Chapter 6- Graphs

library(gRbase)  
library(ggm)  
library(igraph)

##   
## Attaching package: 'igraph'

## The following object is masked from 'package:ggm':  
##   
## pa

## The following objects are masked from 'package:gRbase':  
##   
## edges, is\_dag, topo\_sort

## The following objects are masked from 'package:stats':  
##   
## decompose, spectrum

## The following object is masked from 'package:base':  
##   
## union

library(RBGL)

## Loading required package: graph

## Loading required package: BiocGenerics

##   
## Attaching package: 'BiocGenerics'

## The following objects are masked from 'package:igraph':  
##   
## normalize, path, union

## The following objects are masked from 'package:stats':  
##   
## IQR, mad, sd, var, xtabs

## The following objects are masked from 'package:base':  
##   
## anyDuplicated, aperm, append, as.data.frame, basename, cbind,  
## colnames, dirname, do.call, duplicated, eval, evalq, Filter, Find,  
## get, grep, grepl, intersect, is.unsorted, lapply, Map, mapply,  
## match, mget, order, paste, pmax, pmax.int, pmin, pmin.int,  
## Position, rank, rbind, Reduce, rownames, sapply, setdiff, sort,  
## table, tapply, union, unique, unsplit, which.max, which.min

##   
## Attaching package: 'graph'

## The following objects are masked from 'package:igraph':  
##   
## degree, edges, intersection

## The following objects are masked from 'package:gRbase':  
##   
## addEdge, adj, connComp, edges, nodes, removeEdge, subGraph

##   
## Attaching package: 'RBGL'

## The following objects are masked from 'package:igraph':  
##   
## bfs, dfs, transitivity

## The following objects are masked from 'package:gRbase':  
##   
## is.triangulated, maxClique, separates

library(sets)

##   
## Attaching package: 'sets'

## The following object is masked from 'package:igraph':  
##   
## %>%

## The following objects are masked from 'package:gRbase':  
##   
## %>%, closure

## Directed Acyclic Graph

* We begin by specifying the DAG for the circuit example BN.
* To do that, we use the **ggm::DAG()** function
* We specify the DAG as a sequence of formulas. We specify the DAG as a sequence of formulas. Each formula defines a node of the graph prefixed by the tilde symbol, followed by all the node’s parents separated by the plus symbol.
* The DAG returns the adjacency matrix (AMAT) of the DAG. AMAT is an nxn matrix where n is the number of nodes.
* The element in row i and column j in AMAT equals one if there is an edge from node i to node j and equals zero otherwise.
* The names of the rows and columns of the adjacency matrix are the nodes of the DAG.
* If order = TRUE, the adjacency matrix is permuted to have the nodes listed in topological order where parents precede children in the order, with the resulting adjacency matrix being upper triangular.

G <- ggm::DAG(F ~ D+E+A4, D ~ X+Z+A2, E ~ Y+Z+A3, Z ~ X+Y+A1,order=TRUE)  
print(G)

## A1 A3 Y A2 X Z E D A4 F  
## A1 0 0 0 0 0 1 0 0 0 0  
## A3 0 0 0 0 0 0 1 0 0 0  
## Y 0 0 0 0 0 1 1 0 0 0  
## A2 0 0 0 0 0 0 0 1 0 0  
## X 0 0 0 0 0 1 0 1 0 0  
## Z 0 0 0 0 0 0 1 1 0 0  
## E 0 0 0 0 0 0 0 0 0 1  
## D 0 0 0 0 0 0 0 0 0 1  
## A4 0 0 0 0 0 0 0 0 0 1  
## F 0 0 0 0 0 0 0 0 0 0

We use the function **ggm::plotGrap()** to plot the DAG from the adjacency matrix.

ggm::plotGraph(G)

Figure shows the visualization of the DAG for the circuit BN model resulting from the execution of the ggm::plotGraph function. Notice there are four DAG families corresponding to the nodes “Z,” “S,”“E,” and “F,” which are the outputs of the four gates in the circuit. We use the term a DAG family to mean a child node in the DAG and all its parents. We do not use the term DAG family for the root nodes with no parents, which are all the other nodes in the DAG.

We use the **as()** function to coerce the AMAT, G, to be represented as an igraph.

dag <- as(G,"igraph")

We use the **igraph::V()** function to get a list of all vertices (nodes) of the dag.

igraph::V(dag)

## + 10/10 vertices, named, from cb5e28d:  
## [1] A1 A3 Y A2 X Z E D A4 F

We use the **igraph::E()** function to return a list of all edges in the dag.

igraph::E(dag)

## + 12/12 edges from cb5e28d (vertex names):  
## [1] A1->Z A3->E Y ->Z Y ->E A2->D X ->Z X ->D Z ->E Z ->D E ->F D ->F A4->F

### Getting DAG families- nodes and their parents

We use the **gRbase::vpardag()** function to get a list of all families in the dag, i.e., nodes and their parents in the dag. We extract the family for the node of the F variable.

vpardag <- gRbase::vpar(dag)  
vpardag$F

## [1] "F" "E" "D" "A4"

### Ancestral Sets and Ancestral Graphs

We use the gRbase::ancestralSet() function to return the ancestral set of a given set A; the ancestral set of A is the union of A with its ancestors in the dag.

ancestralSet(c("D","E"),dag)

## [1] "A1" "A3" "Y" "A2" "X" "Z" "E" "D"

We use the **gRbase::ancestralGraph()** function to get the ancestral graph of a given set A in the dag. The ancestral graph is the subgraph induced by the ancestral set of A in the dag.

gRbase::ancestralGraph(c("D","E"),dag)

## IGRAPH 220e436 DN-- 8 9 --   
## + attr: name (v/c), label (v/c)  
## + edges from 220e436 (vertex names):  
## [1] A1->Z A3->E Y ->Z Y ->E A2->D X ->Z X ->D Z ->E Z ->D

## Moralization of the DAG

We use the function **gRbase::moralize()** function to create the moral graph for the DAG, *G*. The moral graph is an undirected graph where all the parents of every node of the DAG are connected.

morG <- gRbase::moralize(G)  
print(morG)

## A1 A3 Y A2 X Z E D A4 F  
## A1 0 0 1 0 1 1 0 0 0 0  
## A3 0 0 1 0 0 1 1 0 0 0  
## Y 1 1 0 0 1 1 1 0 0 0  
## A2 0 0 0 0 1 1 0 1 0 0  
## X 1 0 1 1 0 1 0 1 0 0  
## Z 1 1 1 1 1 0 1 1 0 0  
## E 0 1 1 0 0 1 0 1 1 1  
## D 0 0 0 1 1 1 1 0 1 1  
## A4 0 0 0 0 0 0 1 1 0 1  
## F 0 0 0 0 0 0 1 1 1 0

### Print and Plot the undirected moral graph

First, we map the moral graph adjacency matrix *morG* to an **igraph** object Second, we print the nodes and edges of the moral graph Third, We use the function **ggm::plotGraph()** to plot the moral graph, *morG*

m\_ug <- as(morG, "igraph")  
print(m\_ug)

## IGRAPH f61064b UN-- 10 22 --   
## + attr: name (v/c), label (v/c)  
## + edges from f61064b (vertex names):  
## [1] A1--Y A1--X A1--Z A3--Y A3--Z A3--E Y --X Y --Z Y --E A2--X   
## [11] A2--Z A2--D X --Z X --D Z --E Z --D E --D E --A4 E --F D --A4  
## [21] D --F A4--F

We use the function **ggm::plotGraph()** to plot the moral graph, *morG*

ggm::plotGraph(morG)

#### Getting the Moral Subgraphs for the DAG Families

The moral subgraphs of the DAG families form cliques or complete graphs where all nodes are connected. To demonstrate this

* We coerce the moral graph *morG* to be of type igraph object named m\_ug using the as() function.
* We get the moral subgraph for subsets of nodes constituting DAG families.

The following code computes the moral subgraph, ugF, for the DAG family of the node “F” given by the list vpardag$F. Next, we print the graph, which writes the subgraph nodes and edges.

nodes <- vpardag$F  
print(nodes)

## [1] "F" "E" "D" "A4"

m\_ug <- as(morG, "igraph")  
ugF <- subgraph(m\_ug, vpardag$F)  
print(ugF)

## IGRAPH 2f6e2fc UN-- 4 6 --   
## + attr: name (v/c), label (v/c)  
## + edges from 2f6e2fc (vertex names):  
## [1] E --D E --A4 D --A4 E --F D --F A4--F

### A family in the DAG is a complete Subgraph of the moral graph

Query the undirected moral graph to verify that the DAG family for the node “F” is a complete graph. We use the function gRbase::querygraph() to perform the query. For details on the querygraph() function see the webpage at <https://www.rdocumentation.org/packages/gRbase/versions/1.3.5/topics/querygraph>

gRbase::querygraph(ugF, "is.complete")

## [1] TRUE

### Plot the complete subgraph for the DAG family of the node “F”

We use the ggm::plotGraph() to visualize the complete subgraph of the DAG family of node “F.”

ggm::plotGraph(ugF)

### Families in the DAG are complete sets in the moral graph

Verify that all families in the DAG are complete sets in the moral graph. First, we get all the DAG families, then for each of the four families corresponding to the four gates in our circuit BN. Then, as before, we use the function gRbase::querygraph() to verify that each moral subgraph is complete.

vpardag <- gRbase::vpar(dag)  
querygraph(morG, "is.complete", vpardag$Z)

## [1] TRUE

querygraph(morG, "is.complete", vpardag$D)

## [1] TRUE

querygraph(morG, "is.complete", vpardag$E)

## [1] TRUE

querygraph(morG, "is.complete", vpardag$F)

## [1] TRUE

Maximal prime subgraph decomposition - an adjacency matrix

ggm::plotGraph(m\_ug)  
x <- mpdMAT(morG)  
y <- unlist(x$cliques[1])  
g <- igraph::subgraph(m\_ug, y)  
ggm::plotGraph(g)

### Triangulating the moral graph

TmG <- gRbase::minimal\_triangMAT(morG)  
print(TmG)

## A1 A3 Y A2 X Z E D A4 F  
## A1 0 0 1 0 1 1 0 0 0 0  
## A3 0 0 1 0 0 1 1 0 0 0  
## Y 1 1 0 0 1 1 1 0 0 0  
## A2 0 0 0 0 1 1 0 1 0 0  
## X 1 0 1 1 0 1 1 1 0 0  
## Z 1 1 1 1 1 0 1 1 0 0  
## E 0 1 1 0 1 1 0 1 1 1  
## D 0 0 0 1 1 1 1 0 1 1  
## A4 0 0 0 0 0 0 1 1 0 1  
## F 0 0 0 0 0 0 1 1 1 0

Plot the minimally triangulated graph

ggm::plotGraph(TmG)

Check that the triangulated graph is actually triangulated.

Check if the moral graph was already triangulated.

gRbase::is.triangulated(TmG)

## [1] TRUE

gRbase::is.triangulated(morG)

## [1] FALSE

### Maximum cardinality search on the triangulated graph

Get a perfect ordering of the nodes of the triangulated graph using maximum cardinality search. We use the \*\*gRbase::mcsMAT()**\*\*** function which takes the adjacency matrix of an undirected graph as input and returns a perfect ordering of the nodes if it exists. The function provides a check for decomposability of the input undirected graph.

gRbase::mcsMAT(TmG)

## [1] "A1" "Y" "X" "Z" "E" "A3" "D" "A2" "A4" "F"

### Compare the moral and triangulated graphs

Determine if the moral graph and the triangulated are the same and print the edges in both the moral and minimally triangulated graph.

g1 <- as(morG, "igraph")  
g2 <- as(TmG, "igraph")  
igraph::identical\_graphs(g1, g2)

## [1] FALSE

e\_g1 <- igraph::E(g1)  
e\_g2 <- igraph::E(g2)  
print(e\_g1)

## + 22/22 edges from 5d71551 (vertex names):  
## [1] A1--Y A1--X A1--Z A3--Y A3--Z A3--E Y --X Y --Z Y --E A2--X   
## [11] A2--Z A2--D X --Z X --D Z --E Z --D E --D E --A4 E --F D --A4  
## [21] D --F A4--F

print(e\_g2)

## + 23/23 edges from 4a8bca8 (vertex names):  
## [1] A1--Y A1--X A1--Z A3--Y A3--Z A3--E Y --X Y --Z Y --E A2--X   
## [11] A2--Z A2--D X --Z X --E X --D Z --E Z --D E --D E --A4 E --F   
## [21] D --A4 D --F A4--F

Verify that the triangulated graph is chordal.

igraph::is\_chordal(g2)

## $chordal  
## [1] TRUE  
##   
## $fillin  
## NULL  
##   
## $newgraph  
## NULL

simplicial nodes

A node in an undirected graph is simplicial if its boundary is complete.

gRbase::simplicialNodes(g2)

## [1] "A1" "A3" "A2" "A4" "F"

Verify that the adjacent nodes to a simplical node is a complete set.

neighbors(g2, "A1")

## + 3/10 vertices, named, from 4a8bca8:  
## [1] Y X Z

is.complete(subgraph( g2, unlist(neighbors(g2, "A1"))))

## [1] TRUE

### Maximal prime subgraph decomposition

plot the subgraph for each clique, you can extract each clique from x$cliques, convert it to a vector using unlist(), create the subgraph with **igraph::subgraph()**, and then plot it using ggm::plotGraph().

x <- gRbase::mpdMAT(TmG)  
for(i in 1:length(x$cliques)){  
 y <- unlist(x$cliques[i])  
 g <- igraph::subgraph(g2, y)  
 print(ggm::plotGraph(g))  
}

## $tkp.id  
## [1] 7  
##   
## $igraph  
## IGRAPH 705d6c5 DN-- 4 6 --   
## + attr: name (v/c), edge.arrow.mode (e/n)  
## + edges from 705d6c5 (vertex names):  
## [1] A1->Y A1->X A1->Z Y ->X Y ->Z X ->Z  
##   
## $tkp.id  
## [1] 8  
##   
## $igraph  
## IGRAPH d160238 DN-- 4 6 --   
## + attr: name (v/c), edge.arrow.mode (e/n)  
## + edges from d160238 (vertex names):  
## [1] Y->X Y->Z Y->E X->Z X->E Z->E  
##   
## $tkp.id  
## [1] 9  
##   
## $igraph  
## IGRAPH 2804fd2 DN-- 4 6 --   
## + attr: name (v/c), edge.arrow.mode (e/n)  
## + edges from 2804fd2 (vertex names):  
## [1] A3->Y A3->Z A3->E Y ->Z Y ->E Z ->E  
##   
## $tkp.id  
## [1] 10  
##   
## $igraph  
## IGRAPH cf8a271 DN-- 4 6 --   
## + attr: name (v/c), edge.arrow.mode (e/n)  
## + edges from cf8a271 (vertex names):  
## [1] X->Z X->E X->D Z->E Z->D E->D  
##   
## $tkp.id  
## [1] 11  
##   
## $igraph  
## IGRAPH ea1fce5 DN-- 4 6 --   
## + attr: name (v/c), edge.arrow.mode (e/n)  
## + edges from ea1fce5 (vertex names):  
## [1] A2->X A2->Z A2->D X ->Z X ->D Z ->D  
##   
## $tkp.id  
## [1] 12  
##   
## $igraph  
## IGRAPH c5e1944 DN-- 4 6 --   
## + attr: name (v/c), edge.arrow.mode (e/n)  
## + edges from c5e1944 (vertex names):  
## [1] E ->D E ->A4 E ->F D ->A4 D ->F A4->F

## Junction Tree

To compute the junction tree that orders the maximum cliques in a tree-structured representation called a join tree, we use the **gRbase::rip()** function of the gRbase package. The function takes an adjacency matrix representation of a triangulated graph and outputs a join tree data structure. The function prints the maximum cliques and their order, the tree nodes. The RIP function designates each clique node, its parent node, and the list of separators that annotate the edges of the tree that connect the nodes and their parents.

gRbase::rip(TmG)

## cliques  
## 1 : Y X Z A1   
## 2 : Y X Z E   
## 3 : A3 Y E Z   
## 4 : X Z D E   
## 5 : A2 X D Z   
## 6 : F E A4 D   
## separators  
## 1 :   
## 2 : Y X Z   
## 3 : Y E Z   
## 4 : X Z E   
## 5 : X D Z   
## 6 : E D   
## parents  
## 1 : 0   
## 2 : 1   
## 3 : 2   
## 4 : 2   
## 5 : 4   
## 6 : 4