**Національний технічний університет України**

**«Київський політехнчний інститут імені Ігоря Сікорського»**

**Навчально-науковий інститут атомної та теплової енергетики**

**Кафедра цифрових технологій в енергетиці**

**Звіт**

**з лабораторної роботи №2**

**з дисципліни «Причинний ШІ»**

**Тема: «Методи ШІ відкриття причинних структур»**

Виконав:

студент групи ТР-23

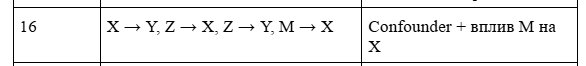
Ровний Г.О.

КИЇВ – 2025

**Мета роботи:**

1. Ознайомитися з методами відновлення причинно-наслідкових графів: **PC, FCI, GES**.
2. Навчитися застосовувати тести на умовну незалежність для різних типів даних і розподілів.
3. Навчитися будувати, орієнтувати та аналізувати причинні графи на Python.
4. Порівняти ефективність різних алгоритмів і тестів за допомогою метрик відповідності графа (SHD, Precision, Recall, F1).
5. Зрозуміти вплив типу даних, розподілів та вибору CI-тесту на точність відновленої причинної структури.

**Варіант роботи:**



Завдання:

1. Використати еталонний DAG з попередньої лабораторної роботи як «істину» причинно-наслідкових залежностей. Використати згенеровані раніше дані як основу. На його основі змоделювати нові набори даних з різними типами змінних та розподілами:

— Лінійні числові (наприклад, нормальний розподіл + адитивний шум).

— Нелінійні числові (наприклад, синус, експонента, квадратична функція + шум).

— Категоріальні (наприклад, бінарні чи багатокласові змінні, які залежать від батьків у DAG).

— Змішані набори (числові + категоріальні змінні).

1. Побудувати причинну структуру з даних.  Тобто, для кожного набору даних застосувати PC, FCI та GES -алгоритми, використовуючи різні тести на умовну незалежність та функцію оцінки (наприклад, BIC — Bayesian Information Criterion).
2. Порівняння знайдений граф з еталонним. Тобто, для кожного алгоритму та набору даних побудувати знайдений граф та порівняти його з еталонним DAG, обчисливши метрики відповідності:

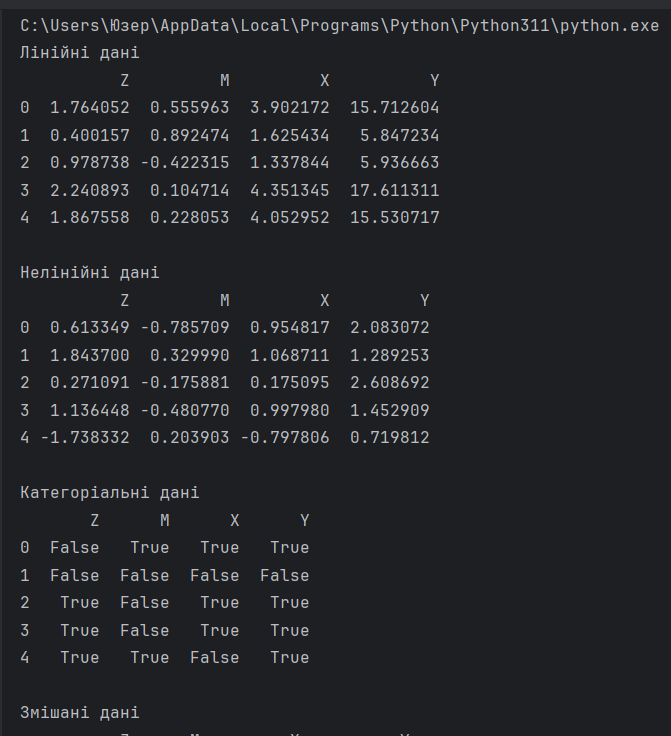
* SHD (Structural Hamming Distance)
* Precision, Recall, F1-score для ребер
* Кількість правильно орієнтованих колайдерів

**Результат виконання роботи**

1. **Генерація даних (перенос коду з ЛР1 як окнова + доповнення у вигляді генерації нових наборів даних з різними типами змінних та розподілами)**

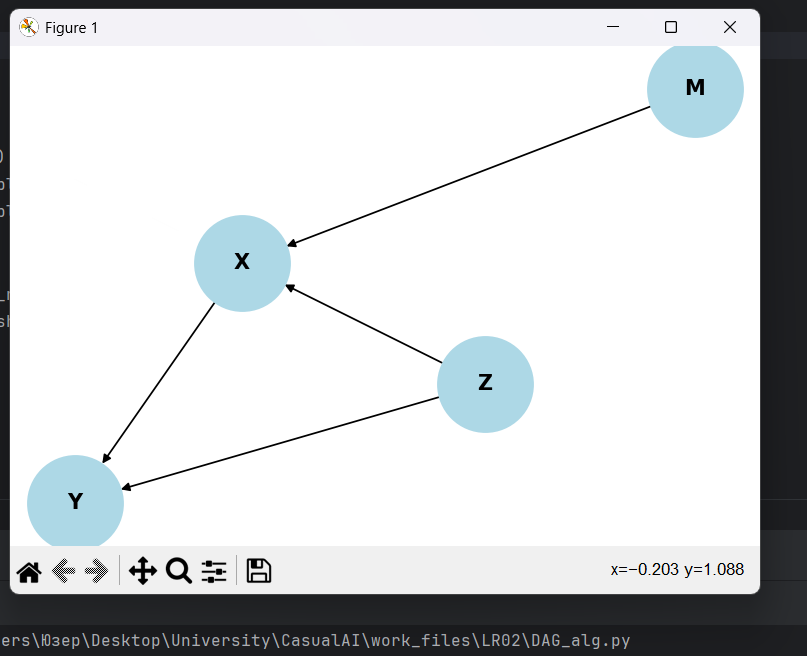
**generate.py:**

import numpy as np  
import pandas as pd  
  
# Генерація даних  
np.random.seed(0)  
n = 1000  
  
# Z — confounder  
Z\_num = np.random.normal(0, 1, n)  
  
# M впливає на X  
M\_num = np.random.normal(0, 1, n)  
  
# X залежить від Z і M  
X\_num = 2 \* Z\_num + 1.5 \* M\_num + np.random.normal(0, 0.3, n)  
  
# Y залежить від X і Z  
Y\_num = 3 \* X\_num + 2 \* Z\_num + np.random.normal(0, 0.3, n)  
  
# === Лінійні числові дані ===  
data\_lin = pd.DataFrame({  
 "Z": Z\_num,  
 "M": M\_num,  
 "X": X\_num,  
 "Y": Y\_num  
})  
  
# Нелінійні дані  
Z\_nonlin = np.random.normal(0, 1, n)  
M\_nonlin = np.random.uniform(-1, 1, n)  
X\_nonlin = np.sin(Z\_nonlin) + 0.5 \* M\_nonlin\*\*2 + np.random.normal(0, 0.2, n)  
Y\_nonlin = np.exp(0.3 \* X\_nonlin) + np.cos(Z\_nonlin) + np.random.normal(0, 0.2, n)  
  
data\_nonlin = pd.DataFrame({  
 "Z": Z\_nonlin,  
 "M": M\_nonlin,  
 "X": X\_nonlin,  
 "Y": Y\_nonlin  
})  
  
# Категоріальні дані  
boolean\_options = [True, False]  
Z\_cat = np.random.choice(boolean\_options, n)  
M\_cat = np.random.choice(boolean\_options, n)  
X\_cat = np.logical\_xor(Z\_cat, M\_cat) # X залежить від Z і M  
Y\_cat = np.logical\_or(X\_cat, Z\_cat) # Y залежить від X і Z  
  
data\_cat = pd.DataFrame({  
 "Z": Z\_cat,  
 "M": M\_cat,  
 "X": X\_cat,  
 "Y": Y\_cat  
})  
  
# Мішані дані  
Z\_mixed = np.random.normal(0, 1, n)  
M\_mixed = np.random.choice(boolean\_options, n)  
X\_mixed = 2 \* Z\_mixed + M\_mixed.astype(int) + np.random.normal(0, 0.5, n)  
Y\_mixed = 3 \* X\_mixed + 0.5 \* Z\_mixed + np.random.normal(0, 0.5, n)  
  
data\_mixed = pd.DataFrame({  
 "Z": Z\_mixed,  
 "M": M\_mixed,  
 "X": X\_mixed,  
 "Y": Y\_mixed  
})  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 print("Лінійні дані")  
 print(data\_lin.head(), "\n")  
  
 print("Нелінійні дані")  
 print(data\_nonlin.head(), "\n")  
  
 print("Категоріальні дані")  
 print(data\_cat.head(), "\n")  
  
 print("Змішані дані")  
 print(data\_mixed.head())

****

1. **Для нанаших наборів даних застосовуємо DAG-алгоритм (DAG\_alg.py):**

import networkx as nx  
import matplotlib.pyplot as plt  
import matplotlib  
matplotlib.use("TkAgg")  
  
# X → Y, Z → X, Z → Y, M → X  
G = nx.DiGraph()  
G.add\_edges\_from([  
 ("X", "Y"),  
 ("Z", "X"),  
 ("Z", "Y"),  
 ("M", "X")  
])  
  
  
def show\_dag(graph: nx.DiGraph):  
 plt.figure(figsize=(6, 4))  
 nx.draw(  
 graph,  
 with\_labels=True,  
 node\_color="lightblue",  
 node\_size=3000,  
 arrows=True,  
 font\_size=12,  
 font\_weight="bold"  
 )  
 plt.title("Еталонний причинно-наслідковий DAG (варіант 16)")  
 plt.show()  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 show\_dag(G)



Далі поріанюємо знайдений граф з еталонним через metrics.py:

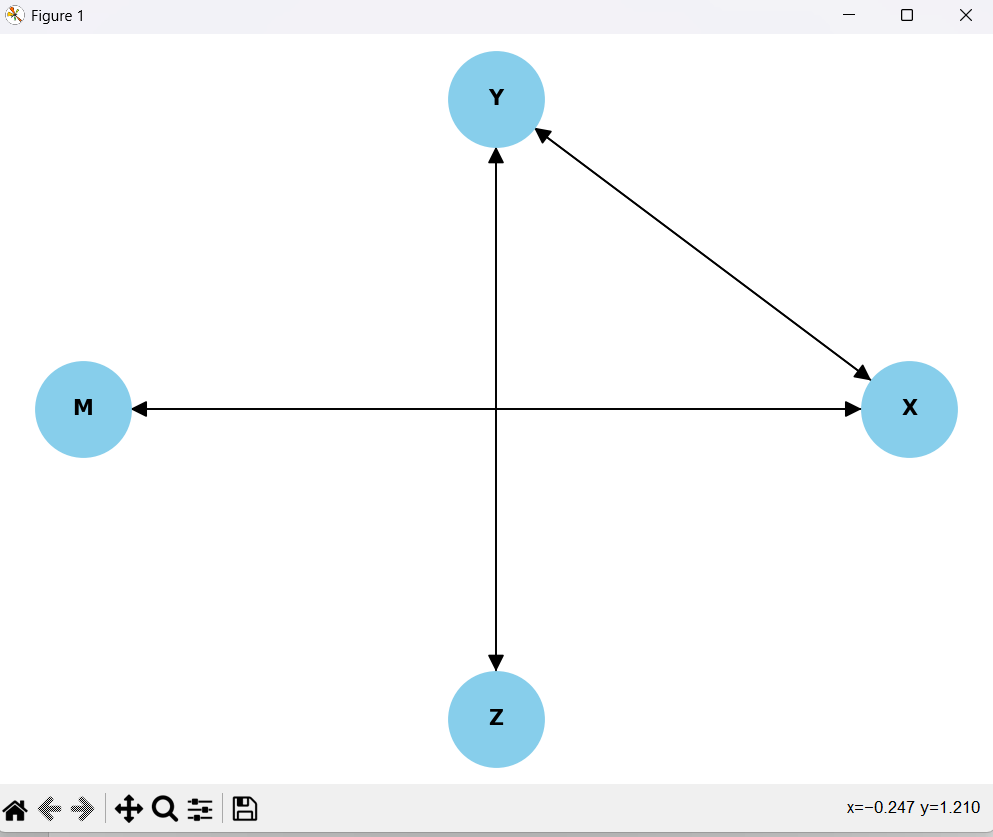
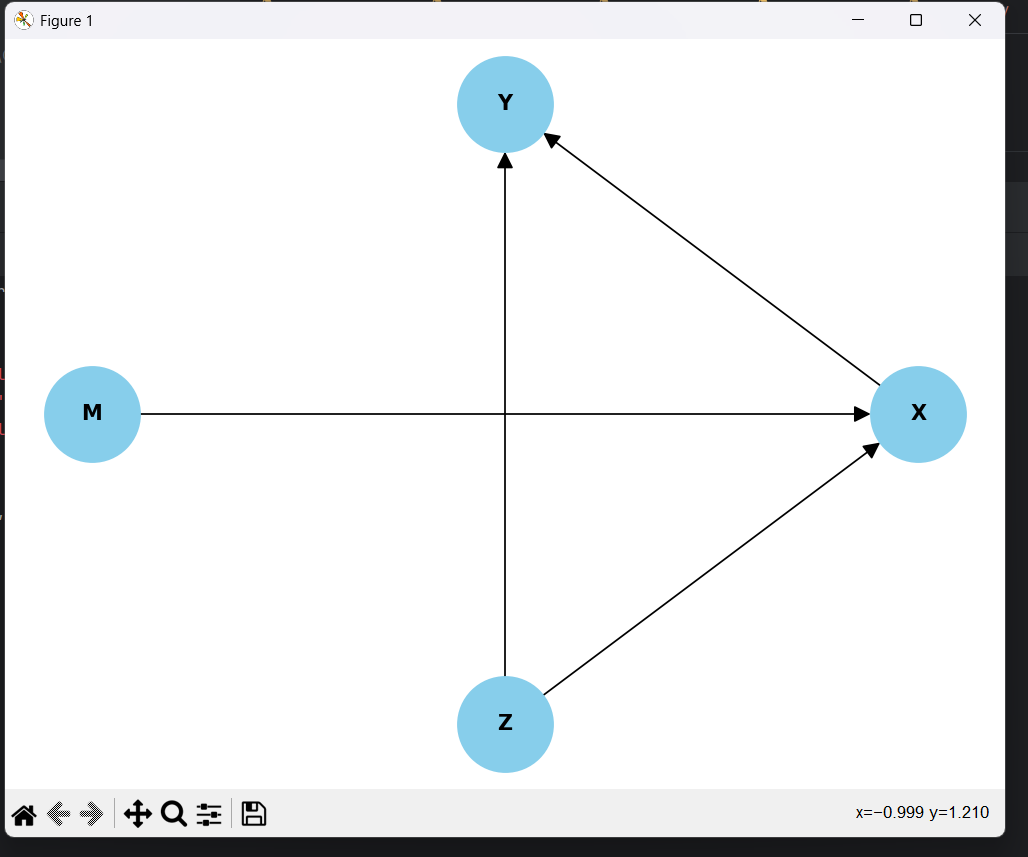
import networkx as nx  
from sklearn.metrics import precision\_score, recall\_score, f1\_score  
  
  
def compare\_graphs(true\_graph: nx.DiGraph, est\_graph: nx.DiGraph):  
 nodes = list(true\_graph.nodes())  
 all\_edges = [(u, v) for u in nodes for v in nodes if u != v]  
 y\_true = [1 if edge in true\_graph.edges() else 0 for edge in all\_edges]  
 y\_pred = [1 if edge in est\_graph.edges() else 0 for edge in all\_edges]  
  
 precision = precision\_score(y\_true, y\_pred, zero\_division=0)  
 recall = recall\_score(y\_true, y\_pred, zero\_division=0)  
 f1 = f1\_score(y\_true, y\_pred, zero\_division=0)  
 shd = sum(yt != yp for yt, yp in zip(y\_true, y\_pred))  
  
 # Колайдери (X -> Z <- Y, без ребра X-Y)  
 def find\_colliders(g):  
 colliders = set()  
 for z in g.nodes():  
 parents = list(g.predecessors(z))  
 for i in range(len(parents)):  
 for j in range(i + 1, len(parents)):  
 x, y = parents[i], parents[j]  
 if not g.has\_edge(x, y) and not g.has\_edge(y, x):  
 colliders.add((tuple(sorted([x, y])) + (z,)))  
 return colliders  
  
 true\_colliders = find\_colliders(true\_graph)  
 est\_colliders = find\_colliders(est\_graph)  
  
 correct\_colliders = len(true\_colliders & est\_colliders)  
  
return {  
 "Structural Hamming Distance": shd, # порівння з матрицею суміжності  
 "Precision": precision, # точність. частка істинно позитивних прогнозів від заг. к-сті  
 "Recall": recall, # Правильна ідентифікація позитивних випадків  
 "F1": f1, # оцінка ефективності моделі класифікації  
 "Correct Colliders": correct\_colliders # спільний наслідок змінних  
}

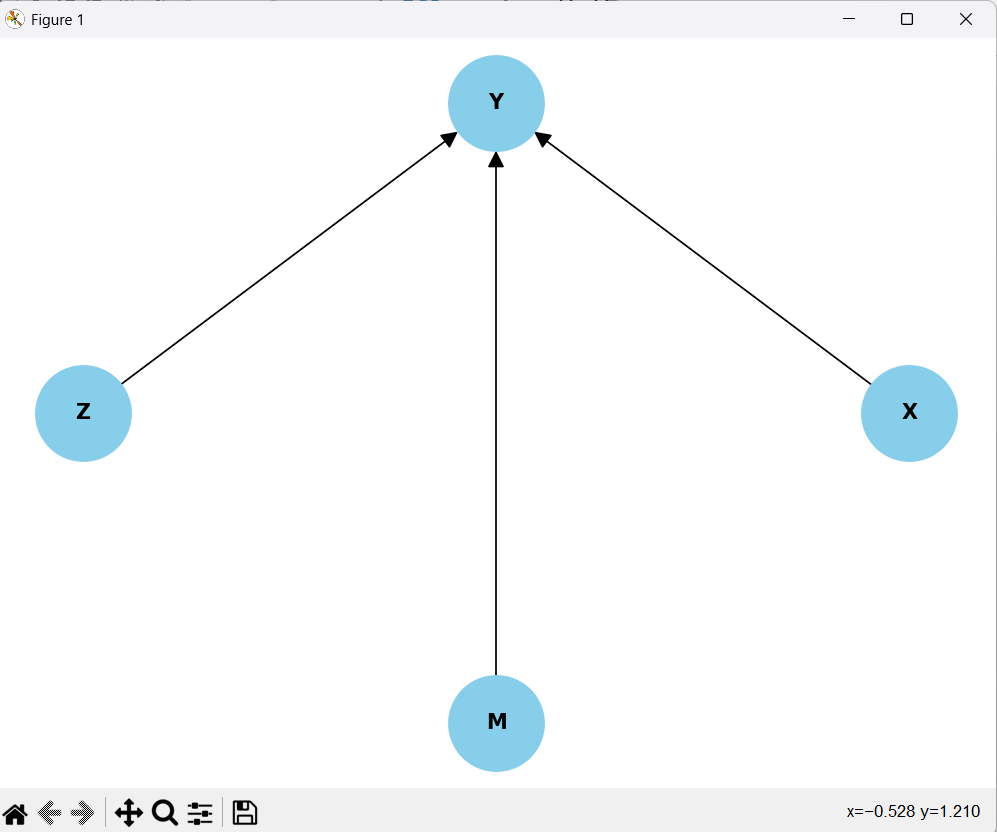
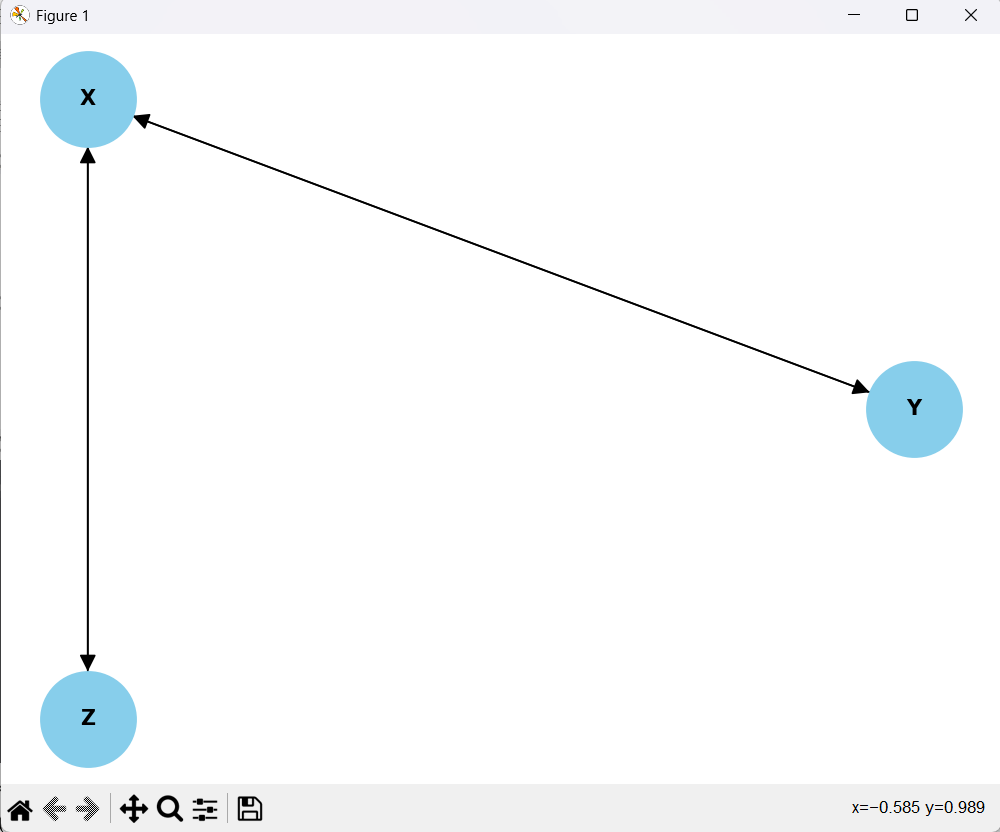
Цей код нам знадобиться далі для виконання наступних алгоритмів

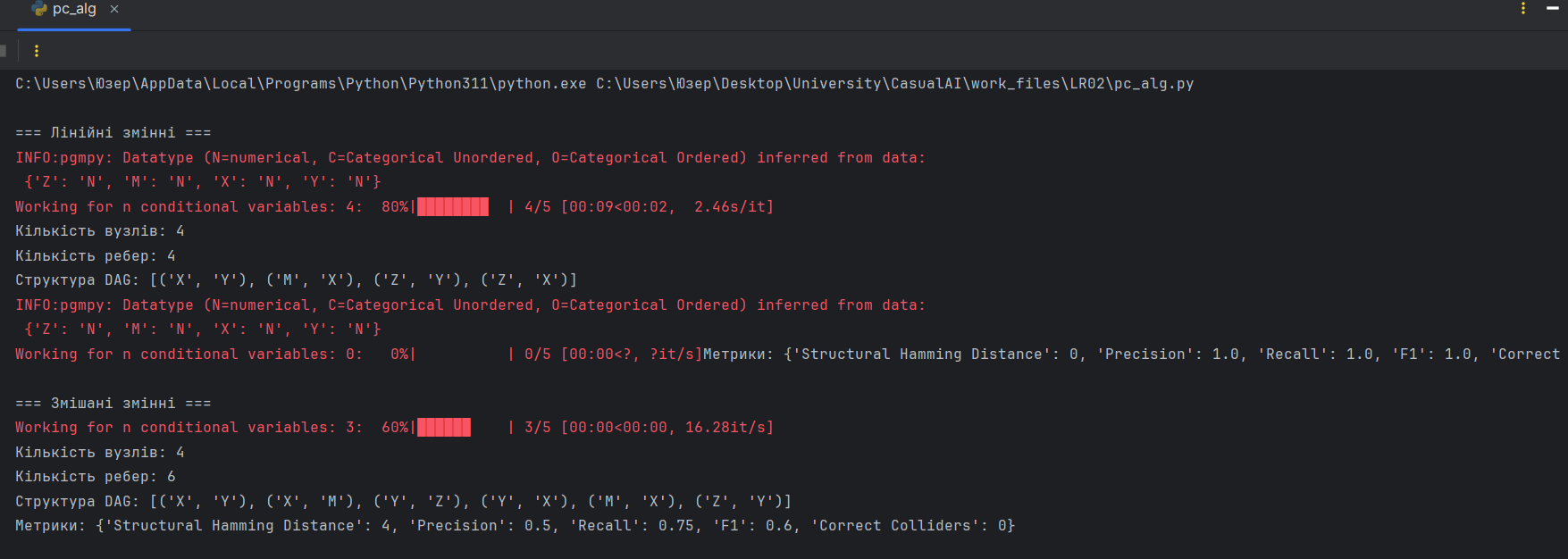
Застосуємо PC-алгоритм PC\_alg.py:

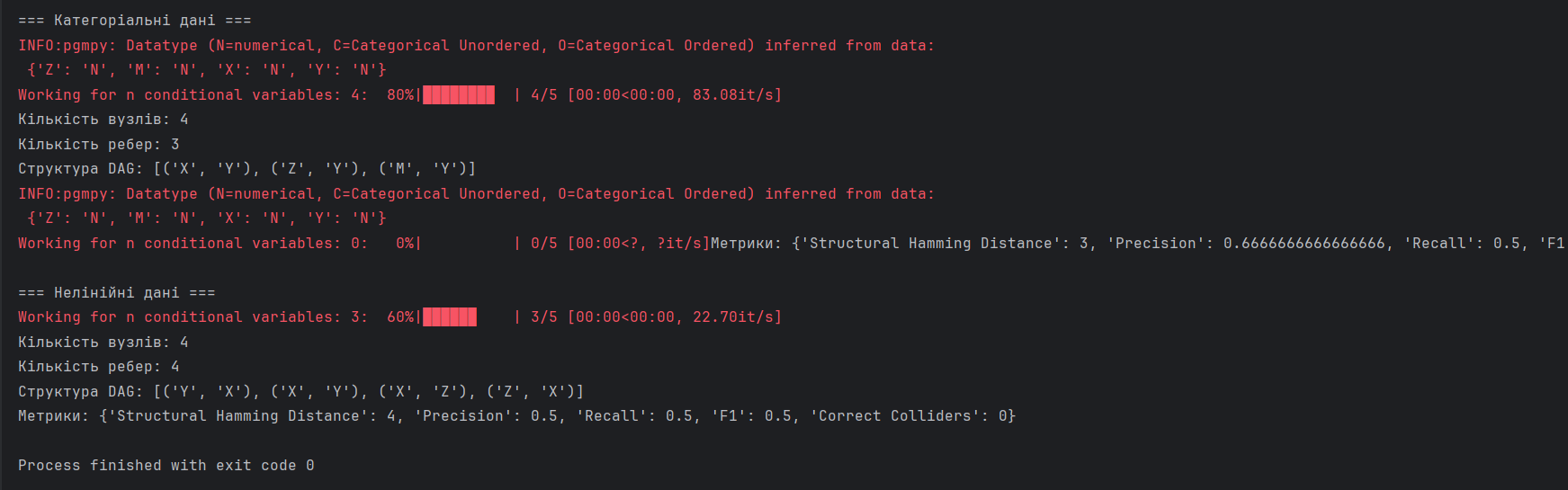
from pgmpy.estimators import PC  
import matplotlib.pyplot as plt  
import networkx as nx  
import matplotlib  
matplotlib.use("TkAgg")  
  
from generate import data\_lin, data\_mixed, data\_cat, data\_nonlin  
from DAG\_alg import G  
from metrics import compare\_graphs  
  
  
def PC\_visualize(df, name):  
 print(f"\n=== {name} ===")  
  
 # Перетворюємо всі значення у float  
 df = df.astype(float)  
  
 # Ініціалізація PC-алгоритму  
 pc = PC(df)  
 model = pc.estimate(ci\_test="pearsonr", alpha=0.05, n\_jobs=-1)  
  
 # Інформація про побудований граф  
 print("Кількість вузлів:", len(model.nodes()))  
 print("Кількість ребер:", len(model.edges()))  
 print("Структура DAG:", list(model.edges()))  
  
 if len(model.nodes()) == 0:  
 print("⚠️ Порожній граф — перевір правильність даних.")  
 return  
  
 # Створюємо networkx-граф для візуалізації  
 nx\_graph = nx.DiGraph()  
 nx\_graph.add\_edges\_from(model.edges())  
  
 # Візуалізація графа  
 plt.figure(figsize=(8, 6))  
 pos = nx.circular\_layout(nx\_graph)  
 nx.draw(  
 nx\_graph,  
 pos,  
 with\_labels=True,  
 node\_size=3000,  
 node\_color="skyblue",  
 font\_size=12,  
 font\_weight="bold",  
 arrows=True,  
 arrowsize=20  
 )  
 plt.title(f"PC-Algorithm Result — {name}")  
 plt.show()  
  
 # Порівняння з еталонним DAG  
 results = compare\_graphs(G, nx\_graph)  
 print("Метрики:", results)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 PC\_visualize(data\_lin, "Лінійні змінні")  
 PC\_visualize(data\_mixed, "Змішані змінні")

PC\_visualize(data\_cat, "Категоріальні дані")  
 PC\_visualize(data\_nonlin, "Нелінійні дані")





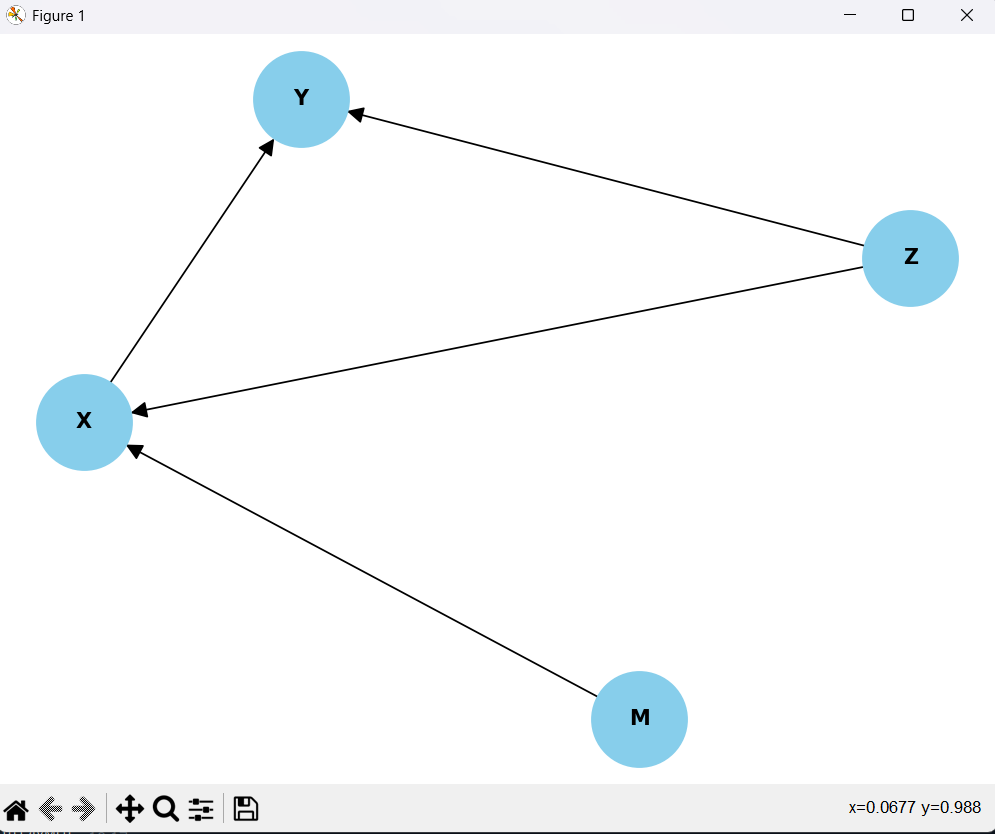
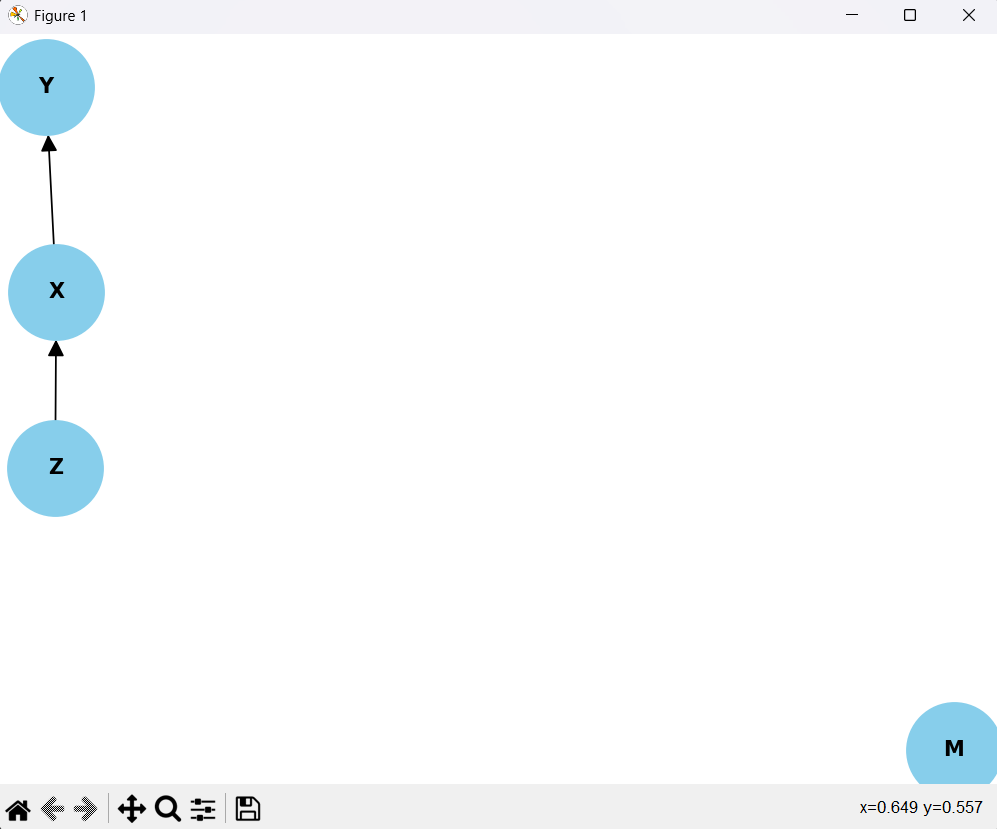


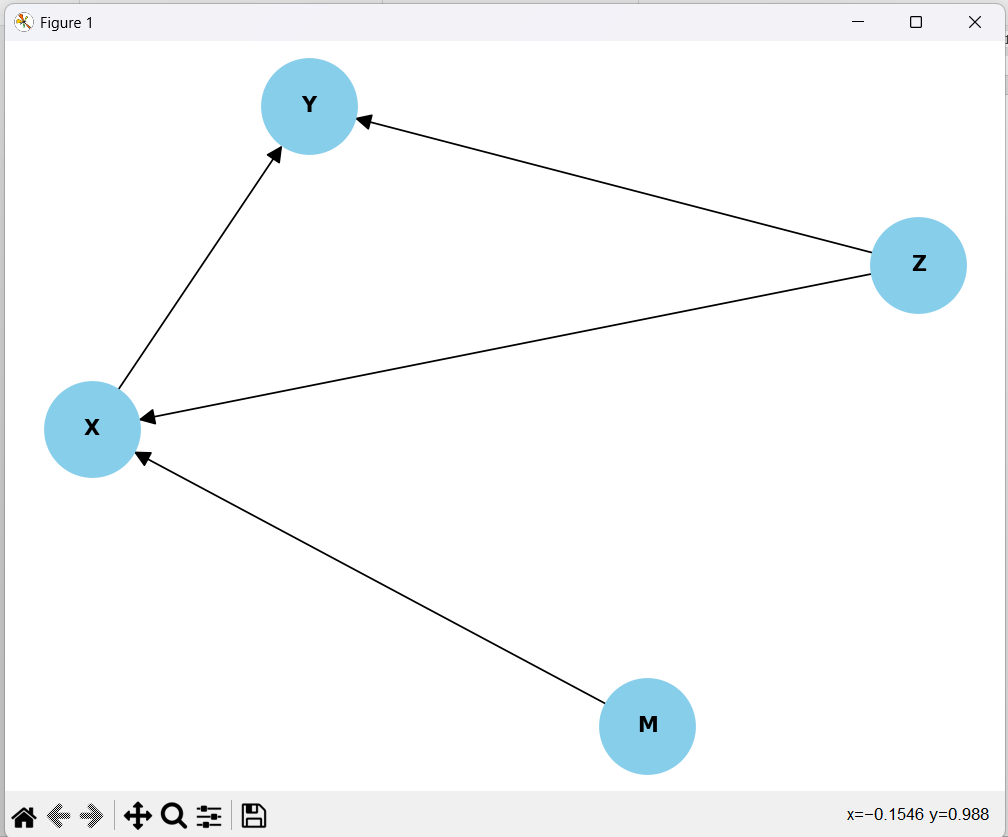
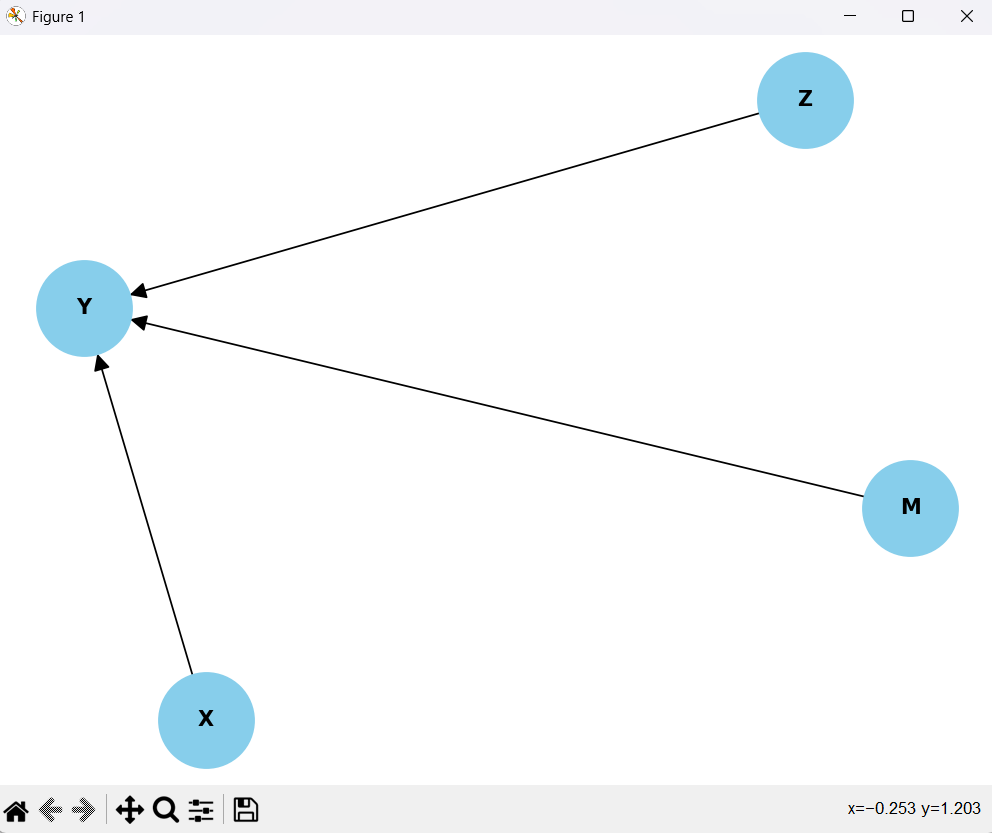
\* PC-алгоритм — це метод для реконструкції орієнтованого ациклічного графа (DAG), що відображає причинні зв'язки. Він працює на основі серії тестів умовної незалежності, послідовно виявляючи та усуваючи зв'язки між змінними, які стають незалежними за умови врахування інших змінних у наборі даних.

\* Баєсівський інформаційний критерій (BIC) — це статистичний критерій для вибору моделі серед скінченного набору моделей. Він враховує як якість підгонки моделі до даних, так і її складність (кількість параметрів). Перевага надається моделі з найнижчим значенням BIC, що дозволяє уникнути перенавчання та обрати найбільш ощадливу модель.

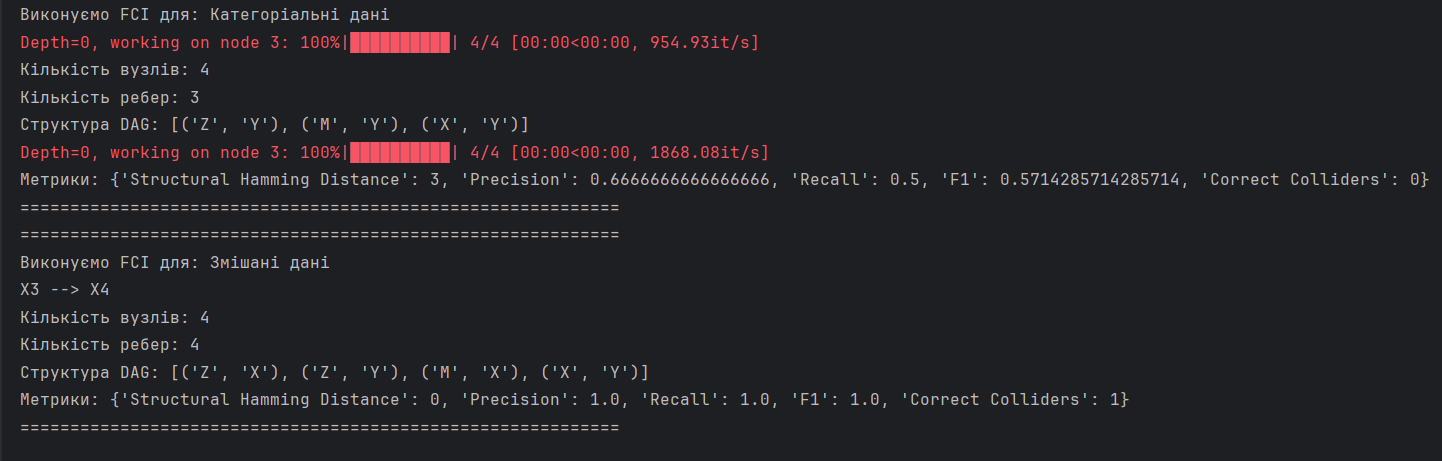
FCI\_alg.py:

import networkx as nx  
from causallearn.search.ConstraintBased.FCI import fci  
import matplotlib.pyplot as plt  
import matplotlib  
import pandas as pd  
import numpy as np  
  
from DAG\_alg import G  
from generate import data\_lin, data\_nonlin, data\_cat, data\_mixed  
from metrics import compare\_graphs  
  
matplotlib.use("TkAgg")  
  
  
def FCI\_visualize(g, var\_names, title):  
 nx\_g = nx.DiGraph()  
 node\_map = {node: var\_names[i] for i, node in enumerate(g.get\_nodes())}  
 nx\_g.add\_nodes\_from(node\_map.values())  
  
 for edge in g.get\_graph\_edges():  
 node1, node2 = node\_map[edge.node1], node\_map[edge.node2]  
 nx\_g.add\_edge(node1, node2)  
  
 plt.figure(figsize=(8, 6))  
 pos = nx.spring\_layout(nx\_g, seed=42)  
 nx.draw(  
 nx\_g,  
 pos,  
 with\_labels=True,  
 node\_size=3000,  
 node\_color="skyblue",  
 font\_size=12,  
 font\_weight="bold",  
 arrows=True,  
 arrowsize=20  
 )  
 plt.title(title)  
 plt.show()  
  
 print("Кількість вузлів:", len(nx\_g.nodes()))  
 print("Кількість ребер:", len(nx\_g.edges()))  
 print("Структура DAG:", list(nx\_g.edges()))  
  
 results = compare\_graphs(G, nx\_g)  
 print("Метрики:", results)  
 print("=" \* 60)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 datasets = [  
 (data\_lin, "Лінійні дані"),  
 (data\_nonlin, "Нелінійні дані"),  
 (data\_cat, "Категоріальні дані"),  
 (data\_mixed, "Змішані дані"),  
 ]  
  
 for dataset, title in datasets:  
 print("=" \* 60)  
 print(f"\nВиконуємо FCI для: {title}")  
  
 df = dataset.copy()  
 df = df.apply(pd.to\_numeric, errors='coerce')  
 df = df.dropna()  
  
 data\_np = df.to\_numpy(dtype=float)  
 fci\_result, \_ = fci(data\_np, independence\_test\_method="fisherz")  
  
 FCI\_visualize(fci\_result, list(df.columns), title)

Отримуємо наступний результат:  
 







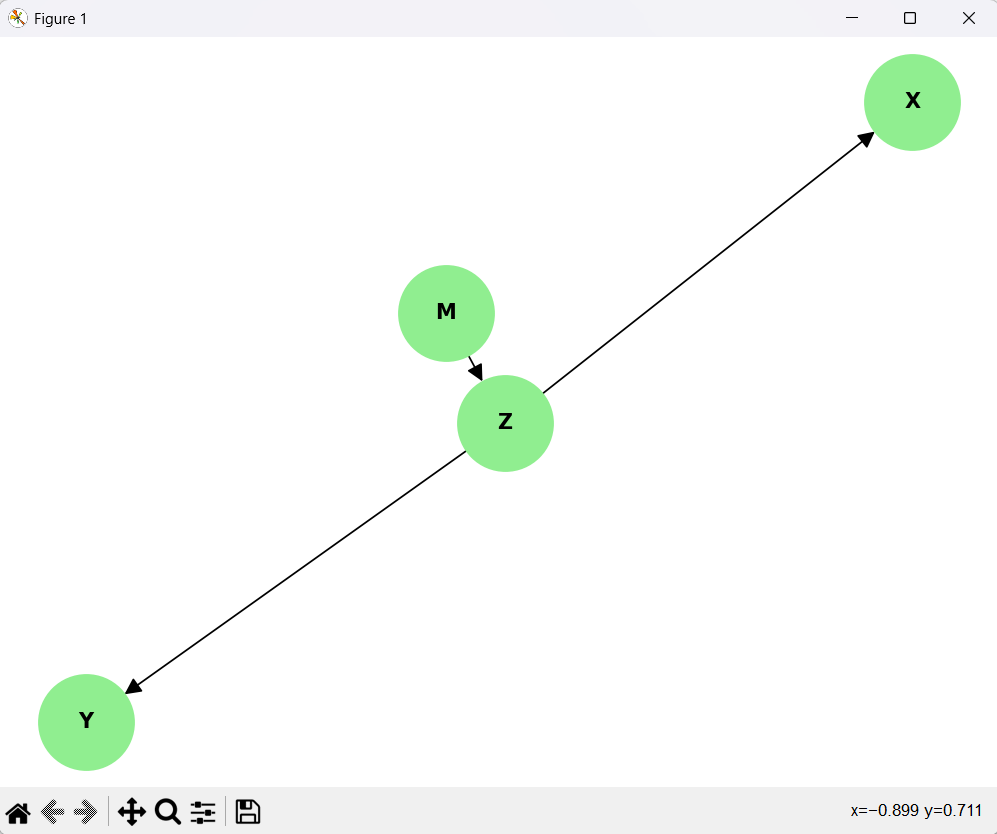
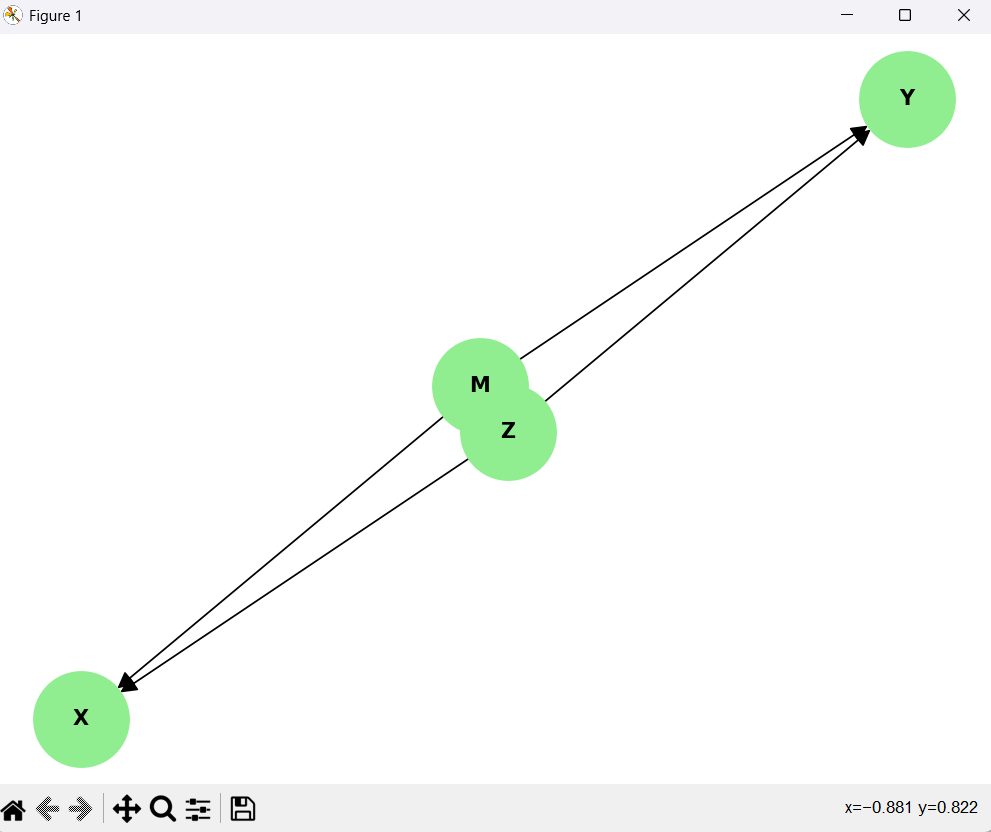
FCI-алгоритм — це вдосконалений метод виявлення причинних зв'язків, який, на відміну від PC-алгоритму, залишається надійним навіть за наявності неспостережуваних (латентних) спільних причин та систематичної помилки вибірки (selection bias).

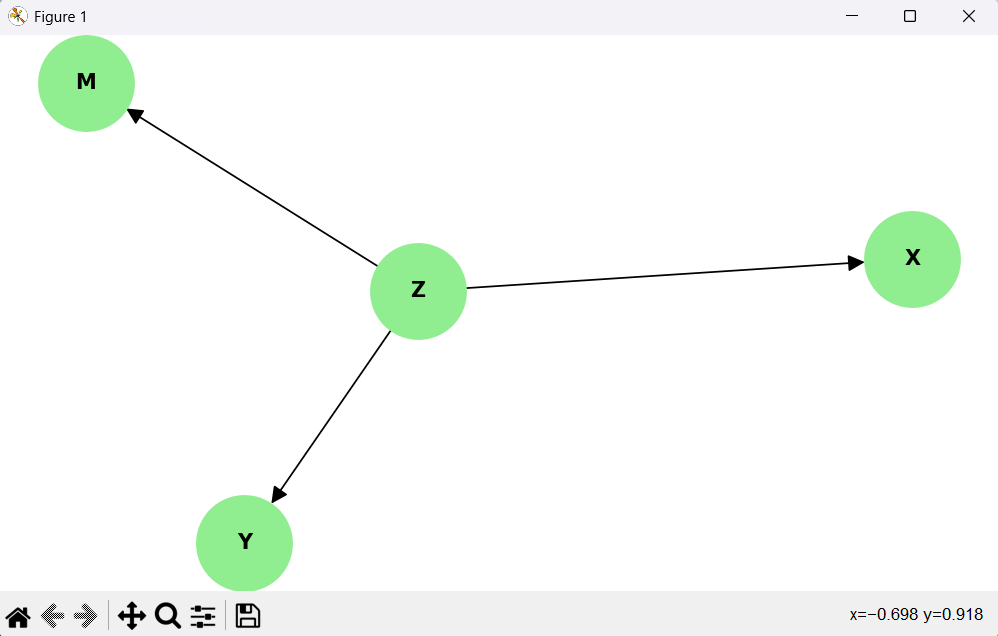
Точний тест Фішера — це непараметричний тест для аналізу таблиць спряженості, який обчислює точну ймовірність спостережуваного зв'язку між двома категоріальними змінними за умови їх незалежності. Він є ідеальним для малих вибірок, де апроксимації (як хі-квадрат) можуть бути неточними.

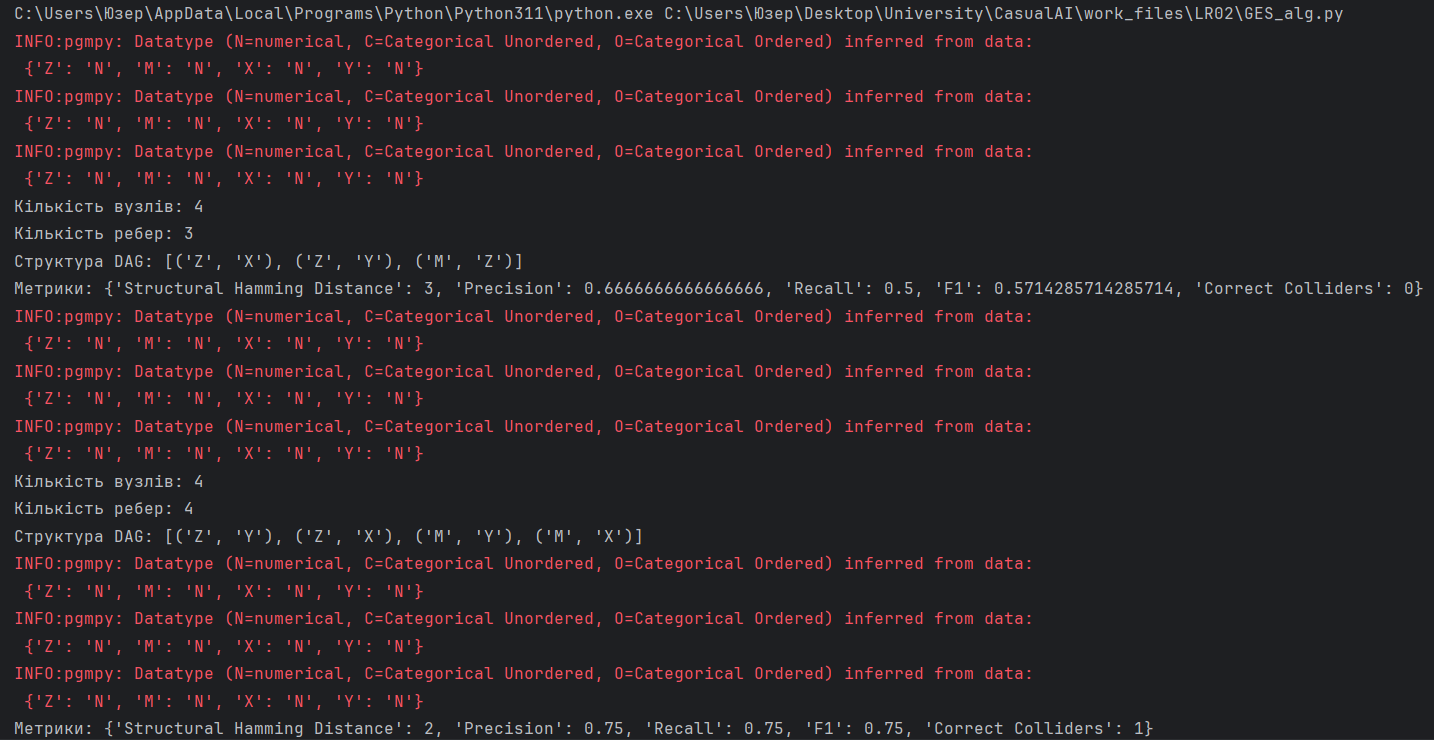
Критерій хі-квадрат — це статистичний тест, що дозволяє оцінити, чи є спостережувана різниця між частотами в категоріях двох змінних статистично значущою, чи її можна пояснити випадковістю. Він перевіряє гіпотезу про незалежність цих змінних

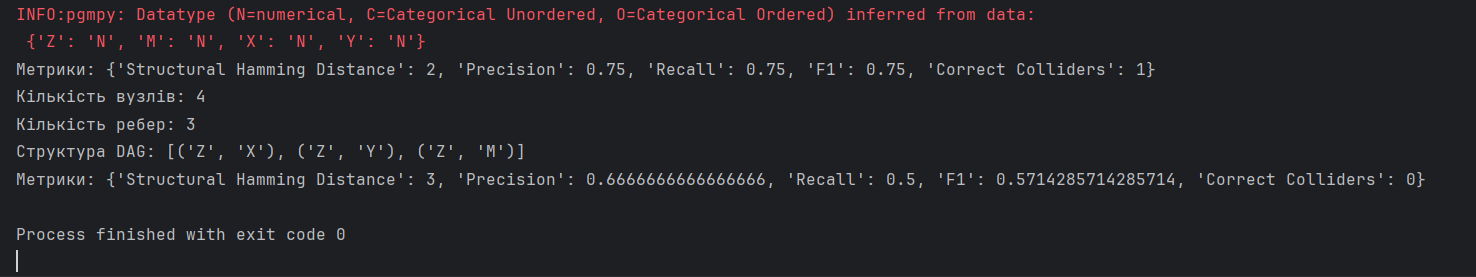
GES\_alg.py:

import networkx as nx  
import matplotlib.pyplot as plt  
import matplotlib  
from pgmpy.estimators import GES  
  
from DAG\_alg import G  
from generate import data\_lin, data\_cat, data\_mixed  
from metrics import compare\_graphs  
  
matplotlib.use("TkAgg")  
  
  
def GES\_visualize(data, data\_name):  
 ges = GES(data)  
 model = ges.estimate(scoring\_method="bds")  
  
 print("Кількість вузлів:", len(model.nodes()))  
 print("Кількість ребер:", len(model.edges()))  
 print("Структура DAG:", model.edges())  
  
 nx\_graph = nx.DiGraph()  
 nx\_graph.add\_edges\_from(model.edges())  
  
 plt.figure(figsize=(8, 6))  
 pos = nx.spring\_layout(nx\_graph, seed=42)  
 nx.draw(  
 nx\_graph, pos,  
 with\_labels=True,  
 node\_size=3000,  
 node\_color="lightgreen",  
 font\_size=12,  
 font\_weight="bold",  
 arrows=True,  
 arrowsize=20  
 )  
 plt.title(f"GES результат — {data\_name}")  
 plt.show()  
  
 # Порівняння з істинним DAG  
 results = compare\_graphs(G, nx\_graph)  
 print("Метрики:", results)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 datasets = {  
 "Лінійні дані": data\_lin,  
 "Категоріальні дані": data\_cat,  
 "Змішані дані": data\_mixed  
 }  
 for name, df in datasets.items():  
 GES\_visualize(df, name)







GES-алгоритм (Greedy Equivalence Search): Це оцінковий (score-based) метод пошуку причинної структури. На відміну від обмежувальних (constraint-based) підходів (як PC/FCI), він не покладається на тести умовної незалежності. Замість цього, він виконує жадібний пошук у просторі всіх можливих графів, ітеративно вносячи зміни (додавання, видалення, реверс ребер) для максимізації певної функції оцінки (наприклад, BIC), доки не буде знайдено локальний оптимум.

Тест Брока-Дечерта-Шейнкмана (BDS): Це непараметричний статистичний тест, призначений для перевірки нульової гіпотези про те, що часовий ряд є незалежним та однаково розподіленим (i.i.d.). Його часто застосовують до залишків моделі (наприклад, ARIMA) для виявлення залишків нелінійної залежності. Позитивний результат тесту вказує на те, що модель неправильно специфікована і не враховує всю структуру даних, що може призвести до хибних висновків, зокрема при ідентифікації критичних переходів у системі.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Набір даних** | **Тип змінних** | **Алгоритм** | **Тест** | **SHD** | **Precision** | **Recall** | **F1** | **Коментар** |
| Dataset 1 | лінійні | РС | BIC | 0 | 1.0 | 1.0 | 1.0 | Гарно показав себе |
| FCI | Fisher’s exact test | 0 | 1.0 | 1.0 | 1.0 | Відновив граф повністю |
| GES | BDS | 3 | 0.6 | 0.5 | 0.57 | Частково відновив граф, пропускає деякі ребра |
| Dataset 2 | Нелінійні | РС | BIC | 4 | 0.5 | 0.5 | 0.5 | Працює посередньо, деякі пропуски є. Низька точність |
| FCI | Fisher’s exact test | 2 | 1.00 | 0.5 | 0.66 | Не додає зайвих ребер але пропускає істині звязки |
| GES | BDS | 3 | 0.67 | 0.5 | 0.57 | відновлює частину структури, додає кілька хибних ребер |
| ­Dataset 3 | Категоріальні | РС | BIC | 3 | 0.66 | 0.5 | 0.57 | Середній результат |
| FCI | Chi-Square test | 3 | 0.6 | 0.5 | 0.57 | відновлює половину істинних зв’язків і допускає хибні |
| GES | BDS | 2 | 0.75 | 0.75 | 0.75 | Найкращий результат |
| Dataset 4 | Змішані | РС | BIC | 4 | 0.5 | 0.75 | 0.6 | знаходить більшість істинних зв’язків, але додає багато хибних ребер |
| FCI | Chi-Square test | 0 | 1.0 | 1.0 | .0 | Хороший результат |
| GES | BDS­ | 3 | 0.66 | 0.5 | 0.57 | відновлює частину структури, додає деякі хибні ребра |

**Висновок:**

У результаті виконання лабораторної роботи було створено методи ШІ для відкриття причинних структур: PC, FCI та GES. Було оцінено їхню ефективність за метриками: SHD (Structural Hamming Distance), Precision, Recall та F1-score. У результаті ми побачили, **PC та FCI** добре працюють на лінійних даних, **FCI** — на змішаних і нелінійних, **GES** — на категоріальних. Можна зробити висновок що вибір алгоритму залежить від типу даних.