)-dim(new.combs)[2])),
                              nrow=dim(new.combs)[1]));
      combs<-rbind(combs, new.combs);
  } }

  if(length(force)!=0)
  {  combs<-cbind(matrix(rep(c(force), dim(combs)[1]),
                         nrow=dim(combs)[1], byrow=T),
                  combs);
  }
  return(combs);
}


is.unknown<-function(x,dag)
{ # internally used by brute.search();
  (dag$names[x]=='unknown' || dag$cov.types[x]==2);
}



add.node<-function(dag, name="unknown", type=1, x=NA, y=NA)
{ # this adds a node in a convenient way;
  # it requires a function because the node is inserted
  # at the 2nd-to-last position in the vectors...;
  nodesN<-length(dag$x);
  if(is.na(x)) x<-0.5*(min(dag$x)+max(dag$x))+sin(nodesN)*(max(dag$x)-min(dag$x))/4;
  if(is.na(y)) y<-0.5*(min(dag$y)+max(dag$y))+cos(nodesN)*(max(dag$y)-min(dag$y))/4;
  dag$cov.types<-c(dag$cov.types[1:(nodesN-1)], type, dag$cov.types[nodesN]);
  dag$x<-c(dag$x[1:(nodesN-1)], x, dag$x[nodesN]);
  dag$y<-c(dag$y[1:(nodesN-1)], y, dag$y[nodesN]);
  dag$names<-c(dag$names[1:(nodesN-1)], name, dag$names[nodesN]);
  dag$arc<-matrix(sapply(X=dag$arc,
                  FUN=function(x){if(x==nodesN) nodesN+1 else x;}),
                  byrow=F, ncol=2);
  return(dag);
}


add.arc<-function(dag, arc, type=0)
{ # this adds arcs;
  # type=0 (directed; default); type=1 (association);
  # it removes those parts of the DAG that need to be re-obtained;
  # CAVE: it uses the node numbering of the DAG, not those used in dag.init;
  dag$arc<-rbind(dag$arc, arc);
  dag$arc.type<-c(dag$arc.type, type);
  dag$paths<-NULL;
  dag$pathsN<-NULL;
  dag$path.status<-NULL;
  return(dag);
}


rm.arc<-function(dag, arc)
{ # this conveniently removes an arc;
  # it removes the evaluated paths etc;
  dag$arc<-dag$arc[-arc,];
  dag$arc.type<-dag$arc.type[-arc];
  dag$curve.x<-dag$curve.x[-arc];
  dag$curve.y<-dag$curve.y[-arc];
  dag$pathsN<-NULL;
  dag$paths<-NULL;
  dag$path.status<-NULL;
  return(dag);
}

rm.node<-function(dag, node)
{ # this conveniently removes a node;
  # also removes all paths, if an arc is removed;
  dag$cov.types<-dag$cov.types[-node];
  dag$x<-dag$x[-node];
  dag$y<-dag$y[-node];
  for(i in nrow(dag$arc):1)
  { if( (dag$arc[i,1]==node) || (dag$arc[i,2]==node) )
    { dag<-rm.arc(dag, i);
    }
```

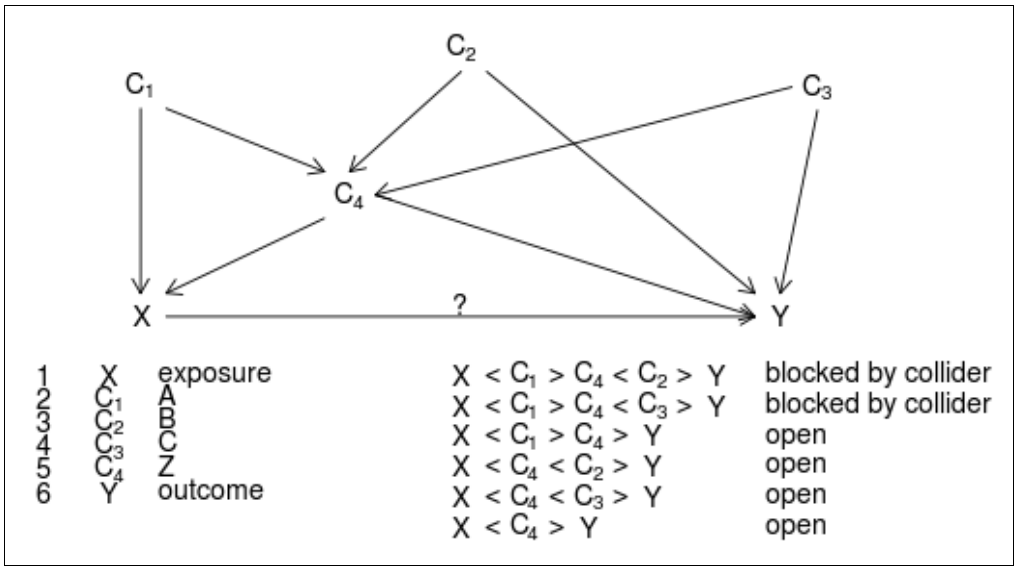
```
  }
  dag$arc<-matrix(sapply(X=dag$arc,
                 FUN=function(x){if(x>node) x-1 else x;}),
                 byrow=F, ncol=2);
  dag$names<-dag$names[-node];
  dag$adj<-dag$adj[dag$adj!=node];
  return(dag);
}


demo.dag0<-function()
{ # creates a DAG that was used during development;
  dag<-dag.init(covs=c(1,1), arcs=c(0,2, 1,2, 1,0, -1,2));
  return(dag);
}

demo.dag1<-function()
{ # re-creates DAG in figure 3 of Fleischer & Diez Roux,
  # J Epidemiol Community Health 2008;62:842;
  dag<-dag.init(y.name="incident CVD", x.name="neighborhood violence",
       covs=c(1,1,1),
       cov.names=c("urban residence", "participation", "family history"),
       arcs=c(1,0, 1,2, 3,2, 3,-1));
  dag$x[3]<-0.505; dag$y[3]<-0.269;
  return(dag);
}

demo.dag2<-function()
{ # re-creates DAG in figure 2a of Shrier & Platt,
  # BMC Med Res Methodol 2008;8:20;
  dag<-dag.init(y.name="injury", x.name="warm-up exercises",
       covs=c(1,2,1,1,1,1,1,1,1,1),
       cov.names=c("coach", "genetics",
                   "fitness", "connective tissue disorder",
                   "pre-game proprioception", "motivation, aggression",
                   "fatigue", "contact sport",
                   "previous injury", "tissue weakness",
                   "intra-game proprioception"));
  dag$x<-c(0.000, 0.261, 0.722, 0.494, 0.995, 0.257,
           0.002, 0.723, 0.505, 0.305, 0.998, 0.502, 1.000);
  dag$y<-c(0.000, 0.852, 0.862, 0.761, 0.735, 0.595,
           0.527, 0.611, 0.449, 0.304, 0.401, 0.149, 0.000);

  dag<-add.arc(dag, c(1,12));
  dag<-add.arc(dag, c(2,7));
  dag<-add.arc(dag, c(2,4));
  dag<-add.arc(dag, c(3,4));
  dag<-add.arc(dag, c(3,8));
  dag<-add.arc(dag, c(3,5));
  dag<-add.arc(dag, c(4,6));
  dag<-add.arc(dag, c(4,8));
  dag<-add.arc(dag, c(5,8));
  dag<-add.arc(dag, c(5,11));
  dag<-add.arc(dag, c(6,1));
  dag<-add.arc(dag, c(7,1));
  dag<-add.arc(dag, c(7,10));
  dag<-add.arc(dag, c(8,12));
  dag<-add.arc(dag, c(8,13));
  dag<-add.arc(dag, c(9,12));
  dag<-add.arc(dag, c(9,10));
  dag<-add.arc(dag, c(11,13));
  dag<-add.arc(dag, c(12,13));

  return(dag);
}

demo.dag3<-function()
{ # re-creates DAG example 3 in Knueppel,
  # DAG v0.11 documentation Oct 21, 2009;
  dag<-dag.init(covs=c(1,1,1,1), cov.names=c("A","B","C","Z"),
                arcs=c(1,0, 1,4, 2,4, 2,-1, 3,4, 3,-1, 4,0, 4,-1));
  dag$x<-c(0.000, 0.000, 0.501, 1.058, 0.327, 1.000);
  dag$y<-c(0.000, 0.495, 0.574, 0.491, 0.260, 0.000);
  return(dag);
}

demo.dag4<-function()
{ # creates a miscellaneous DAG;
  # check out adjustment for the exposure's child!
  dag<-dag.init(covs=c(1,1,1), arcs=c(0,1, 0,2, 1,2, 1,3, 3,-1));
  dag$y[3]<-0.25;
  return(dag);
}

demo.dag5<-function()
```

```
{ # creates a miscellaneous DAG;
  # check out adjustment for the outcome's child!
  dag<-dag.init(covs=c(2,1), arcs=c(1,-1, -1,2));
  return(dag);
}

demo.dag6<-function()
{  # creates a miscellaneous DAG;
  dag<-dag.init(covs=c(2,1,1,1,1), arcs=c(1,0, 1,2, 3,2, 3,-1, 4,3, 5,3));
  dag$x<-c(0.000, 0.211, 0.492, 0.492, 0.236, 0.098, 1.000);
  dag$y<-c(0.000, 0.300, 0.300, 0.663, 0.550, 0.816, 0.000);
  return(dag);
}
```

eFigure 1. Output of the function call "dag.draw(dag.adjust(demo.dag1(), A=3))" with dagR v1.0.1, demonstrating the opening of a backdoor path by inappropriate adjustment. Motivated by reference 3 in the main text.



eFigure 2. Output of the function call "dag.draw(dag.adjust(demo.dag3()))" (top) and the result of function call "allSets<-brute.search(demo.dag3())" (bottom; this evaluates all possible adjustment sets; X1 to X4 indicate which nodes [numbering as in the DAG legend] are adjusted for in the respective set) with dagR v1.0.1, reproducing the evaluation of the DAG of example 3 in the DAG program manual (reference 1 in the main text).




```
R> allSets
   X1 X2 X3 X4 totalPaths openPaths
1  NA NA NA NA          6         4
2   2 NA NA NA          6         3
3   3 NA NA NA          6         3
4   4 NA NA NA          6         3
5   5 NA NA NA         26         4
6   2  3 NA NA          6         2
7   2  4 NA NA          6         2
8   2  5 NA NA         26         0
9   3  4 NA NA          6         2
10  3  5 NA NA         26         1
11  4  5 NA NA         26         1
12  2  3  4 NA          6         1
13  2  3  5 NA         26         0
14  2  4  5 NA         26         0
15  3  4  5 NA         26         0
16  2  3  4  5         26         0
R>
```

eFigure 3. Output of the function call "dag.draw(demo.dag2())" with dagR v1.0.1, as an example of a more complicated DAG (motivated by reference 4 in the main text); the evaluation of all possible adjustment sets for identifying sufficient sets using the dagR function "brute.search(demo.dag2())" would take about 50 minutes on an AMD Athlon(tm) 64 processor 3200+ running Ubuntu 9.10 and R 2.9.2.