Causal Graph Identification by LLMs

1. Introduction

Advances in causal inference is vital for many fields and contexts, such as the medical one, where most of the asked research questions are not associational, but causal in nature; with these questions, researchers try to uncover the cause-and-effect relationships between variables (e.g., treatments, interventions, outcomes). These questions cannot be answered from observed data alone and could require specific and expert domain knowledge.

Although expert opinion remains one of if not the best tool for causal analysis (e.g., causal discovery for building causal graphs), it can be very time and resource consuming, since the amount of research data becomes larger and larger (reaching dimensions that limit the possibility of parsing through the enormity of evidence for building DAGs), and experts may eventually commit errors or miss important graph details.   
These difficulties could be partially solved by using LLMs, which have been trained on immense amounts of textual data [5].

1. Project Objective

**The primary goal of this project is to conduct an empirical study to assess the possibility of performing causal analysis using Large Language Models (LLMs).**

The project focuses on the operation of causal discovery, which is the task of learning the structure of causal relationships between variables and entities; its output is a directed graph that represents the underlying data-generation process (DGPO) and provides insight into the true causal relationships between variables. The generated graph is used as a base for many other (if not all) fundamental tasks in causal analysis (e.g. effect inference, prediction, attribution) [1].

The causal discovery task is performed starting from natural language, that is textual data, such as scientific papers and research publications.

The data is then processed to extract the main textual entities; a discovery procedure is then used to find the causal relationship between these entities.

The final operation creates the causal graph using the causal relationships found in the previous step and plots the directed acyclic graph (DAG).

1. Implementation

The project can be divided in two main steps: data collection and data analysis. The former one consisted in collecting the necessary data for the latter, which can itself be divided into multiple other sub-operations.

First, the data collection process will be presented, highlighting the utilization of the National Center for Biotechnology Information (NCBI) API for requesting the necessary textual data. Subsequently, the operations of data processing and causal analysis will be discussed.

PubMed scraping

The first step of the project consisted in collecting the necessary textual data for testing the causal discovery capabilities of the GPT LLM.

The used data is taken from the PubMed database, a free search engine accessing primarily the MEDLINE database of references and abstracts on life sciences and biomedical topics [Pubmed - wikipedia].

Initially, only abstracts (and extra details) were extracted from the PubMed database as a preliminary effort.

**Scraping pipeline**

A pipeline handling the essential operations was created for extracting the necessary textual data from the PubMed database. To automate this extraction process, a python script was written using the public API provided by the NCBI as stable interface into its query and database system.

The pipeline allows the user to extract textual data from PubMed by searching for specific terms.

Immagine che contiene testo, schermata, Carattere, design

Descrizione generata automaticamente

Figure 1 - PubMed data extraction flow

The pipeline's main operations are handled by the *search\_by\_terms*, *get\_articles\_data*, and *clean\_data* procedures.

***search\_by\_terms***

The *search\_by\_terms* procedure is the first operation of the pipeline. As the name suggests, it allows the user to search for articles in the PubMed database containing the specified search terms. The search terms are joined as query parameters in the request URL. An API\_KEY is also sent in the request URL, to allow up to 10 requests per second and to ensure smooth and supported access to the desired resources.

def search\_by\_terms(terms, db, retmax, use\_history):

    terms\_string = '+AND+'.join([s.strip().replace(' ', '+') for s in terms])

    url = f'{base\_url}esearch.fcgi?db={db}&term={terms\_string}&retmax={retmax}&api\_key={api\_key}'

    if use\_history:

        url += '&usehistory=y'

    response = requests.get(url)

    ids = re.findall(r"<Id>(\d+)</Id>", response.text)

    if use\_history:

        web\_match = re.search(r"<WebEnv>(\S+)</WebEnv>", response.text)

        web = web\_match.group(1) if web\_match else None

        key\_match = re.search(r"<QueryKey>(\d+)</QueryKey>", response.text)

        key = key\_match.group(1) if key\_match else None

        return ids, web, key

    return ids

The response is in a xml format, and it is processed to extract all ID numbers of the articles found in the specified NCBI database, which in this case is PubMed.

This simplified version of the first pipeline operation only returns the extracted IDs. However, the script allows users to utilize the NCBI *Entrez History* feature, which proves to be significantly more efficient when dealing with tasks that involve searching for or downloading a substantial number of records. This approach helps streamline the process and optimize the retrieval of records in a more efficient manner, making it possible to upload many IDs or download several hundred records at once.

***get\_articles\_data***

The *get\_articles\_data* procedure is the second step of the data acquisition pipeline. It queries the NCBI for the actual content of the articles with the specified ID.

def get\_articles\_data(ids=[], web\_env='', query\_key='', db='pubmed', retmax=1000):

    url = f'{base\_url}efetch.fcgi?db={db}'

    if use\_web\_env:

        url += f'&query\_key={query\_key}&WebEnv={web\_env}'

    else:

        ids\_string = [str(id) for id in ids]

        url += '&id=' + ','.join(ids\_string)

    url += f'&rettype=abstract&retmode=xml&api\_key={api\_key}&retmax={retmax}'

    response = requests.get(url)

    soup = BeautifulSoup(response.text, features="xml")

    articles = soup.find\_all('PubmedArticle')

    if not articles:

        print('ERROR: No articles found')

        return None

    data = pd.DataFrame(columns=['id', 'title', 'abstract', 'keywords', 'pub\_date'])

    for article in articles:

        id = article.find('PMID').get\_text()

        date = article.find('PubMedPubDate', {'PubStatus': 'received'})

        pub\_date = datetime.strptime(f'{date.find("Day").get\_text()} {date.find("Month").get\_text()} {date.find("Year").get\_text()}', "%d %m %Y")

        title = article.find('ArticleTitle').get\_text()

        abstract = ''.join([a.get\_text() for a in article.find\_all('AbstractText')])

        keywords = [k.get\_text() for k in article.find\_all('Keyword')]

        data = pd.concat([data, pd.DataFrame({'id': id, 'title': title, 'abstract': abstract, 'keywords': [keywords], 'pub\_date': pub\_date})]).reset\_index(drop=True)

    return data

Just like before, the shown function is a simplified version of the second step of the pipeline: the NCBI API allows users to query article data with the article ID or by using the *Entrez History* feature, which can provide a more efficient data retrieval. A URL parameter of the request defines the main data content requested, which, in this case, are the abstracts (rettype=abstract).

The returned data is in a xml format, and it is processed and parsed to extract the necessary information. The recovered data include the abstract of the article and some additional information about the article itself. The additional information are the article ID number, the title, the keywords, and the publication date.

***clean\_data***

The *clean\_data* procedure is the third and last step of the data acquisition pipeline. It performs cleaning operations on the obtained data, e.g., by removing null abstract values, duplicates, and eventually removing data of articles published in a particular date range.

def clean\_data(data, drop\_id\_duplicates, drop\_empty\_abstracts, drop\_nan\_abstracts, drop\_date\_nan, drop\_date\_before, drop\_date\_after, search\_terms):

    if data is None or data.empty:

        print('ERROR: No data provided')

        return None

    if drop\_id\_duplicates:

        data = data.drop\_duplicates(subset=['id']).reset\_index(drop=True)

    if drop\_empty\_abstracts:

        data = data.loc[data['abstract'] != ''].reset\_index(drop=True)

    if drop\_nan\_abstracts:

        data = data.dropna(subset=['abstract']).reset\_index(drop=True)

    if drop\_date\_nan:

        data = data.dropna(subset=['pub\_date']).reset\_index(drop=True)

    if drop\_date\_before:

        data = data.loc[data['pub\_date'] > drop\_date\_before].reset\_index(drop=True)

    if drop\_date\_after:

        data = data.loc[data['pub\_date'] < drop\_date\_after].reset\_index(drop=True)

    if search\_terms:

        data['search\_terms'] = [search\_terms]\*len(data)

    return data

Causal Analysis

After completing the preliminary phase of data collection, the main focus of the project shifted towards the actual analysis operations. The primary objective was to investigate the causal capabilities of LLMs, specifically focusing on causal discovery.

This causal analysis operation consists of learning the causal graphs from a given dataset and context, by uncovering the cause-and-effect relationships and dependencies between the variables and entities of the system of interest: this is done by answering questions such as "Which variables directly affect each other?" or "What is the causal directionality between variables?".

To introduce the causal analysis operation, the next section provides a set of definitions for the main concepts and assumptions on graph theory and causality.

**Definitions** [6]

Definition 1 (Graph). A graph G = (V, E) is a mathematical object represented by a tuple of two sets: a finite set of vertices V and a finite set of edges E ⊆ V × V.

Definition 2 (Directed Graph). A directed graph (DG) G is a graph where the edge (X, Y) is distinct from the edge (Y, X).

Definition 3 (Path). A path π = (X − · · · − Y) is a tuple of non-repeating vertices, where each vertex is connected to the next in the sequence with an edge.

Definition 4 (Directed Path). A directed path π = (X → · · · → Y) is a tuple of non-repeating vertices, where each vertex is connected to the next in the sequence with a directed edge.

Definition 5 (Cycle). A cycle is a path that starts and ends at the same vertex.

Definition 6 (Directed Acyclic Graph). A directed acyclic graph (DAG) is a directed graph G that has no cycles.

Definition 7 (Causal Graph). A causal graph G is a graphical description of a system in terms of cause-effect relationships, i.e., the causal mechanism. Causal graphs, usually in the form of Directed Acyclic Graphs (DAGs), encode contextual knowledge of variables (both observable and unobservable) and their causal dependency.

As mentioned, DAGs are graphical representations of causal relationships among variables, where the arrows indicate the direction of causality.

Definition 8 (Direct and Indirect Cause). For each directed edge (X, Y) ∈ E, X is a direct cause of Y and Y is a direct effect of X. Recursively, every cause of X that is not a direct cause of Y, is an indirect cause of Y.

In a causal graph, the nodes represent the context entities and variables (e.g., in a medical context they would be symptoms, illnesses, diseases, treatments, medications, outcomes, etc. …) while the edges represent the causal relationship between said entities (e.g., a medical treatment can cause a particular outcome or side effect).

The graph encodes the causal relationships between entities and variables. These relationships are represented by different types of edges: directed edges indicate direct causes, bi-directed edges represent entities that are causally correlated with the two variables having an unobserved or latent common cause, and non-existent edges indicate the absence of a causal relationship between the variables.

The acyclic property of DAGs is crucial for ensuring their interpretability and for preserving the causal relationships they represent. This property is fundamental for many reasons: it ensures logical consistency, temporal ordering, identifiability of causal effects, facilitates counterfactual reasoning, and aids in prediction and intervention tasks.

**Causal Analysis Pipeline**

This step of the project consisted in working with the collected data from the PubMed database to extract information from the abstracts. The necessary information consisted of the main named entities of the textual data, which were then used to perform the actual causal analysis.

This second part of the project was implemented in a python script, in the form of a single pipeline of multiple sub-steps, called *causal\_discovery\_pipeline*: these operations include extracting entities from the textual data, performing the actual causal analysis on the found entities, and ultimately generate the resulting causal graph.

Immagine che contiene testo, schermata, Carattere, design

Descrizione generata automaticamente

Figure 2 - Causal analysis flow

Part of the sub-steps of this second part of the project, which involves causal analysis, relies on the GPT API.

**GPT API** [2]

The GPT API is a tool that provides access to OpenAI's GPT models, allowing the integration of natural language processing capabilities into applications. It works by sending requests to the API endpoint with a given prompt, and in return, it generates quality, context-aware text based on the provided input.

The GPT API is used by sending HTTP requests to the API endpoint, specifying the model to use, and providing the necessary parameters such as the prompt and optional additional messages to contextualize the use and behavior of the model, i.e., how the model should answer to requests. The API then processes the request and returns the generated text as a response.

The GPT API can be employed in a variety of applications and use cases. It can be used to generate conversational agents, draft emails or other pieces of writing, provide language translation, answer questions, or assist with content creation.

**Using the GPT API**

The GTP API is used by specifying the model to use (e.g., gpt-3.5-turbo) and additional messages, such as the *system* message and the *user* message. These act as instructions to the model, with the former being a system level instruction to guide the model's behavior throughout the conversation (e.g., asking the model to answer or to act in a specific way), and the latter functioning as the actual request the model is required to answer.

In particular, the system message is used to contextualize the model and its behavior, to make it more useful and accurate for the required operation: for the project’s causal analysis tasks, for example, the prompt was prepended with the message “You are a helpful assistant for causal reasoning”, to try steering the output space to more causally consistent answers. This was shown being an effective prompt-engineering technique that result in more accurate answers [1].

The system message helps set the behavior of the assistant, by modifying the personality of the assistant or providing specific instructions about how it should behave throughout the conversation. However, the system message is optional and the model’s behavior without a system message is likely to be like using a generic message such as "You are a helpful assistant." [2].

messages=[{"role": "system", "content": system\_msg},

          {"role": "user", "content": user\_msg}])

The following is a complete example of a GPT API chat completion request that uses the *gpt-3.5-turbo* model, and specifies system, assistant, and user messages.

openai.ChatCompletion.create(

  model="gpt-3.5-turbo",

  messages=[

        {"role": "system", "content": "You are a helpful assistant."},

        {"role": "user", "content": "Who won the world series in 2020?"},

        {"role": "assistant", "content": "The Los Angeles Dodgers won the World Series in 2020."},

        {"role": "user", "content": "Where was it played?"}

    ]

)

An example of response looks as follows:

{

  "choices": [

    {

      "finish\_reason": "stop",

      "index": 0,

      "message": {

        "content": "The 2020 World Series was played in Texas at Globe Life Field in Arlington.",

        "role": "assistant"

      }

    }

  ],

  "created": 1677664795,

  "id": "chatcmpl-7QyqpwdfhqwajicIEznoc6Q47XAyW",

  "model": "gpt-3.5-turbo-0613",

  "object": "chat.completion",

  "usage": {

    "completion\_tokens": 17,

    "prompt\_tokens": 57,

    "total\_tokens": 74

  }

}

In Python, the assistant’s reply can be extracted as follows:

response['choices'][0]['message']['content']

**GPT Prompt Engineering** [3][4]

Depending on the task, the GPT LLM can produce satisfactory answers when asked to answer a question. However, there are some expedients that have shown to be beneficial and to increase the results accuracy when querying the LLM. These techniques are part of the discipline known as prompt-engineering, a set of rules and instructions used to improve the capacity of LLMs on a wide range of common and complex tasks.

Prompt engineering has emerged as a powerful technique to enhance the performance and control the behavior of Language Models (LMs), particularly Large Language Models (LLMs) such as the used GPT-3.5. Prompt engineering involves crafting system and user messages that guide the model's responses and shape its output according to specific requirements.

The goal of prompt engineering is to provide contextual cues and instructions to the language model, enabling it to generate more accurate, relevant, and desired responses. By designing prompts, it is possible to tailor the behavior of LLMs, making them more suitable for various tasks, domains, and user needs.

Among the many prompt engineering techniques, the main ones applied were (i) prepending the prompt with the message “You are a helpful assistant for {task}” (e.g. for the causal discovery task, the system message was “You are a helpful assistant for causal reasoning”) to try steering the output space to more context consistent answers, (ii) asking a single question (e.g., regarding the direction of the causal dependency: whether A 🡪 B or A 🡨 B), (iii) to answer with a step-by-step explanation [1], and (iv) to ultimately give the final answer in an easily parsable way (e.g., in the “<Answer></Answer>” tags).

**NER: Extracting Medical Entities from Text**

As previously mentioned, the second part of the project consisted in working with the collected data. The first step of the operation involved performing Named Entity Recognition on the abstracts, a fundamental procedure to extract and classify named entities. This step was essential for further processing and analysis.

Named Entity Recognition (NER) is a crucial natural language processing (NLP) task that aims to identify and classify named entities within text. In the context of medical texts, NER plays a vital role in extracting specific medical entities such as diseases, symptoms, treatments, drugs, anatomical terms, and medical procedures. Medical texts pose challenges for NER due to their specialized terminology, which often includes abbreviations and multiple names referring to the same concept. Additionally, the complex language structures found in medical texts, along with the diverse sources from which they originate, further complicate the NER process.

The NER operation was performed using the GPT LLM.

**NER gpt prompt messages**

To enhance the performance of the Language Model (LLM) for the NER task, both the system and user messages were designed accordingly.

The system message employed was "You are a helpful assistant for medical Named Entity Recognition" to provide guidance to the model and improve its understanding of the task at hand.

To further aid the model's comprehension, the user message was crafted using the abstract of the medical text, complemented with additional information about the types of entities to be extracted. In this case, since the texts were focused on medical literature and research publications, the model was explicitly instructed to identify entities, with a particular emphasis on "diseases, medications, treatments, and symptoms".

The intention of customizing the user message by providing relevant context and specific entity requirements, was to guide the LLM towards producing more accurate and relevant results for the ongoing NER operation.

def gpt\_ner(text):

    system\_msg = 'You are a helpful assistant for medical Named Entity Recognition'

    user\_msg = f'Given the following text, please identify the named entities, especially diseases, medications, treatments, symptoms. <Text>{text}</Text>. Answer within the tags <Answer><Entity>...</Entity</Answer>.'

    response = gpt\_request(system\_msg, user\_msg)

    if not response:

        return []

    answer\_text = response.choices[0].message.content

    soup = BeautifulSoup(answer\_text, 'xml')

    entities = [entity.text for entity in soup.find\_all('Entity')]

    return entities

The result of the *gpt\_ner* function is an array containing all the found entities; it is then used for the subsequent causal analysis.

The *causal\_discovery\_pipeline* also allows users to add an optional step for entity optimization: by using the GPT api, the pipeline operation focuses on “*removing redundant or not particularly useful entities that are not diseases, medications, treatments, or symptoms (e.g., “lung cancer” is a valid entity, "lungs" is not).*”

**Causal discovery**

With the NER operation completed and the entities extracted begins the main step of the pipeline, which is the causal discovery operation.

This step consists of the *gpt\_causal\_discovery* function, which processes the text passed as input to the main causal discovery pipeline to perform causal discovery.

The approach for this operation uses a naïve discovery procedure, as it tries to infer the causal relationship across the different variables by querying the LLM on the direction of the pairwise causal relationships for each possible pair combination. The type of causal relationship between a pair of entities corresponds to the edge orientation of the causal graph.

**Causal edge direction**

In a causal Directed Acyclic Graph (DAG), the relationship between a pair of entities can be formalized as either a directed, a bidirected, or a non-existing edge.

The **directed edge** (*A 🡪 B*) denotes a direct causal dependence between the two features A and B, where A is a direct cause of B, without excluding the possible presence of a common cause of both A and B.

The **bi-directed edge** (*A <-> B* or both *A 🡪 B* and *A 🡨 B*) represents a causal relationship where A and B are causally correlated, and the two variables have an unobserved or latent common cause.

A **non-existent** edge denotes that no causal relationship exists between the two variables.

**Possible answers**

To infer the causal edge direction, the pipeline function queries the LLM on which cause-and-effect relationship is more likely given the two entities.

As previously stated, the *system message* used for this operation is *'You are a helpful assistant for causal reasoning'*, to try steering the output space to more causally consistent answers.   
On the other hand, the *user message* presents the current pair of entities of interest, asking a single question about the direction of the causal dependency, and requesting a step-by-step explanation in response. The possible answers the LLM is requested to choose from are also listed in the user message, and these are:

1. "X" causes "Y";
2. "Y" causes "X";
3. "X" and "Y" are not causally related;
4. there is a common factor that is the cause for both "X" and "Y"

def gpt\_causal\_discovery(entities, text, use\_pretrained\_knowledge, reverse\_variable\_check):

    graph\_edges = []

    system\_msg = 'You are a helpful assistant for causal reasoning'

    text\_msg = ''

    text\_msg += ''

    if text:

        text\_msg += f'the following medical text <Text>{text}</Text> '

        if use\_pretrained\_knowledge:

            text\_msg += 'and '

        else:

            text\_msg += ', '

    if use\_pretrained\_knowledge:

        text\_msg += 'your pre-trained knowledge, '

    for i1, e1 in enumerate(entities):

        for i2, e2 in enumerate(entities):

            if i1 == i2:

                continue

            if not reverse\_variable\_check and i1 >= i2:

                continue

            user\_msg = f'Given {text\_msg}the entities "{e1}" and "{e2}", Which cause-and-effect relationship is more likely? A. "{e1}" causes "{e2}"; B. "{e2}" causes "{e1}"; C: "{e1}" and "{e2}" are not causally related; D: there is a common factor that is the cause for both "{e1}" and "{e2}";  Lets work this out in a step by step way to be sure that we have the right answer. Then provide your final answer within the tags <Answer>[answer]</Answer>, (e.g. <Answer>C</Answer>).'

            response = gpt\_request(system\_msg, user\_msg)

            if response:

                graph\_edges.append(((e1, e2), response.choices[0].message.content))

    return graph\_edges

In a default execution mode, the pipeline checks the combinations (without repetition) of all pair of entities found: the total number of queries (one for each pair) is

in the case of ten entities, the total queries are .

**Double variables edge test**

The pipeline also allows for a double test for each pair of variables, by checking all the possible variations without repetition (i.e., relationship “X” - “Y” and “Y” - “X”): this results in querying the LLM about the causal relationship between variables twice for each pair of entities, with a total number of variations of

With ten entities, the total queries are . In this case, the pipeline also performs a check on the answers to ensure compatibility: the answer to the query about the causal relationship between "X" and "Y" must be consistent with the answer to the query about the relationship between "Y" and "X". This is handled by the *check\_invalid\_answers* function, which differentiates between valid and invalid edge directions in terms of answer consistency.

The edge direction and causal relationship between entities associated with "invalid" answers are then re-queried using the *correct\_invalid\_edges* function; the answers are then added to the previously found “valid” edges.

The output of this step of the pipeline is an array containing the type of causal relationship between each pair of entities.

Plotting the causal graph

The next and last operation of the project consists in plotting the resulting causal graph.

**Graph preprocessing**

Before plotting the graph, an intermediate operation of edge and node preprocessing is performed. This preprocessing step aids in the subsequent ones by decoding the LLM answers and converting them into sets of nodes and normalized directed edges (i.e., in the form of *X → Y*).

The main procedure responsible for this operation is the *preprocess\_edges* function. It generates a set consisting of all nodes (previously extracted entities), an array containing the normalized edges, and a dictionary representing the graph, where its keys are the nodes in the graph, and its values are list of the nodes that the given node points to: this dictionary encodes only directed edges that represent a direct causal relationship between entities.  
The normalization of edges is performed by the *normalize\_edge\_direction* function, which takes the nodes and the LLM's answer regarding their causal relationship as input, as well as the dictionary representing the graph. This function processes the output of the LLM to add the resulting node-to-node adjacency to the *graph* dictionary. It then returns the corresponding edge, in the form of X → Y, which represents the causal dependency between the nodes.

def preprocess\_edges(edges):

    graph = {}

    processed\_edges = []

    for (n1, n2), answer in edges:

        if n1 not in graph:

            graph[n1] = []

        if n2 not in graph:

            graph[n2] = []

        direction = normalize\_edge\_direction(n1, n2, answer, graph)

        if direction:

            processed\_edges.extend(direction)

    nodes = graph.keys()

    return nodes, processed\_edges, graph

def normalize\_edge\_direction(e1, e2, answer, graph):

    if answer in arrows:

        if arrows[answer] == forward\_arrow:

            graph[e1].append(e2)

            return [(e1, e2)]

        elif arrows[answer] == backward\_arrow:

            graph[e2].append(e1)

            return [(e2, e1)]

        elif arrows[answer] == bidirectional\_arrow:

            return [(e2, e1), (e1, e2)]

        else:

            return None

    else:

        return None

An additional step is taken before plotting the graph, where a check is performed to determine whether the resulting graph is acyclic: this is done using the *graph* dictionary previously constructed.

**Cycle check**

As previously mentioned, the resulting graph should be a DAG (Directed Acyclic Graph), which is a directed graph without cycles. In the context of causal analysis and causality in general, the acyclic property of graphs is crucial to maintain the logical meaning and coherence encoded within the graph. The absence of cycles ensures that there are no circular dependencies or contradictory relationships, allowing for a clear and meaningful representation of causality in the system.

The *has\_cycle* function is a dedicated procedure designed to check whether the constructed graph has any cycles. It takes a dictionary as an input parameter, representing the graph as an adjacency list with directed edges, meaning only edges with specific directions and no bidirectional ones.   
The function utilizes the *dfs* subroutine, which implements the Depth-First Search (DFS) algorithm to detect cycles in the graph. The DFS algorithm is based on the concept that a graph contains a cycle if there exists a back edge, meaning a node that points to one of its ancestors in the traversal path. If a cycle is found, the function returns *True*; otherwise, it returns *False*, indicating that the graph is acyclic.

The *has\_cycle* function iterates through each node of the graph and calls the *dfs* subroutine, passing the current node as the starting one, along with sets to track visited nodes and nodes in the current recursion path as parameters. The function recursively explores the graph, marking nodes as visited and checking for cycles. If a node is encountered in the current recursion path, it indicates a cycle, and the function returns *True*. If a node has already been visited and does not lead to a cycle, it returns *False*. The function then iterates through all nodes in the graph, initiating *dfs* calls from unvisited nodes.   
If a cycle is found in any of these *dfs* calls, the function returns *True*, indicating the presence of a cycle in the graph. Otherwise, it returns *False*, confirming that the graph is acyclic.

**What to do if the graph has cycles???**

def has\_cycle(graph):

    def dfs(node, visited, recursion\_stack):

        if node in recursion\_stack:

            return True

        if node in visited:

            return False

        visited.add(node)

        recursion\_stack.add(node)

        for neighbor in graph.get(node, []):

            if dfs(neighbor, visited, recursion\_stack):

                return True

        recursion\_stack.remove(node)

        return False

    visited = set()

    for node in graph:

        if node not in visited:

            if dfs(node, visited, set()):

                return True

    return False

**Plotting the graph**

The operation of plotting the causal graph is the last step of the casual discovery pipeline, and it is processed by the *plot\_interactive\_graph* procedure. This function uses Pyvis, a Python library that allows the creation of interactive network graphs in a fast and easy way, with very few lines of code.

The *plot\_interactive\_graph* function takes as input parameters an array containing all the graph’s nodes and an array with all the graph’s edges, normalized in the form of directed edges *X 🡪 Y*.

The resulting interactive graph is then exported as an .html file.

from pyvis.network import Network

def plot\_interactive\_graph(nodes, edges, name='mygraph.html'):

    if not nodes:

        return None

    net = Network(directed=True, notebook=True)

    node\_ids = {}

    for i, node in enumerate(nodes):

        net.add\_node(i, label=node)

        node\_ids[node] = i

    for e1, e2 in edges:

        net.add\_edge(node\_ids[e1],node\_ids[e2])

    net.show(f'../graphs/{name}.html')

Immagine che contiene cerchio, diagramma, Elementi grafici, design

Descrizione generata automaticamente

Immagine che contiene cerchio, diagramma, schermata, Blu elettrico

Descrizione generata automaticamente

**Implementazione**

**Benchmarks**

**Risultati (benchmark + testi veri)**

**Discussione risultati**

**Diagramma flusso aggiornato**

**In flow pipeline analisi causale aggiungere che si parte con un testo 🡪 NER 🡪 …**

Sources:

[1] Causal Reasoning and Large Language Models: Opening a New Frontier for Causality, 02/06 (published May, 2023)

[2] [GPT - OpenAI API](https://platform.openai.com/docs/guides/gpt), 29/06

[3] [Prompt Engineering Guide | Prompt Engineering Guide (promptingguide.ai)](https://www.promptingguide.ai/), 30/06

[4] [Investigating causal understanding in LLMs](https://www.lesswrong.com/posts/yZb5eFvDoaqB337X5/investigating-causal-understanding-in-llms), 02/06

[5] Can Large Language Models Build Causal Graphs?, 12/06 (published March 2023)

[6] [[PDF] A Survey on Causal Discovery: Theory and Practice | Semantic Scholar](https://www.semanticscholar.org/reader/8733ea46dc4909017c7ad186d9c6202933c67b29), 01/06

[7] [Pyvis: Visualize Interactive Network Graphs in Python | by Khuyen Tran | Towards Data Science](https://towardsdatascience.com/pyvis-visualize-interactive-network-graphs-in-python-77e059791f01), 01/07