Stage d’été – Semaine 9

QGNN-TimeCausality
Avancement du projet

Université de Sherbrooke

UNIVERSITÉ TÉLUQ

01

La Rédaction

Resume des dernieres semaines :

J’ai commencé a écrire

le papier : [ici]

Pour la structure, j’ai pris

l’inspiration de ce document

Causal Discovery in Time Series
using Quantum Graph Neural Networks (QGNN)

Sahar Saoudi
Université de Sherbrooke
Sherbrooke, QC, Canada
sahar.saoudi@usherbrooke.ca

Belkacem Chikhaoui
Université TÉLUQ
Montreal, QC, Canada
belkacem.chikhaoui@teluq.ca

Miloud Mihoubi
Université TÉLUQ
Montreal, QC, Canada
miloud.mihoubi@teluq.ca

Abstract

Causal discovery in multivariate time series is a fundamental task in many scientific fields, including finance, neuroscience, and climate science. In this work, we investigate both classical and quantum approaches for inferring causal structures from temporal data. We implement and compare traditional methods such as Granger causality and PCMCI, as well as machine learning models based on Graph Neural Networks (GNNs) and Quantum Graph Neural Networks (QGNNs). The models are evaluated on simulated financial time series with known causal ground truths, allowing for a controlled analysis of their strengths and limitations. This study highlights the potential of quantum-enhanced architectures for capturing complex temporal dependencies and contributes to the growing exploration of quantum machine learning for causal inference.

1 Introduction

1.1 General Context

When working with time series data, it’s often useful to know which variables are affecting others. This is what causal discovery is about. It helps in making predictions, understanding patterns, and making better decisions. In fields like finance or climate science, being able to figure out these causal links can be really important because everything is connected and constantly changing.

1.2 Motivation and Objective

There are already some methods that people use to find causal relationships in data, like Granger causality and VAR models. But they don’t always work well when the data gets too complicated—like when the relationships aren’t linear or there’s too much noise. That’s why researchers started trying newer methods like Graph Neural Networks (GNNs). They look at the data as

graphs and learn how variables are connected. For this project, we were also interested in trying something more recent: quantum machine learning. The idea is that quantum models might be able to find patterns that classical models can’t. So in this project, we tried both classical and quantum approaches to see how they compare for causal discovery.

1.3 Dataset description

In this project, we use a simulated financial time series dataset introduced in the book Causality, Probability, and Time by S. Kleinberg. The dataset consists of 25 financial variables and includes ten different causal structures, each applied to two distinct 4,000-day time periods. These structures represent various configurations, such as random causal links with fixed or variable time lags and many-to-one relationships. Each dataset also comes with a ground truth causal graph, allowing us to evaluate each method’s ability to recover the correct structure. This controlled setup makes it ideal for benchmarking both classical and quantum approaches to causal discovery.

2 Methodology

2.1 Classical approach - Granger

To start with a baseline for causal discovery, we implemented the Granger causality test, which is one of the most commonly used statistical methods for time series analysis. The idea behind Granger causality is that if knowing the past values of one variable X helps to predict the future values of another variable Y, then X is said to *Granger-cause* Y. It is important to note that this does not imply true causality in the philosophical sense, but rather a predictive relationship based on lagged dependencies.

In practice, we applied the test to all possible pairs of time series in the dataset. For each pair (X_i, X_j), the test checks whether including the past values of X_j significantly improves the prediction of X_i, compared to using only the past of X_i. We used a maximum lag of 3, and from the test results, we extracted the smallest p-value across all lags for each pair.

Since testing multiple variable pairs increases the risk of false positives, we applied a multiple testing correction using the Benjamini-Hochberg False Discovery Rate (FDR) method. This helps control the expected proportion of incorrect causal links among those identified as significant. After the correction, we constructed a predicted adjacency matrix indicating which variable pairs were considered causally linked according to the Granger test. This matrix was then used for comparison with the known ground truth.

2.2 Classical approach - PCMCI

For more flexible and time-aware causal discovery, we used the PCMCI algorithm from the Tigramite package. PCMCI (Peter and Clark Momentary Conditional Independence) is a method designed for detecting causal links in multivariate time series under the assumptions of causal sufficiency and stationarity. It performs conditional independence tests at multiple time lags to uncover directed causal links in the past, and optionally contemporaneous ones. The method requires choosing a statistical test for checking conditional independence between variables. We tested PCMCI with two different tests: **ParCorr**, which is based on linear correlation, and **GPDC**, which can capture more general, nonlinear dependencies.

Resume des dernieres semaines :

La structure que j'ai en tête :

(a) PCMCI with ParCorr The first version we tested used the ParCorr test, which is short for partial correlation. This test assumes linear dependencies and Gaussian noise, and checks whether two variables are conditionally independent given the rest. We applied PCMCI-ParCorr to the financial time series dataset, setting a maximum time lag of 3. For each pair of variables and each lag, the algorithm outputs a p-value indicating whether there is evidence of a causal link from one variable to another at that lag.

After running the algorithm, we applied a threshold on the corrected p-values (using an alpha level of 0.05) to build a binary adjacency matrix representing the predicted causal structure. A directed edge from variable X_j to X_i was added if the p-value for X_j at lag τ influencing X_i at time t was below the significance threshold.

(b) PCMCI with GPDC To evaluate how PCMCI performs under more general (nonlinear) dependencies, we also used the GPDC (Gaussian Process Dependency Criterion) as the conditional independence test. GPDC is based on kernel methods and allows detecting additive nonlinear relationships, which ParCorr might miss. Just like before, we ran PCMCI with a maximum lag of 3 and thresholded the corrected p-values to obtain the predicted causal graph.

Using both ParCorr and GPDC allowed me to compare the impact of the independence test on the performance of PCMCI, and to see how sensitive the method is to linear versus nonlinear relationships in the time series data.

02

L’implémentation - version classique (granger)

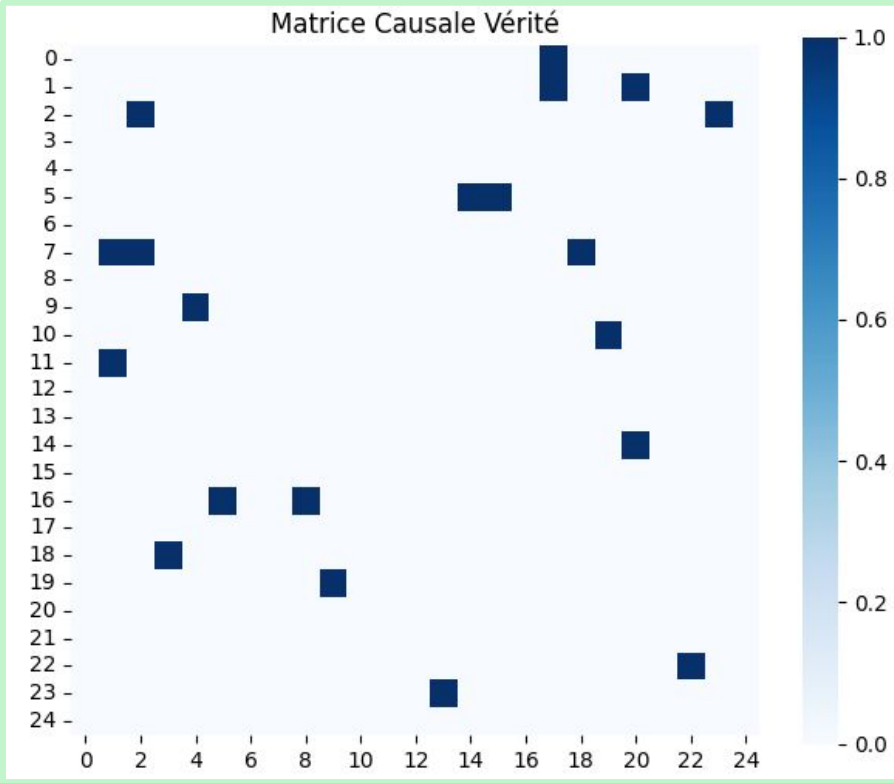
Application du test de Granger

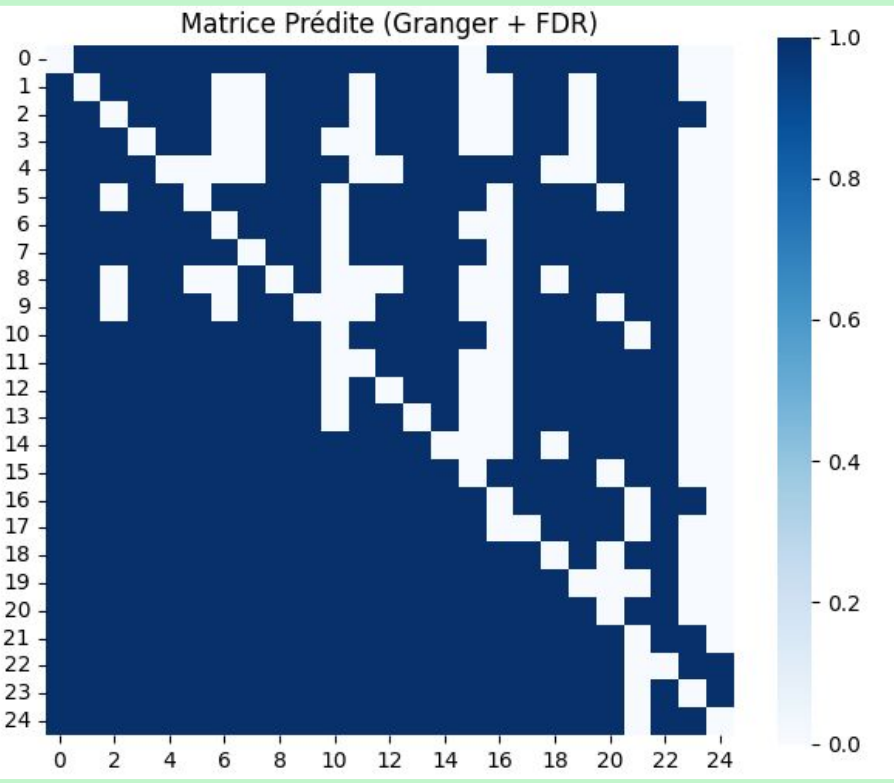
- On applique le **test de Granger** à chaque paire de variables/noeud, sauf sur la diagonale.
- Lags testés jusqu'à 3 pas de temps pour détecter des relations avec retard
- Pour chaque paire de variables, on sélectionne la **p-value minimale** parmi les lags testés
- Cette p-value est utilisée dans la **correction FDR** pour déterminer si un lien est significatif.
- Correction des tests multiples avec la méthode FDR (On considère un lien significatif seulement si la p-value corrigée est inférieure à **α = 0.05**)
- Construction d'une matrice d'adjacence causale binaire à partir des liens significatifs et comparaison avec la vérité terrain

```python
# === Collecte des p-values pour chaque paire (i, j) ===
all_p_values = []
pairs = []

print("Testing Granger causality with FDR correction...")
for i in range(num_vars):
    for j in range(num_vars):
        if i == j:
            continue
        try:
            data_pair = np.column_stack([df.iloc[:, j].values, df.iloc[:, i].values])  #
            result = grangercausalitytests(data_pair, maxlag=max_lag, verbose=False)
            min_p = min(result[lag][0]['ssr_ftest'][1] for lag in range(1, max_lag + 1))
            all_p_values.append(min_p)
        except:
            all_p_values.append(1.0)
        pairs.append((i, j))
```

```python
# === Correction FDR (Benjamini-Hochberg) ===
rejected, pvals_corrected, _, _ = multipletests(all_p_values, alpha=alpha, method='fdr_bh')

# === Matrice prédite corrigée ===
predicted = np.zeros((num_vars, num_vars), dtype=int)
for idx, (i, j) in enumerate(pairs):
    if rejected[idx]:
        predicted[i, j] = 1
```

Résultats obtenus





Résultats obtenus

- **Rappel = 1.0 :** tous les liens vrais ont été retrouvés
- **Précision ≈ 3.7% :** beaucoup de faux positifs
- **F1-score faible**

Forces :

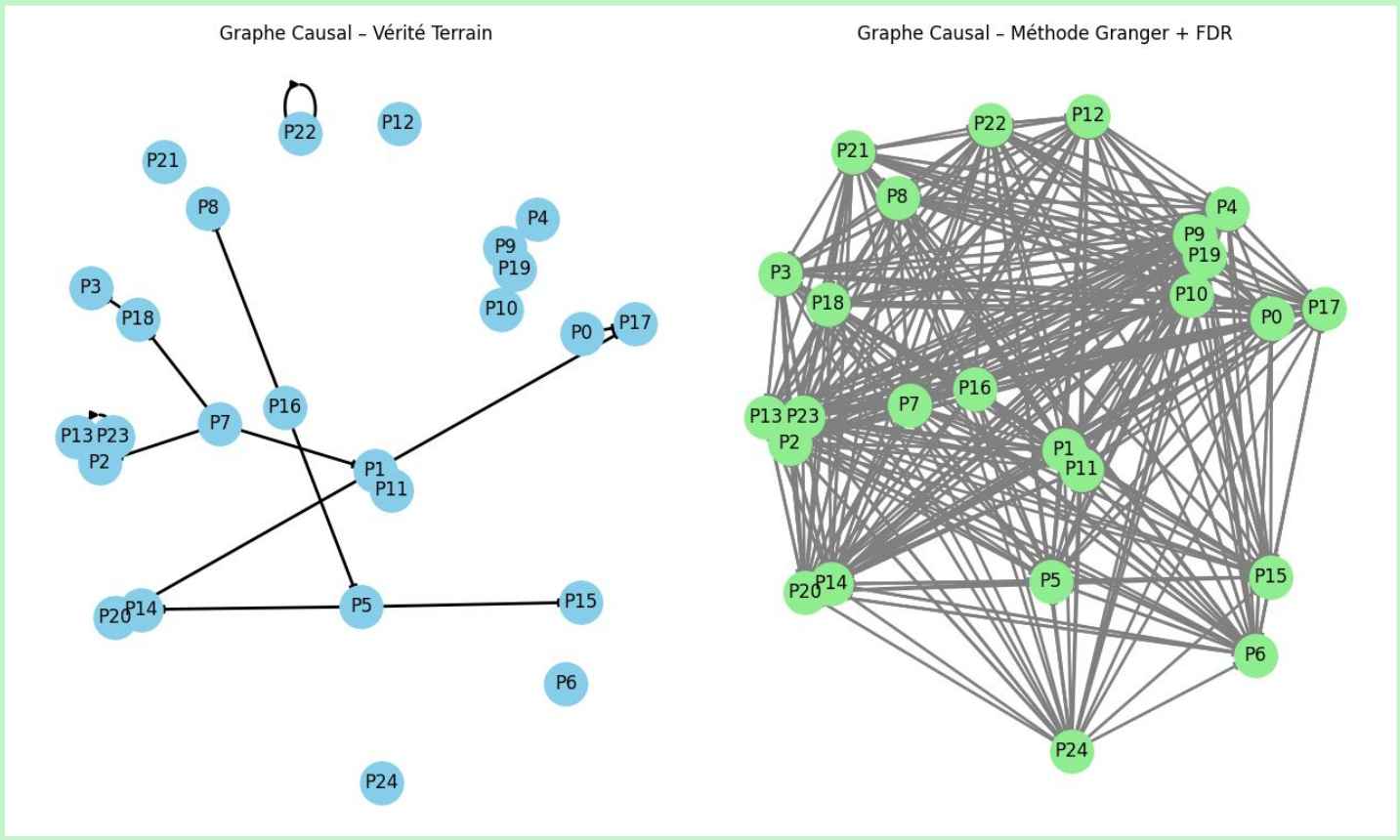
- Méthode rapide, statistiquement fondée.
- Très bon rappel : bonne couverture des vrais liens.

```
=== Résultats avec FDR (Granger) ===
Precision : 0.0373
Recall    : 1.0000
F1-Score  : 0.0719
```

Faiblesses :

- Nombreux faux positifs (précision très faible).
- Ne gère pas bien les relations complexes ou non linéaires.
- Le dataset simulé (random-rels_20_1A) a **20 liens vrais sur 600 possibilités** → très déséquilibré

Visualisation des Résultats



03

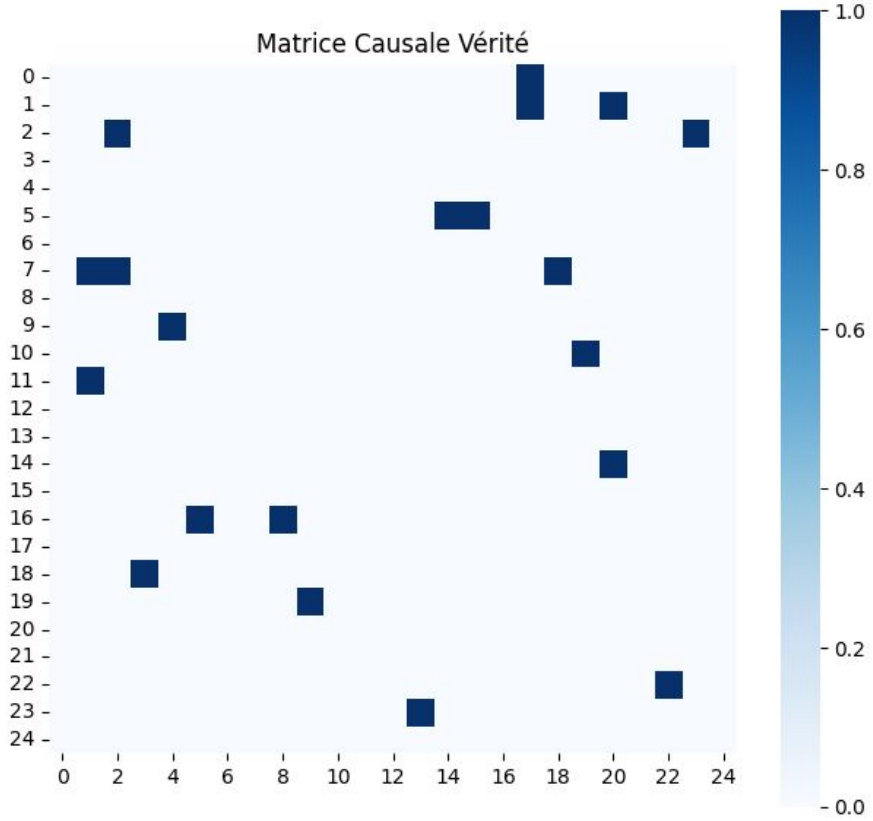
L’implémentation - version classique (PCMCI)

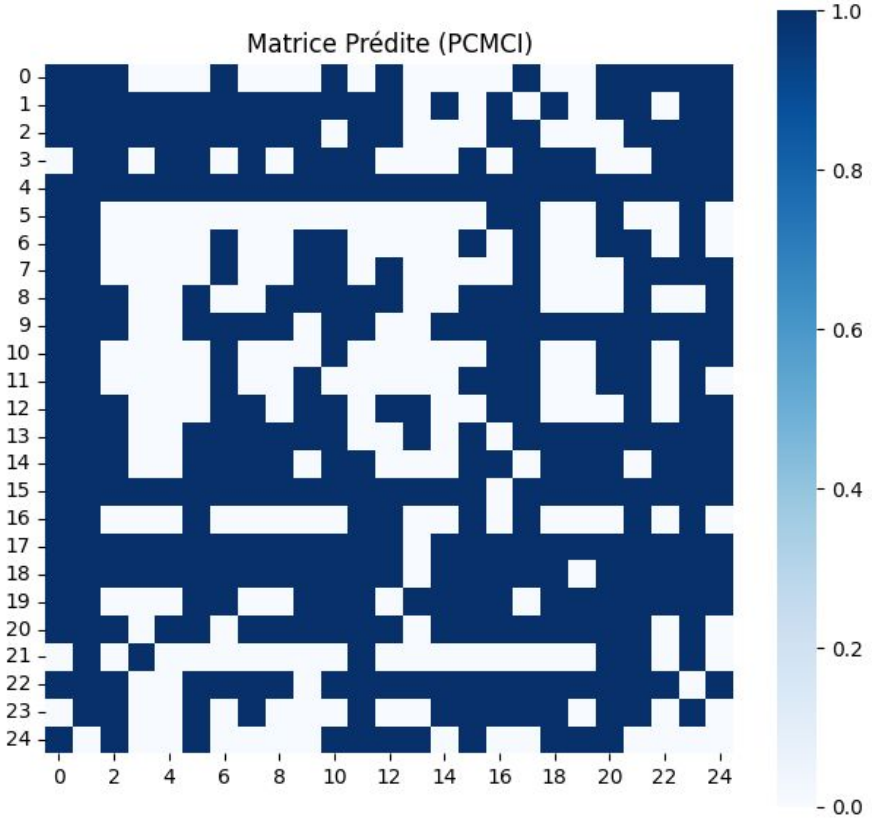
Application de PCMCI (ParCorr)

- Algorithme de découverte causale pour séries temporelles multivariées.
- Utiliser le test **ParCorr** (partial correlation) pour détecter les dépendances linéaires conditionnelles.
- Testé avec un **lag maximal de 3** pour capturer les relations dans le passé.
- Un lien est ajouté si la **p-value corrigée < 0.05**.
- Produit une **matrice causale prédite** comparée à la vérité terrain.
- Évalué avec précision, rappel, F1-score + visualisation par heatmap.

```python
# === Initialisation PCMCI ===
parcorr = ParCorr(significance='analytic')
pcmci = PCMCI(dataframe=dataframe, cond_ind_test=parcorr)

# === Paramètres ===
max_lag = 3
alpha_level = 0.05  # seuil de significativité

# === Exécution de PCMCI ===
results = pcmci.run_pcmci(tau_max=max_lag, pc_alpha=None)

# === Seuil sur les p-values corrigées ===
q_matrix = results['p_matrix']
val_matrix = results['val_matrix']
adj_pred = np.zeros((num_vars, num_vars), dtype=int)

for i in range(num_vars):
    for j in range(num_vars):
        for lag in range(1, max_lag + 1):
            pval = q_matrix[i, j, lag]
            if pval < alpha_level:
                adj_pred[j, i] = 1  # j at t-lag → i at t
```

Résultats obtenus





Résultats obtenus

- Rappel = 0.5 : PCMCI a retrouvé 50% des vrais liens causaux présents dans le graphe de vérité.
- Précision ≈ 2.4% : le graphe prédit contient beaucoup de faux positifs.
- F1-score faible = 0.046 : le modèle prédit trop de liens incorrects comparé à ceux qu'il retrouve correctement

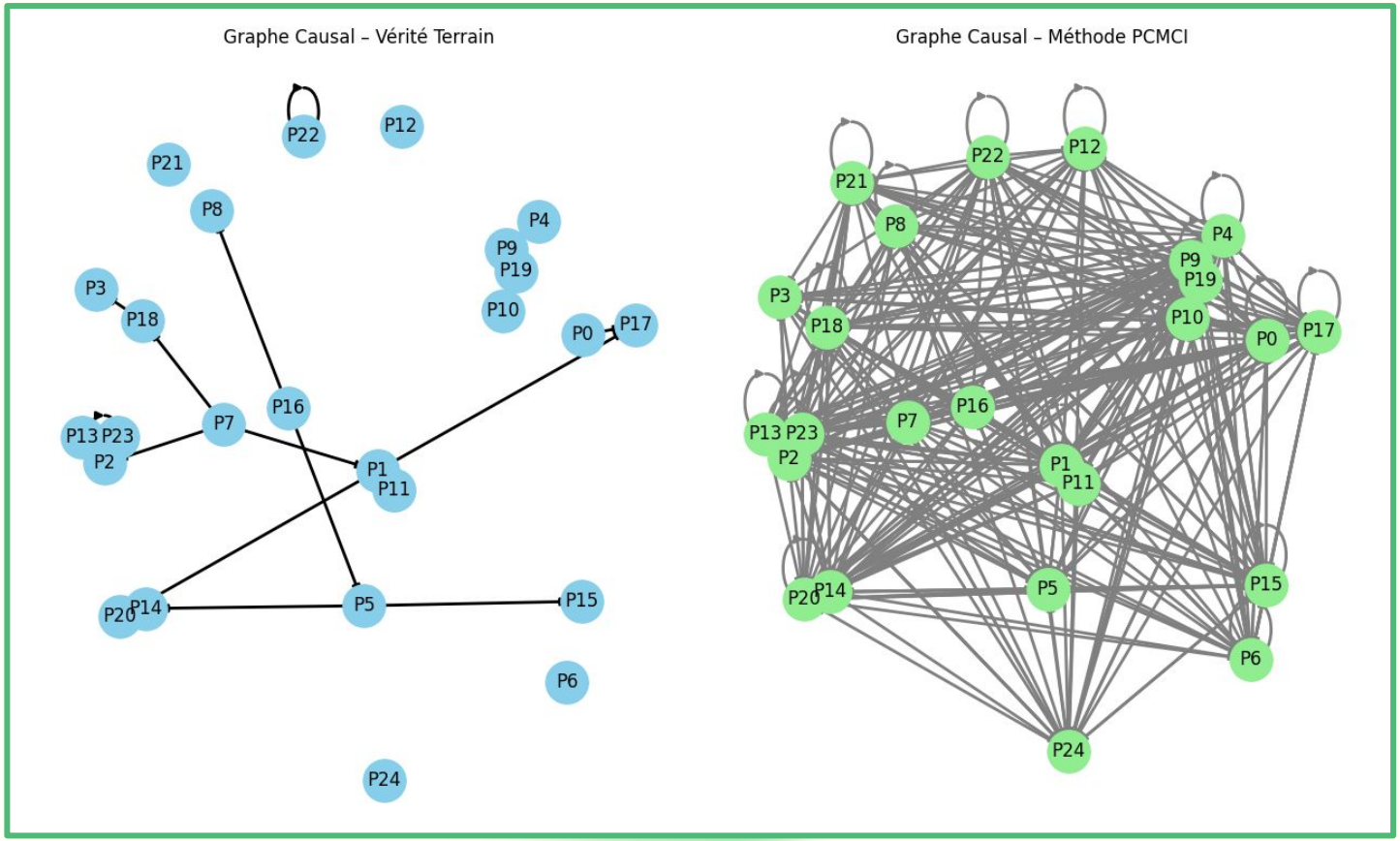
```
=== Résultats avec PCMCI ===
Precision : 0.0241
Recall    : 0.5000
F1-Score  : 0.0460
```

Faible précision :
Le seuil de significativité choisi (alpha = 0.05) peut être trop permissif
Le test d'indépendance utilisé est linéaire (ParCorr) → ne capture pas les relations non-linéaires présentes dans les données.

Rappel moyen :
PCMCI a détecté la moitié des vrais liens, ce qui montre qu'il détecte une partie de la structure causale.
Mais le reste est noyé dans le bruit des faux positifs.

Structure des données :
Les séries du dataset sont fortement corrélées ou avec des lags complexes.
PCMCI fonctionne mieux quand les dépendances sont claires et stationnaires, ce qui n'est pas garanti ici.

Visualisation des Résultats



Application de PCMCI (GPDC)

- Variante de l'algorithme PCMCI utilisant le test **GPDC (Gaussian Process Dependency Criterion)**.
- GPDC permet de détecter des **relations non linéaires** (contrairement à ParCorr qui est linéaire).
- Lag maximum = 3, Seuil de significativité plus strict : α = 0.01.
- Construction de la matrice causale en retenant les p-values corrigées < 0.01.
- Comparaison à la vérité terrain fournie dans le dataset simulé.
- Évaluation à l'aide de précision, rappel, F1-score.

```python
# === Initialisation de PCMCI avec GPDC
ci_test = GPDC(significance='analytic')
pcmci = PCMCI(dataframe=dataframe, cond_ind_test=ci_test)

# === Paramètres
max_lag = 3
alpha_level = 0.01  # Plus strict que 0.05

# === Exécution PCMCI
results = pcmci.run_pcmci(tau_max=max_lag, pc_alpha=None)

# === Construction de la matrice causale prédite
q_matrix = results['p_matrix']
adj_pred = np.zeros((num_vars, num_vars), dtype=int)

for i in range(num_vars):
    for j in range(num_vars):
        for lag in range(1, max_lag + 1):
            if q_matrix[i, j, lag] < alpha_level:
                adj_pred[j, i] = 1  # j at t-lag → i at t
```

Résultats obtenus

- L’implémentation avec GPDC a été extrêmement lente dès le début.
- Même après avoir réduit les données à 10 variables et 1000 points temporels, l’exécution a pris presque des heures.
- L’approche utilise des kernels non linéaires, très coûteux en temps de calcul pour chaque test d’indépendance.
- J’ai tenté différentes optimisations, mais les temps restaient impraticables pour tester plusieurs structures.
- Finalement, j’ai décidé d’abandonner cette approche dans l’évaluation comparative complète.
- Cela montre aussi les limitations actuelles des méthodes avancées sur des machines classiques avec des ressources limitées

04

L’implémentation - version classique (GNN)

05

L’implémentation - version quantique

Resume des dernieres semaines :

Le raisonnement jusqu'à maintenant:

- **(1)** Transformer les séries temporelles en graphes
- → **(2)** Réduire la taille pour rester dans les limites quantiques (clusters)
- → **(3)** Utiliser un circuit QGNN pour apprendre les influences
- → **(4)** Mesurer et entraîner par rétropropagation
- → **(5)** Inférer les relations causales
- → **(6)** Évaluer et interpréter les résultats

Ajouts/modifications

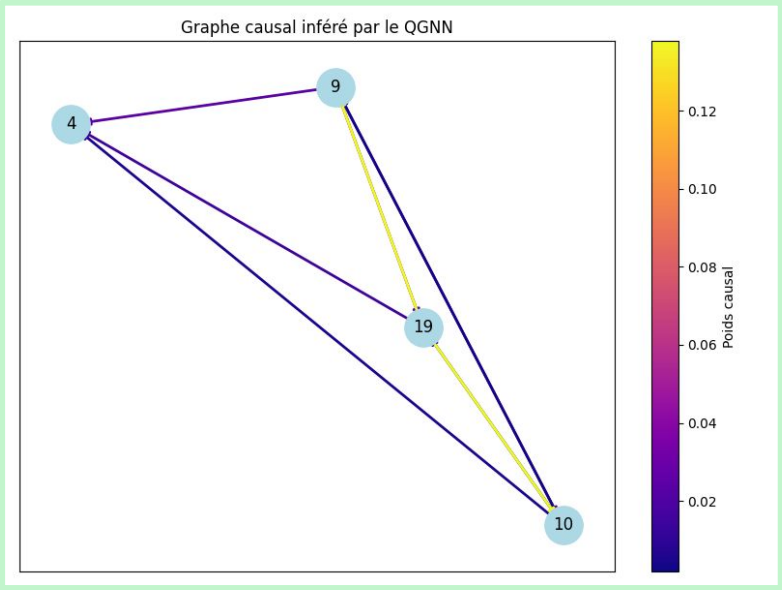
```python
def create_qgrnn_cluster_circuit(graph):
    n_qubits = graph.number_of_nodes()
    dev = qml.device("default.qubit", wires=n_qubits)

    @qml.qnode(dev)
    def circuit(inputs, weights):
        node_list = list(graph.nodes())
        index_map = {node: idx for idx, node in enumerate(node_list)}
        encode_features(inputs)
        qgrnn_layer(graph, weights, index_map)

        # Deuxième couche d'entanglement pour augmenter la profondeur
        for i in range(n_qubits):
            qml.RZ(0.3, wires=i)
        for i in range(n_qubits - 1):
            qml.CNOT(wires=[i, i+1])
        return [qml.expval(qml.PauliZ(idx)) for idx in range(n_qubits)]

    return circuit, dev
```

Deuxième couche de entanglement

```python
prev_losses = []
for epoch in range(epochs):
    weights = opt.step(cost_fn, weights)
    current_loss = cost_fn(weights)
    if epoch % 10 == 0:
        print(f"Époque {epoch}: Perte = {current_loss:.4f}")

    prev_losses.append(round(float(current_loss), 4))
    if len(prev_losses) >= 4:
        if prev_losses[-1] == prev_losses[-2] == prev_losses[-3]:
            print(f"Arrêt anticipé à l'époque {epoch} (early stopping)")
            break
```

Early event/stopping

Toujours pas de bons résultats..




```
mohamedsaoudi@MBP-de-mohamed code % /usr/local/bin/python3
Target Matrix (vraies relations):
[[0. 0. 0. 1.]
 [0. 0. 1. 0.]
 [1. 0. 0. 0.]
 [0. 0. 0. 0.]]
Époque 0: Perte = 0.1783
Époque 10: Perte = 0.1751
Époque 20: Perte = 0.1724
Époque 30: Perte = 0.1702
Époque 40: Perte = 0.1682
Arrêt anticipé à l'époque 46 (early stopping)
Matrice d'influence causale après apprentissage:
[[0.0276 0.0277 0.0281 0.0238]
 [0.0028 0.0155 0.002  0.0054]
 [0.1378 0.1377 0.0679 0.0151]
 [0.     0.     0.     0.0013]]
Précision: 0.50, Rappel: 0.33, F1-score: 0.40
Poids appris par le QGNN (CRY):
[-0.3362 -2.2066  2.4104]
```

3. Construction du circuit quantique QGNN

encode_features(inputs)

- Encode chaque valeur d'entrée avec une rotation RY.
- Ajoute une couche CZ pour entanglement simple entre les qubits voisins.

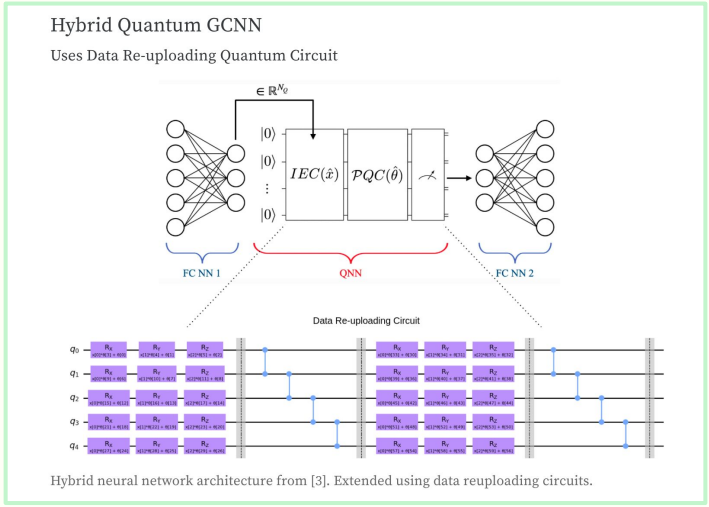
qgrnn_layer(...)

- Pour chaque arête du graphe, applique une porte CRY entre les deux nœuds impliqués.

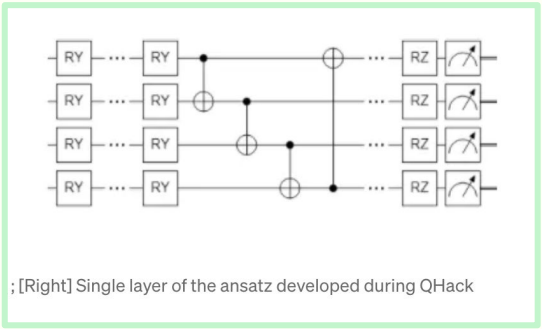
create_qgrnn_cluster_circuit(graph)

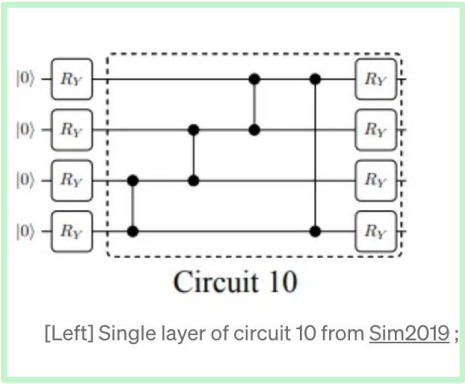
- Crée un circuit quantique :
  - Encode les features.
  - Applique qgrnn_layer.
  - Ajoute une **seconde couche de complexité** : RZ + CNOT (entanglement).
  - Retourne une expectation PauliZ par qubit.

Construction du circuit QGNN (inspiration) :



QGNN for HEP at the LHC





Quantum Graph Neural Networks Applied

4. Mesure d’influence causale

measure_causal_influence(...)

- Mesure l’effet **causal d’un nœud sur les autres**
  - Pour chaque variable i, perturbe son entrée.
  - Compare la sortie du circuit avec celle sans perturbation.
  - Calcule l’effet global de la perturbation i⟶j
- Produit une **matrice d’influence causale** [i,j] = effet de i sur j.

5. Entraînement du QGNN

build_target_matrix(...)

- Crée une matrice binaire avec les vraies relations causales à l’intérieur du cluster choisi.

custom_loss(influence, target)

- Fonction de perte :
  - Pénalise l’absence d’arêtes vraies (faux négatifs).
  - Pénalise la présence d’arêtes fausses (faux positifs).

train_qgrnn_multi_time(...)

- Entraîne les poids CRY du QGNN :
  - Pour plusieurs instants t, reconstruit le graphe local.
  - Mesure l’influence causale à chaque t.
  - Moyenne les pertes.
  - Optimise les poids avec AdamOptimizer.
  - S’arrête à **100 époques**.

Questions

- ....

Prochaines étapes

- Regarder le GNN
- Travailler plus sur le QGNN
- Avancer dans l'écriture du papier

Merci de votre attention!

Université de Sherbrooke

Université TÉLUQ