Chapter 6- Graphs

if (!requireNamespace("BiocManager", quietly = TRUE))  
 install.packages("BiocManager")  
  
BiocManager::install("graph")

## Bioconductor version 3.19 (BiocManager 1.30.23), R 4.4.0 (2024-04-24)

## Warning: package(s) not installed when version(s) same as or greater than current; use  
## `force = TRUE` to re-install: 'graph'

# Tree Clustering

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, 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 9d1aed8:  
## [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 9d1aed8 (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.

gRbase::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)

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

## IGRAPH ddd6439 DN-- 8 9 --   
## + attr: name (v/c), label (v/c)  
## + edges from ddd6439 (vertex names):  
## [1] A1->Z A3->E Y ->Z Y ->E A2->D X ->Z X ->D Z ->E Z ->D

### d-separation

We use **igraph::dSep()** to determine if, in a directed acyclic graph *G*, two sets of nodes, *X* and *Y*, are d-separated by a third set *Z* of nodes. The function takes four arguments. The first argument, *G*, is a Boolean matrix representing the adjacency matrix of a directed acyclic graph. The function does not check the graph’s acyclicity. In that order, the following three arguments are nodes X, Y, and Z subsets.

The code below is an example of checking that node {Y,Z} d-separates nodes E and X in the DAG of the circuit example.

(E ⟂ X | Y, Z)

dSep(G, "E", "X", c("Y","Z"))

## [1] TRUE

dSep(G, c("A4","F"), c("A2", "X", "A1", "Y", "A3"), c("D","E","Z"))

## [1] TRUE

## Moralization

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

### Getting the 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 ba18f14 UN-- 10 22 --   
## + attr: name (v/c), label (v/c)  
## + edges from ba18f14 (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)

### DAG Families and their Moral Subgraphs

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 799254c UN-- 4 6 --   
## + attr: name (v/c), label (v/c)  
## + edges from 799254c (vertex names):  
## [1] E --D E --A4 D --A4 E --F D --F A4--F

#### The Moral Subgraphs of DAG Families are complete subgraphs

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

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

### Finding the Maximal Prime subgraph Decomposition of the Moral Graph

The maximal prime subgraph decomposition of a graph is the smallest subgraphs into which the graph can be decomposed.

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

### Algorithm d\_separates

The following code implements the algorithm **d\_separtes** To determine if, in a directed acyclic graph *G*, two sets of nodes, *X* and *Y*, are d-separated by a third set, *Z*. The algorithm checks the input sets to ensure they are mutually pairwise disjoint. Also the algorithm

library(igraph) # for the graph data structures  
library(gRbase) # for the ancestralGraph and moralize functions  
  
d\_separates <- function(dag, X, Y, Z) {  
   
 if (!is.vector(X)) { X <- c(X) }  
 if (!is.vector(Y)) { Y <- c(Y) }  
 if (!is.vector(Z)) { Z <- c(Z) }  
   
 # Ensure X, Y, and Z are disjoint  
 if (length(intersect(X, Y)) > 0 || length(intersect(Y, Z)) > 0 || length(intersect(X, Z)) > 0) {  
 stop("X, Y, and Z must be disjoint sets")  
 }  
   
 # 1. Get the ancestral graph  
 anG <- gRbase::ancestralGraph(union(X, union(Y, Z)), dag)  
 ggm::plotGraph(anG)  
   
 # 2. Moralize the ancestral graph  
 m\_anG <- gRbase::moralize(anG)  
 ggm::plotGraph(m\_anG)  
   
 # 3. Remove all nodes in Z from m\_anG   
 m2\_anG <- delete\_vertices(m\_anG, Z)  
 ggm::plotGraph(m2\_anG)  
  
 # 4. Test if X and Y are connected in m2\_anG  
 are\_connected <- any(sapply(X, function(x) {  
 # If the minimum distance from x to any node in Y is not infinity, they are connected.  
 min(igraph::distances(m2\_anG, v = x, to = Y)) != Inf  
 }))  
  
 return(! are\_connected)  
}

print(dag)

## IGRAPH 9d1aed8 DN-- 10 12 --   
## + attr: name (v/c), label (v/c)  
## + edges from 9d1aed8 (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

d\_separates(dag, "E", "X", c("Y","Z"))

## [1] TRUE

max\_cardinality\_ordering

max\_cardinality\_ordering0 <- function(ug, x = NULL) {  
 # Ensure the graph is undirected  
 if(igraph::is\_directed(ug)) {  
 stop("The input graph must be undirected.")  
 }  
   
 # Initialize node weights  
 weights <- rep(0, vcount(ug))  
 names(weights) <- V(ug)$name  
   
 # If a starting node is provided, use it as the first node  
 if(!is.null(x)) {  
 if(x %in% V(ug)$name) {  
 weights[x] <- max(weights) + 1  
 } else {  
 stop("The specified starting node does not exist in the graph.")  
 }  
 }  
   
 # Construct adjacency list and ordering vector  
 adj\_list <- as\_adj\_list(ug) # changing get.adjlist() to as\_adj\_list()  
 order\_vec <- c()  
   
 # Main loop  
 for(i in 1:vcount(ug)) {  
 # Select a node with maximum weight  
 max\_node <- names(which.max(weights))  
 order\_vec <- c(order\_vec, max\_node)  
   
 # Increment the weights of the neighbors of the maximum node  
 weights[adj\_list[[max\_node]]] <- weights[adj\_list[[max\_node]]] + 1  
   
 # Remove the maximum node from the graph, weights vector and adjacency list  
 ug <- delete\_vertices(ug, max\_node) # changing delete.vertices() to delete\_vertices()  
 weights <- weights[!names(weights) %in% max\_node]  
 adj\_list <- adj\_list[!names(adj\_list) %in% max\_node]  
 }  
   
 # Return the order in which nodes were selected, as a named vector  
 return(setNames(order\_vec, paste0("Node\_", 1:length(order\_vec))))  
}

print(m\_ug)

## IGRAPH 3209627 UN-- 10 22 --   
## + attr: name (v/c), label (v/c)  
## + edges from 3209627 (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

max\_cardinality\_ordering0(m\_ug,"A1")

## Node\_1 Node\_2 Node\_3 Node\_4 Node\_5 Node\_6 Node\_7 Node\_8 Node\_9 Node\_10   
## "A1" "Y" "Z" "A3" "X" "E" "D" "F" "A2" "A4"

max\_cardinality\_ordering <- function(ug, x = NULL) {  
 # Ensure the graph is undirected  
 if(igraph::is\_directed(ug)) stop("The input graph must be undirected.")  
   
 # Construct adjacency list  
 adj\_list <- as\_adj\_list(ug)  
  
 # Number of nodes, and create an empty named vector to hold the result  
 n <- vcount(ug)  
 order\_vec <- character(n)  
   
 # Initialize node weights to 0 and set the names as node names  
 weights <- rep(0, n)  
 names(weights) <- names(adj\_list)  
   
 # If a starting node is provided, and exists in the graph  
 if(!is.null(x) && x %in% names(adj\_list)) {  
 order\_vec[1] <- x # Set the starting node as the first node  
 weights[x] <- NA # So it won't be selected next time  
 }  
   
 # For each node from 2 to n  
 for(i in 2:n) {  
 x <- order\_vec[i-1] # The node selected in the last round  
 if (!is.na(x)) weights[adj\_list[[x]]] <- weights[adj\_list[[x]]] + 1 # Increase the weight of its neighbors  
  
 # From those nodes not already in the order,  
 # select a node with maximum weight randomly and add it to the order  
 remaining\_nodes <- setdiff(names(weights), order\_vec)  
 max\_weight\_node <- remaining\_nodes[which.max(weights[remaining\_nodes])]  
 order\_vec[i] <- max\_weight\_node  
 weights[max\_weight\_node] <- NA # So it won't be selected next time  
 }  
   
 # Return the final dictionary as a named vector  
 return(setNames(order\_vec, paste0("Node\_", 1:n)))  
}

ordering <- max\_cardinality\_ordering(m\_ug,"A1")

## Triangulation

# preserve\_order\_intersect <- function(x, y) {  
# y[match(x, y)]  
# }  
#   
# triangulate <- function(ug, ordering) {  
# if(igraph::is\_directed(ug)) stop("The graph must be undirected.")  
#   
# # Make a copy of the graph and use this for the operations   
# ug\_triangulated <- ug  
#   
# # The ordering is in sequential order while we need it to be reversed  
# r\_ordering <- rev(ordering)  
#   
# # Process each node as per the reversed ordering  
# for(i in seq\_along(r\_ordering)) {  
#   
# # get the i-th node  
# vi\_name <- r\_ordering[i]  
#   
# if (vi\_name %in% V(ug\_triangulated)$name) {  
# # get the nodes before i-th node  
# nodes\_before\_vi <- r\_ordering[seq(i+1, length(r\_ordering))]  
#   
# # get the neighbors of i-th node (which are before i-th node as per the ordering)  
# neighbors\_of\_vi <- preserve\_order\_intersect(nodes\_before\_vi, neighbors(ug\_triangulated, vi\_name)$name)  
#   
# # process each unique pair of nodes  
# for(j in seq\_along(neighbors\_of\_vi)) {  
# for(k in seq(j+1, length(neighbors\_of\_vi))) {  
# node1 <- neighbors\_of\_vi[j]  
# node2 <- neighbors\_of\_vi[k]  
#   
# # check if these nodes are valid and not the same  
# if(node1 %in% V(ug\_triangulated)$name & node2 %in% V(ug\_triangulated)$name & node1 != node2) {  
# node1\_vertex <- V(ug\_triangulated)[node1]  
# node2\_vertex <- V(ug\_triangulated)[node2]  
# if(length(E(ug\_triangulated)[node1\_vertex %--% node2\_vertex]) == 0) {  
# # connect these nodes  
# ug\_triangulated <- igraph::add\_edges(ug\_triangulated, c(node1, node2))  
# }  
# }  
# }  
# }  
# }  
# }  
#   
# return(ug\_triangulated)  
# }

triangulate <- function(ug, ordering) {  
 if(igraph::is\_directed(ug)) stop("The graph must be undirected.")   
  
 ug\_triangulated <- ug   
 r\_ordering <- rev(ordering)  
  
 for(i in seq\_along(r\_ordering)) {  
 vi\_name <- r\_ordering[i]  
 print(paste("Processing node: ", vi\_name))  
  
 if (vi\_name %in% V(ug\_triangulated)$name) {  
 nodes\_before\_vi <- r\_ordering[seq(i+1, length(r\_ordering))]  
  
 all\_neighbors\_vi <- neighbors(ug, vi\_name)$name # Fetch neighbors from the original graph 'ug'  
 neighbors\_of\_vi <- intersect(all\_neighbors\_vi, nodes\_before\_vi)  
  
 print(paste("Neighbors of ", vi\_name, " : ", paste0(neighbors\_of\_vi, collapse = ", ")))  
  
 for(j in seq\_along(neighbors\_of\_vi)) {  
 for(k in seq(j+1, length(neighbors\_of\_vi))) {  
 nodes = c(neighbors\_of\_vi[j], neighbors\_of\_vi[k])  
 nodes = nodes[order(match(nodes, r\_ordering))]  
  
 node1 <- nodes[1]  
 node2 <- nodes[2]  
  
 if(node1 %in% V(ug\_triangulated)$name & node2 %in% V(ug\_triangulated)$name & node1 != node2) {  
 node1\_vertex <- V(ug\_triangulated)[node1]  
 node2\_vertex <- V(ug\_triangulated)[node2]  
 if(length(E(ug\_triangulated)[node1\_vertex %--% node2\_vertex]) == 0) {  
 print(paste("Adding edge between", node1, "and", node2))  
 ug\_triangulated <- igraph::add\_edges(ug\_triangulated, c(node1, node2))  
 }  
 }  
 }  
 }  
 }  
 }  
  
 return(ug\_triangulated)  
}

print(m\_ug)

## IGRAPH 3209627 UN-- 10 22 --   
## + attr: name (v/c), label (v/c)  
## + edges from 3209627 (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(ordering)

## Node\_1 Node\_2 Node\_3 Node\_4 Node\_5 Node\_6 Node\_7 Node\_8 Node\_9 Node\_10   
## "A1" "Y" "X" "Z" "A3" "E" "D" "A2" "A4" "F"

Tmg2 <- triangulate(m\_ug, ordering)

## [1] "Processing node: F"  
## [1] "Neighbors of F : E, D, A4"  
## [1] "Processing node: A4"  
## [1] "Neighbors of A4 : E, D"  
## [1] "Processing node: A2"  
## [1] "Neighbors of A2 : X, Z, D"  
## [1] "Processing node: D"  
## [1] "Neighbors of D : X, Z, E"  
## [1] "Adding edge between E and X"  
## [1] "Processing node: E"  
## [1] "Neighbors of E : A3, Y, Z"  
## [1] "Processing node: A3"  
## [1] "Neighbors of A3 : Y, Z"  
## [1] "Processing node: Z"  
## [1] "Neighbors of Z : A1, Y, X"  
## [1] "Processing node: X"  
## [1] "Neighbors of X : A1, Y"  
## [1] "Processing node: Y"  
## [1] "Neighbors of Y : A1"  
## [1] "Processing node: A1"  
## [1] "Neighbors of A1 : "

print(Tmg2)

## IGRAPH a6102a1 UN-- 10 23 --   
## + attr: name (v/c), label (v/c)  
## + edges from a6102a1 (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 X --E

gRbase::is.triangulated(Tmg2)

## [1] TRUE

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 3b6eef9 (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 9236735 (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 9236735:  
## [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(morG)  
for(i in 1:length(x$cliques)){  
 y <- unlist(x$cliques[i])  
 g <- igraph::subgraph(g2, y)  
 print(ggm::plotGraph(g))  
}

## $tkp.id  
## [1] 10  
##   
## $igraph  
## IGRAPH 5ae77a0 DN-- 4 6 --   
## + attr: name (v/c), edge.arrow.mode (e/n)  
## + edges from 5ae77a0 (vertex names):  
## [1] A1->Y A1->X A1->Z Y ->X Y ->Z X ->Z  
##   
## $tkp.id  
## [1] 11  
##   
## $igraph  
## IGRAPH ab9c487 DN-- 5 9 --   
## + attr: name (v/c), edge.arrow.mode (e/n)  
## + edges from ab9c487 (vertex names):  
## [1] Y->X Y->Z Y->E X->Z X->E X->D Z->E Z->D E->D  
##   
## $tkp.id  
## [1] 12  
##   
## $igraph  
## IGRAPH 6b18ffe DN-- 4 6 --   
## + attr: name (v/c), edge.arrow.mode (e/n)  
## + edges from 6b18ffe (vertex names):  
## [1] A3->Y A3->Z A3->E Y ->Z Y ->E Z ->E  
##   
## $tkp.id  
## [1] 13  
##   
## $igraph  
## IGRAPH 1bd1597 DN-- 4 6 --   
## + attr: name (v/c), edge.arrow.mode (e/n)  
## + edges from 1bd1597 (vertex names):  
## [1] A2->X A2->Z A2->D X ->Z X ->D Z ->D  
##   
## $tkp.id  
## [1] 14  
##   
## $igraph  
## IGRAPH 9b7d087 DN-- 4 6 --   
## + attr: name (v/c), edge.arrow.mode (e/n)  
## + edges from 9b7d087 (vertex names):  
## [1] E ->D E ->A4 E ->F D ->A4 D ->F A4->F

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] 15  
##   
## $igraph  
## IGRAPH d36692e DN-- 4 6 --   
## + attr: name (v/c), edge.arrow.mode (e/n)  
## + edges from d36692e (vertex names):  
## [1] A1->Y A1->X A1->Z Y ->X Y ->Z X ->Z  
##   
## $tkp.id  
## [1] 16  
##   
## $igraph  
## IGRAPH 4f4d1c3 DN-- 4 6 --   
## + attr: name (v/c), edge.arrow.mode (e/n)  
## + edges from 4f4d1c3 (vertex names):  
## [1] Y->X Y->Z Y->E X->Z X->E Z->E  
##   
## $tkp.id  
## [1] 17  
##   
## $igraph  
## IGRAPH 6699db9 DN-- 4 6 --   
## + attr: name (v/c), edge.arrow.mode (e/n)  
## + edges from 6699db9 (vertex names):  
## [1] A3->Y A3->Z A3->E Y ->Z Y ->E Z ->E  
##   
## $tkp.id  
## [1] 18  
##   
## $igraph  
## IGRAPH 1e6391a DN-- 4 6 --   
## + attr: name (v/c), edge.arrow.mode (e/n)  
## + edges from 1e6391a (vertex names):  
## [1] X->Z X->E X->D Z->E Z->D E->D  
##   
## $tkp.id  
## [1] 19  
##   
## $igraph  
## IGRAPH 356a575 DN-- 4 6 --   
## + attr: name (v/c), edge.arrow.mode (e/n)  
## + edges from 356a575 (vertex names):  
## [1] A2->X A2->Z A2->D X ->Z X ->D Z ->D  
##   
## $tkp.id  
## [1] 20  
##   
## $igraph  
## IGRAPH c22784c DN-- 4 6 --   
## + attr: name (v/c), edge.arrow.mode (e/n)  
## + edges from c22784c (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