École Polytechnique Fédérale de Lausanne

Assignment 2

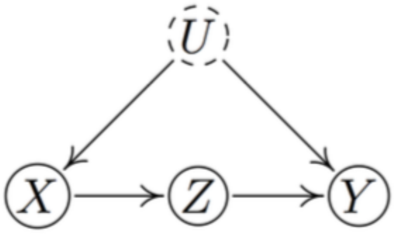
Network Analytics MGT - 416

Group 06

Gabriel Muret (250754)
Benoit Fontannaz (250809)
Diego Canton (258304)
Emery Sébastien (258565)
Sami Sellami (272658)
Romain Pichard (273060)

24th April 2020

Problem 1



We need to compute the observational equivalent of $P_{do(X=x)}(y,z)$. We have :

$$\begin{aligned}
P_{do(X=x)}(y,z) &= \sum_u P_{do(X=x)}(y,z,u) \\
&= \sum_u P_{do(X=x)}(y|z,u)P_{do(X=x)}(z|x)P_{do(X=x)}(u) \\
&= \sum_u P(y|z,u)P(z|x)P(u) \\
&= P(z|x)\sum_u P(y|z,u)P(u) \\
&= P(z|x)\sum_u \sum_x P(y|z,u,x)P(u|x,z)P(x) \\
&= \boxed{P(z|x)\sum_x P(y|x,z)P(x)}
\end{aligned}$$

Problem 2

The subroutine has been coded using Python. For the given matrix and nodes X and Y, the code has been used to determine all the backdoor sets :

```
adj = np.matrix([[0, 1, 0, 1, 1, 0],
          [0, 0, 1, 0, 1, 0],
          [0, 0, 0, 1, 0, 1],
          [0, 0, 0, 0, 1, 0],
          [0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 1, 0]])
X = 2
Y = 4


backdoor_sets = find_backdoor_sets(adj, X, Y)

print('The backdoor sets for P(Y |do(X)), according to the given DAG: \n\n',␣
 ↪backdoor_sets)
```

```
The backdoor sets for P(Y |do(X)), according to the given DAG:

 [1, (0, 1)]
```

Figure 1: Result using python subroutine

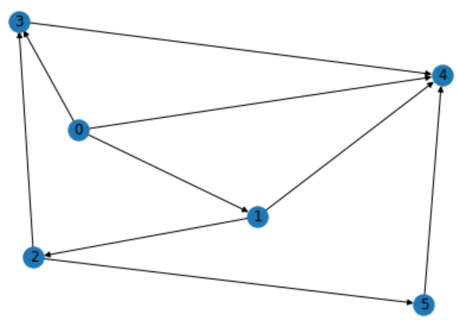
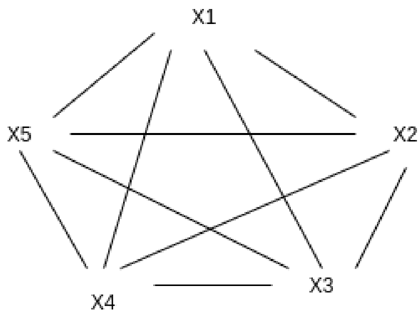


Figure 2: DAG Representation

The complete code is showed in the appendix.

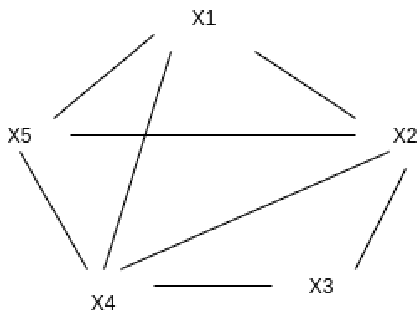
Problem 3

Complete graph (undirected)

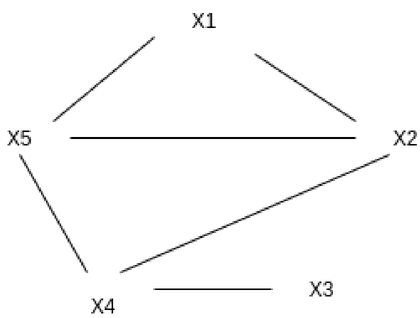


Edge elimination

a)zero order test
$X_3 \perp X_5$
$X_3 \perp X_1$



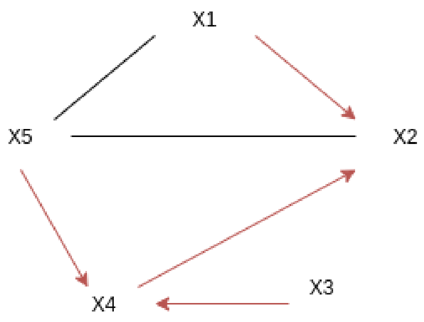
b) First order test
$X_1 \perp X_4 | X_5$
$X_2 \perp X_3 | X_4$



=> Right skeleton, no further edge elimination.

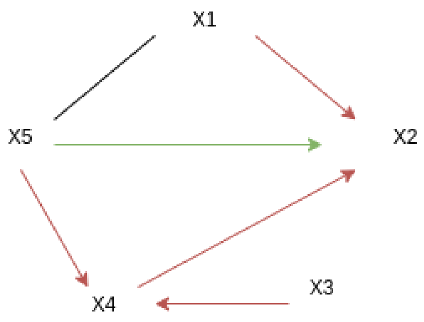
Orientation

a)
First, the statistical orientation must be studied



$X_5 \perp X_3 and X_5 \not\perp X_3|X_4$
$X_1 \perp X_4|X_5 and X_1 \not\perp X_4|X_2, X_5$

b)
Then, we'll study the logical orientation:



=> Can't recover the complete structure because X_1, X_5 direction can't be recovered.

Problem 4

I

-$[(X_3, X_5), P(X_5|do(X_3))]$

No descendant of $X_3 -> X_4$
Block all backdoor path from X_3 to X_5
$Z = \{X_6\} or \{X_2, X_6\} or \{X_1, X_6\} or \{X_1, X_2, X_6\}$

-$[(X_2, X_5), P(X_5|do(X_2))]$

No descendant of $X_2 -> X_3, X_4$
Block all backdoor path from X_3 to X_5
$Z = \{X_6\} or \{X_1, X_6\}$

II

- P($X_5|do(X_3), do(X_4))$

Conditioning and summing over X_6 :

$P(X_5|do(X_3), do(X_4)) = \sum_{X_6} P(X_5|do(X_3), do(X_4), X_6) \cdot P(X_6|do(X_3), do(X_4))$

Using Rule 2 : $X_5 \perp X_3|X_4, X_6 \; in \; G_{\underline{X_3}\overline{X_4}}$

$P(X_5|do(X_3), do(X_4)) = \sum_{X_6} P(X_5|X_3, do(X_4), X_6) \cdot P(X_6|do(X_3), do(X_4))$

Using Rule 3 : $X_6 \perp X_3|X_4 \; in \; G_{\overline{X_3 X_4}}$

$P(X_5|do(X_3), do(X_4)) = \sum_{X_6} P(X_5|X_3, do(X_4), X_6) \cdot P(X_6|do(X_4))$

Using Rule 3 : $X_6 \perp X_4 \; in \; G_{\overline{X_4}}$

$P(X_5|do(X_3), do(X_4)) = \sum_{X_6} P(X_5|X_3, do(X_4), X_6) \cdot P(X_6)$

Using Rule 3 : $X_5 \perp X_4|X_3, X_6 \; in \; G_{\overline{X_4}}$

$$\boxed{P(X_5|do(X_3), do(X_4)) = \sum_{X_6} P(X_5|X_3, X_6) \cdot P(X_6)}$$

-$P(X_5|do(X_2), do(X_3))$

Using Rule 3 : $X_5 \perp X_3|X_2 \; in \; G_{\overline{X_2 X_3}}$

$P(X_5|do(X_2), do(X_3)) = P(X_5|do(X_2))$

Conditioning and summing over X_6 :

$P(X_5|do(X_2), do(X_3)) = \sum_{X_6} P(X_5|do(X_2), X_6) \cdot P(X_6|do(X_2))$

Using Rule 3 : $X_6 \perp X_2 \; in \; G_{\overline{X_2}}$

$P(X_5|do(X_2), do(X_3)) = \sum_{X_6} P(X_5|do(X_2), X_6) \cdot P(X_6|X_2)$

Using Rule 2 : $X_5 \perp X_2|X_6 \ in \ G_{\underline{X_2}}$

$$\boxed{P(X_5|do(X_2), do(X_3)) = \sum_{X_6} P(X_5|X_6) \cdot P(X_6|X_2)}$$

Appendices

Hw2_ex2

April 23, 2020

[1]:
```python
import networkx as nx
import numpy as np
import itertools
from past.builtins import xrange
from itertools import chain
import copy
import matplotlib.pyplot as plt
```

[2]:
```python
class Node(object):
    """
    Node in a directed graph
    """
    def __init__(self, name=""):
        """
        Construct a new node, and initialize the list of parents and children.
        Each parent/child is represented by a (key, value) pair in dictionary,
        where key is the parent/child's name, and value is an Node object.
        Args:
            name: a unique string identifier.
        """
        self.name = name
        self.parents = dict()
        self.children = dict()

    def add_parent(self, parent):
        """
        Args:
            parent: an Node object.
        """
        if not isinstance(parent, Node):
            raise ValueError("Parent must be an instance of Node class.")
        pname = parent.name
        self.parents[pname] = parent

    def add_child(self, child):
        """
        Args:
```

```
            child: an Node object.
        """
        if not isinstance(child, Node):
            raise ValueError("Parent must be an instance of Node class.")
        cname = child.name
        self.children[cname] = child


class BN(object):
    """
    Bayesian Network
    """
    def __init__(self):
        """
        Initialize the list of nodes in the graph.
        Each node is represented by a (key, value) pair in dictionary,
        where key is the node's name, and value is an Node object
        """
        self.nodes = dict()

    def add_edge(self, edge):
        """
        Add a directed edge to the graph.

        Args:
            edge: a tuple (A, B) representing a directed edge A-->B,
                where A, B are two strings representing the nodes' names
        """
        (pname, cname) = edge

        ## construct a new node if it doesn't exist
        if pname not in self.nodes:
            self.nodes[pname] = Node(name=pname)
        if cname not in self.nodes:
            self.nodes[cname] = Node(name=cname)

        ## add edge
        parent = self.nodes.get(pname)
        child = self.nodes.get(cname)
        parent.add_child(child)
        child.add_parent(parent)

    def print_graph(self):
        """
        Visualize the current graph.
        """
        print("Bayes Network:")
```

```python
        for nname, node in self.nodes.iteritems():
            print("\tNode " + nname)
            print("\t\tParents: " + str(node.parents.keys()))
            print("\t\tChildren: " + str(node.children.keys()))

    def find_obs_anc(self, observed):
        """
        Traverse the graph, find all nodes that have observed descendants.
        Args:
            observed: a list of strings, names of the observed nodes.
        Returns:
            a list of strings for the nodes' names for all nodes
            with observed descendants.
        """
        visit_nodes = copy.copy(observed) ## nodes to visit
        obs_anc = set() ## observed nodes and their ancestors

        ## repeatedly visit the nodes' parents
        while len(visit_nodes) > 0:
            next_node = self.nodes[visit_nodes.pop()]
            ## add its' parents
            for parent in next_node.parents:
                obs_anc.add(parent)

        return obs_anc

    def is_dsep(self, start, end, observed):
        """
        Check whether start and end are d-separated given observed.
        This algorithm mainly follows the "Reachable" procedure in
        Koller and Friedman (2009),
        "Probabilistic Graphical Models: Principles and Techniques", page 75.
        Args:
            start: a string, name of the first query node
            end: a string, name of the second query node
            observed: a list of strings, names of the observed nodes.
        """

        ## all nodes having observed descendants.
        obs_anc = self.find_obs_anc(observed)

        ## Try all active paths starting from the node "start".
        ## If any of the paths reaches the node "end",
        ## then "start" and "end" are *not* d-separated.
        ## In order to deal with v-structures,
        ## we need to keep track of the direction of traversal:
        ## "up" if traveled from child to parent, and "down" otherwise.
```

```python
        via_nodes = [(start, "up")]
        visited = set() ## keep track of visited nodes to avoid cyclic paths

        while len(via_nodes) > 0:
            (nname, direction) = via_nodes.pop()
            node = self.nodes[nname]

            ## skip visited nodes
            if (nname, direction) not in visited:
                visited.add((nname, direction))

                ## if reaches the node "end", then it is not d-separated
                if nname not in observed and nname == end:
                    return False

                ## if traversing from children, then it won't be a v-structure
                ## the path is active as long as the current node is unobserved
                if direction == "up" and nname not in observed:
                    for parent in node.parents:
                        via_nodes.append((parent, "up"))
                    for child in node.children:
                        via_nodes.append((child, "down"))
                ## if traversing from parents, then need to check v-structure
                elif direction == "down":
                    ## path to children is always active
                    if nname not in observed:
                        for child in node.children:
                            via_nodes.append((child, "down"))
                    ## path to parent forms a v-structure
                    if nname in observed or nname in obs_anc:
                        for parent in node.parents:
                            via_nodes.append((parent, "up"))
        return True

def Z_subsets_disjoint_X_Y(myBN, X, Y):
    """
    Find all Z subsets that are disjoint with X and Y .
    Returns:
        a list of subsets
    """
    a = list(range(len(myBN.nodes)))
    Z_subsets = []
    Z_subsets.append(a)
    for i in xrange(2,len(a)+1):
        Z_subsets.append(list(itertools.combinations(a,i)))
    # flatten the list
    Z_subsets = [item for sublist in Z_subsets for item in sublist]
```

```
    # filter Z_subsets by keeping only disjoint Z subsets with X and Y
    Z_subsets_disj = [subset for subset in Z_subsets if set([X]).
↪isdisjoint(set([subset])) and set([Y]).isdisjoint(set([subset]))]

    return Z_subsets_disj

def find_backdoor_sets(adj, X, Y):
    """
    Find backdoor sets for P(Y |do(X)), given an adjacency matrix, X and Y
    Returns:
        backdoor sets
    """
    # Create a graph given adjacency matrix
    G =nx.from_numpy_matrix(adj)
    edges = list(G.edges)
    myBN = BN()
    for edge in edges:
        myBN.add_edge(edge)


    # draw all possible Z subsets, such that disjoint with X and Y
    Z = Z_subsets_disjoint_X_Y(myBN, X, Y)

    # add node I parent of X
    I = len(myBN.nodes)
    myBN.add_edge((I,X))

    # find backdoor sets
    backdoor_sets = []
    for subset in Z:

        XZ = list(chain(*(i if isinstance(i, tuple) else (i,) for i in␣
↪[X,subset])))
        # Check if Y d-sep I|X,Z
        if myBN.is_dsep(Y, I, XZ):
            # Check also if Z d-sep I
            if isinstance(subset, int):
                if myBN.is_dsep(subset, I, []):
                    backdoor_sets.append(subset)
            else :
                backdoor_bool = True
                for elem in subset:
                    if not myBN.is_dsep(elem, I, []):
                        backdoor_bool = False
                if backdoor_bool:
                    backdoor_sets.append(subset)
    return backdoor_sets
```

1 Execution:

[3]:
```
adj = np.matrix([[0, 1, 0, 1, 1, 0],
        [0, 0, 1, 0, 1, 0],
        [0, 0, 0, 1, 0, 1],
        [0, 0, 0, 0, 1, 0],
        [0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0]])
X = 2
Y = 4


backdoor_sets = find_backdoor_sets(adj, X, Y)

print('The backdoor sets for P(Y |do(X)), according to the given DAG: \n\n',␣
 ↪backdoor_sets)
```

```
The backdoor sets for P(Y |do(X)), according to the given DAG:

 [1, (0, 1)]
```

[ ]:
```
```