Towards Formally Describing Program Traces of Chains of Language Model Calls with Causal Influence Diagrams: A Sketch

v1.7

Brian Muhia* [Fahamu Inc]
Joar Skalse[1], Lauro Langosco[1], Esben Kran[1], Fazl Barez[1]
[Apart Research]

August 27 2023

Abstract

In this report, we analyze eleven agent designs implemented by an interface built within the factored cognition framework (https://primer.ought.org/), and use casual influence diagrams (CIDs) (Everitt et al, 2021) to formally describe the agent interactions, their influences and how they contribute towards performing a user query, based on context from a document archive.
This helps us in documenting the system while making plans for future changes, such as adding a reranking step, or adding verifiers that remove unwanted outputs or associations.
A research program on interface design is outlined, wherein we plan the use of mechanistic interpretability tooling to improve the causal influence diagrams by adding some weak notion of conditional probability distributions, and hypothesise their possible contribution towards formally modeling and representing the introduction of deceptive answers in a dataset of responses from agents involved in generating subquestions, filtering, answering sub-questions and aggregating context.

Introduction

When we compose small, independent contributions from agents that know about the data they're being asked about, we introduce a bias for representing the documents provided, which is counterbalanced by the bias from the models involved and how they have been trained, which may improve the answer quality by faithfully and factually representing the context, or they may entirely ignore relevant parts of the context and focus on others, or go off topic entirely.

We aim to illuminate this complex problem first by formally describing how the designs of two agent calls interact with context data and then by discussing how we might in the future analyze the dataset using a smaller local model with a causal tracing framework in order to visualize the factuality of agent responses with an animated CID. The specifics of this and analysis of the related data are left to future work.

[1]Research conducted at the Apart Research Alignment Jam #8 (Safety Verification), 2023 (see https://alignmentjam.com/jam/verification)
*Primary author email: brian[at]fahamuai.com

Related Work

The Causal Incentives working group (https://causalincentives.com/) has supported the release of the Python library PyCID (https://github.com/causalincentives/pycid), which we use to automate the visualisation process. Prior to this, a graphical formalism that has been demonstrated to describe large language model interactions is Language Model Cascades (https://arxiv.org/abs/2207.10342)

Process-Based Supervision

We aim to understand how systems designed through the task decomposition framework generate answers by observing and controlling data flow through the process by which answers are generated. We hope to learn more about how the specific type of answering process is working in practice by tracing chained calls to large language models and formalising the resulting program trace in a diagram that documents it. For this, we have several data points to build on, including:

- User question
- Answer
- Data Source (list of paragraphs before filtering)
- Context (list of paragraphs after classifying and filtering)
- (Optional) Sub-questions & Sub-answers (Generated by sub-agents)

We aim to use data to study the causal relationships between each of these steps, as they are occurring at different times in the process of answering a user's question, and thus influence each other in a directed acyclic graph.

Causal Influence Diagrams in Practice

(Try the colab for the code: (https://colab.research.google.com/drive/1roLQgXhEtI83Q5vX1q24Q9iDgu5LFFWA?usp=sharing)

Definition: A **Causal influence Diagram (CID)** (Everitt et al, 2021) is a tuple (**V**,**E**) where:

- (**V**,**E**) is a directed acyclic graph with a set of
- vertices **V**
- connected by directed edges $\mathbf{E} \subseteq \mathbf{V} \times \mathbf{V}$

These vertices are partitioned into:

- $\mathbf{D} \subseteq \mathbf{V}$ is a set of decision nodes represented by rectangles.
- $\mathbf{U} \subseteq \mathbf{V} \setminus \mathbf{D}$ is a set of utility nodes represented by diamonds and utility nodes have no successors.
- $\mathbf{X} = \mathbf{V} \setminus (\mathbf{D} \cup \mathbf{U})$ is the set of chance nodes represented by ovals.

Definition: A **Causal influence Model (CIM)** (Everitt et al, 2021) is a tuple (**V**,**E**,*θ*) where (**V**,**E**) is a CID and:

- *Θ* is a parameterisation over the nodes in the CID specifying:
    - A finite domain $dom(V)$ for each node $V \in \mathbf{V}$
    - Real-valued domains $dom(U) \subset \mathbb{R}$ for all utility nodes $U \in \mathbf{U}$.
    - A set of conditional probability distributions (CPDs), $\Pr(\mathbf{V} \mid pa_V)$, for every chance and utility node $\mathbf{X} \cup \mathbf{U} \in \mathbf{V}$. Taken together, these form a partial distribution $\Pr(\mathbf{X},\mathbf{U}:\mathbf{D}) = \prod_{V \in \mathbf{V} \setminus \mathbf{D}} \Pr(V \mid Pa_V)$ over the variables in the CID.

In this report we only describe CIDs, leaving the automated program tracing and mechanistic interpretability work that would get us CIMs for future research.
Examples showing eleven pycid CIDs. These are manually written to describe two LLM chains that are currently being tested. The future hope is to use mechanistic interpretability[11] first to introduce numerical weights and thus find a basis for representing CPDs with some *weak* notion of (positive or negative) contribution to the total factuality of the response. If there is merit to this, we aim to design a mechanism for automating the generation of these *weakly* factuality-weighted CIDs when provided with a program trace.
Notation:

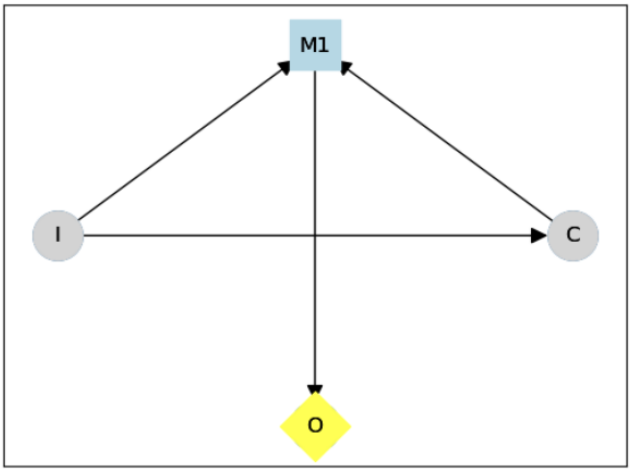
- $\mathbf{M_n}$ = Models/Agents are represented as **decision nodes**
- **C** = Context. This is the set of agent instructions and related documents
- **I** = User Input (or Intent)
- **O** = Output is a utility node

Single Agent

This first call instantiates an agent **M1** to answer a single user question, and a list of paragraphs **C** obtained after a separate search and filtering process as context, using it to answer the question as output **O**. We take the arrow to mean conditioning via prompting in this example.

```
# !pip install pycid
# code to generate the diagrams appears next to each diagram

import pycid
m1_cid  = pycid.CID(
    [
        ("C", "M1"),
        ("I", "C"),
        ("I", "M1"),
        ("M1", "O")
    ],
    decisions=["M1"],
    utilities=["O"]
)
m1_cid.draw()
```



Single interaction with the model **M1**. To explain this diagram, imagine the user **I** (for intent), typing a sentence in a search bar and pressing "Ask", which sends a message to a search process in the database to find documents that will get filtered for context **C**. The paragraphs that end up in **C** are there because of what the user asked**.** So the path **(I, C)** means "user asks a question and the search process finds paragraphs for context". If there is an arrow from the **I** node to a model $\mathbf{M_n}$ we say **(I, $\mathbf{M_n}$)**, which means the user's intent is included in the model's context. If there is no arrow from the **I** node to a model $\mathbf{M_n}$, we say the user's intent is omitted. The twin paths **(I, M1)** and **(C, M1)** mean "the user question and paragraphs are sent to the model". The path **(M1, O)** means "the model outputs its response and gets feedback".

Multiple Agents

Here we explore the idea that it may be safer, or more robust in some cases to design an agent that can itself call more agents to help it gather more context, or verify its output.

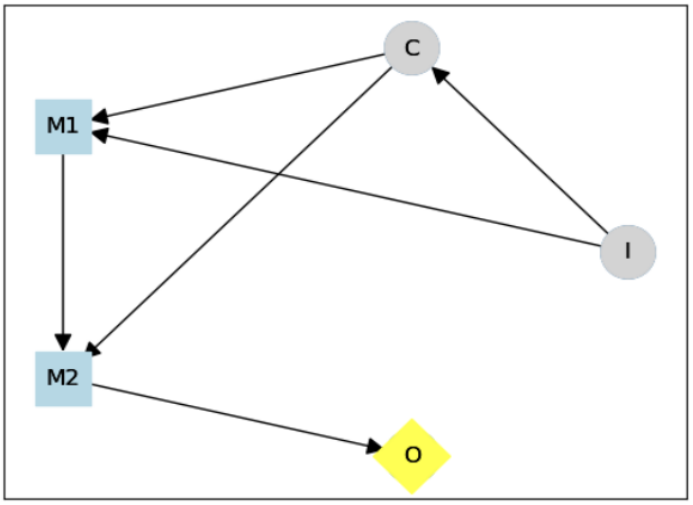
2-Agent and 3-Agent Scenarios

We explore two simple cases that show the power of the CID formalism for explaining and understanding, in principle, the data flow and interactions we expect to happen in practice. We can think about what makes sense, and these can become a place to discuss any needed interventions before implementing complex architectures.

Sequential pair of agents

Agent **M1** is instructed to ask one related question that can help answer the user's question/intent **I**. The second agent **M2** takes agent **M1**'s output and the context from the user search. Note the missing **(I, M2).** This means that the user's intent would not be directly passed to the agent that actually answered them, so if an accidental inaccuracy was introduced by **M1**, or if **M2** doesn't fully take context **C** into account, it might be missing important context that would have been passed from **(C, M2)** or **(M1, M2)** and would deceive the user**.**

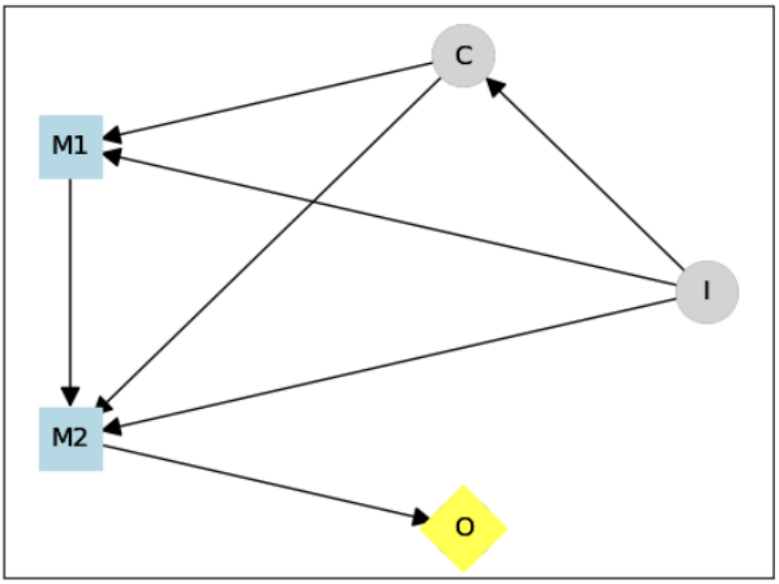
```
pair_1  = pycid.CID(
    [
        ("I", "C"),
        ("I", "M1"),
        ("C", "M1"),
        ("C", "M2"),
        ("M1", "M2"),
        ("M2", "O"),
    ],
    decisions=["M1", "M2"],
    utilities=["O"]
)
pair_1.draw()
```



We correct this in the next drawing by adding **(I, M2).**

One purpose of this study is to explore intent alignment using some notation that helps us think about actual design decisions that could be made. Throughout the rest of the examples we will show variants of multi-agent designs where we omit the user's intent from certain steps in the calls. Empirically, we do this by removing the user's question from the sub-agent's prompt. Think about (or empirically verify) what happens in the cases where we do this vs where it is maintained. In Appendix A we develop three rules that encode these intuitive arguments in a SAT solver that automatically checks if the diagrams are satisfiable.

```
pair_1_correct  = pycid.CID(
    [
        ("I", "C"),
        ("I", "M1"),
        ("I", "M2"),
        ("C", "M1"),
        ("C", "M2"),
        ("M1", "M2"),
        ("M2", "O"),
    ],
    decisions=["M1", "M2"],
    utilities=["O"]
)
pair_1_correct.draw()
```

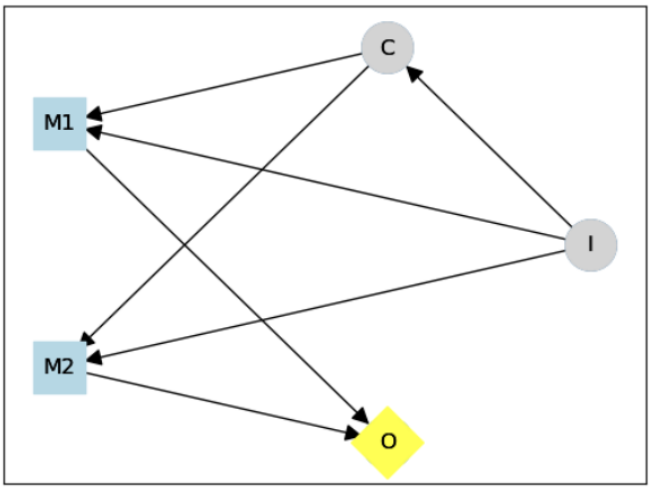


When the graph is correct we see how the user's intent is present in every step of the call chain, which increases the likelihood that it is satisfied, even if each agent **M1** and **M2** is equally unreliable. However any unreliability in **M1** would be amplified by its output being piped to **M2**.

Parallel 2- and 3-agent designs.

We want to think about parallel agents as well, primarily due to latency.

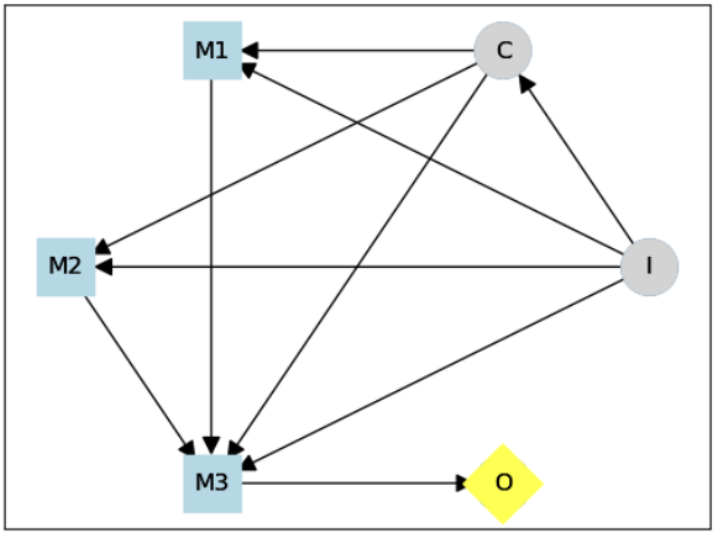
```
parallel_1  = pycid.CID(
    [
        ("I", "C"),
        ("I", "M1"),
        ("I", "M2"),
        ("C", "M1"),
        ("C", "M2"),
        ("M2", "O"),
    ],
    decisions=["M1", "M2"],
    utilities=["O"]
)
parallel_1.draw()
```



This breaks down a bit if both **M1** and **M2** are wrong, we would have two wrong answers. That seems much worse. Also, for interface reasons we wouldn't want output from two models stored in a data store and shown to the user, we instead want output from one model

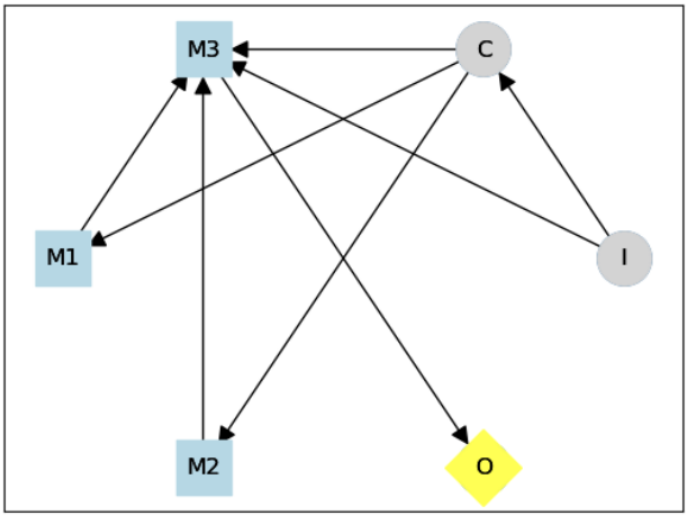
that aggregates, modifies or otherwise filters **M1** and **M2**'s responses. Here's the parallel example updated to add a third agent to aggregate the output from parallel agents.

```
parallel_2  = pycid.CID(
    [
        ("I", "C"),
        ("I", "M1"),
        ("I", "M2"),
        ("I", "M3"),
        ("C", "M1"),
        ("C", "M2"),
        ("M1", "M3"),
        ("M2", "M3"),
        ("C", "M3"),
        ("M3", "O"),
    ],
    decisions=["M1", "M2", "M3"],
    utilities=["O"]
)
parallel_2.draw()
```



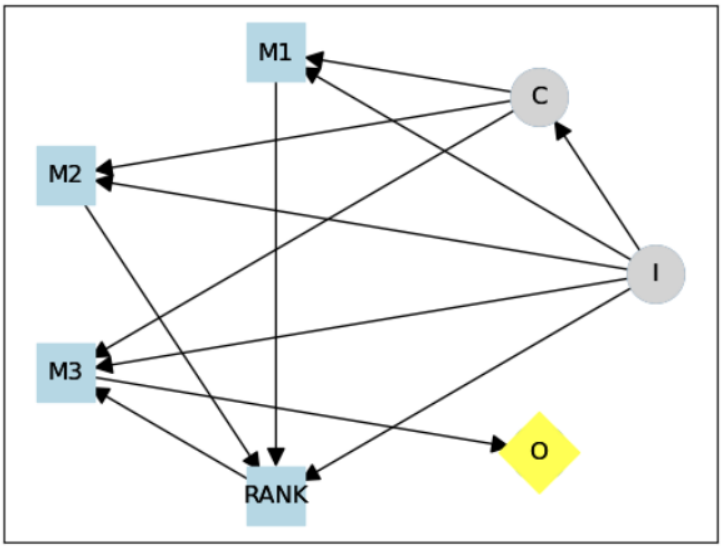
It’s a bit unusual to look for parallel steps and find 3, instead of 2 agents that do the right thing. Maybe it’s a rule. Maybe if you want parallel steps, you don't want two agents, you want three agents. Also, parallel agents involve at least two “sequential steps”. In terms of information flow, there are two parallel paths from **C** to **M3**, through **M1** and **M2** at once. Note how, by counting the incoming arrows, both models **M1** and **M2** have two sources of information, while **M3** has four. Claim: the more agents there are in the population, the more pressure there will be on at least one of them to get it right. To start thinking about how this might help with alignment, which is relevant for the more complex cases, we can show a variant of this which omits the user’s intent from the parallel steps **M1** and **M2**. In this case, they get no user input and are just asked to do something with the context **C**. Claim: if both **M1** and **M2** get it wrong or can’t generalise and make something up, **M3**’s output will fail to satisfy user intent.

```
parallel_2_omit  = pycid.CID(
    [
        ("I", "C"),
        ("I", "M3"),
        ("C", "M1"),
        ("C", "M2"),
        ("M1", "M3"),
        ("M2", "M3"),
        ("C", "M3"),
        ("M3", "O"),
    ],
    decisions=["M1", "M2", "M3"],
    utilities=["O"]
)
parallel_2_omit.draw()
```



Next, we show a variant which adds a ranking step **RANK**, as a new **decision node**, to re-rank or filter answers from **M1** and **M2,** while maintaining user intent at all steps.

```
parallel_2_rank  = pycid.CID(
    [
        ("I", "C"),
        ("I", "M1"),
        ("I", "M2"),
        ("I", "M3"),
        ("C", "M1"),
        ("C", "M2"),
        ("I", "RANK"),
        ("M1", "RANK"),
        ("M2", "RANK"),
        ("RANK", "M3"),
        ("C", "M3"),
        ("M3", "O"),
    ],
    decisions=["M1", "M2", "M3", "RANK"],
    utilities=["O"]
)
parallel_2_rank.draw()
```



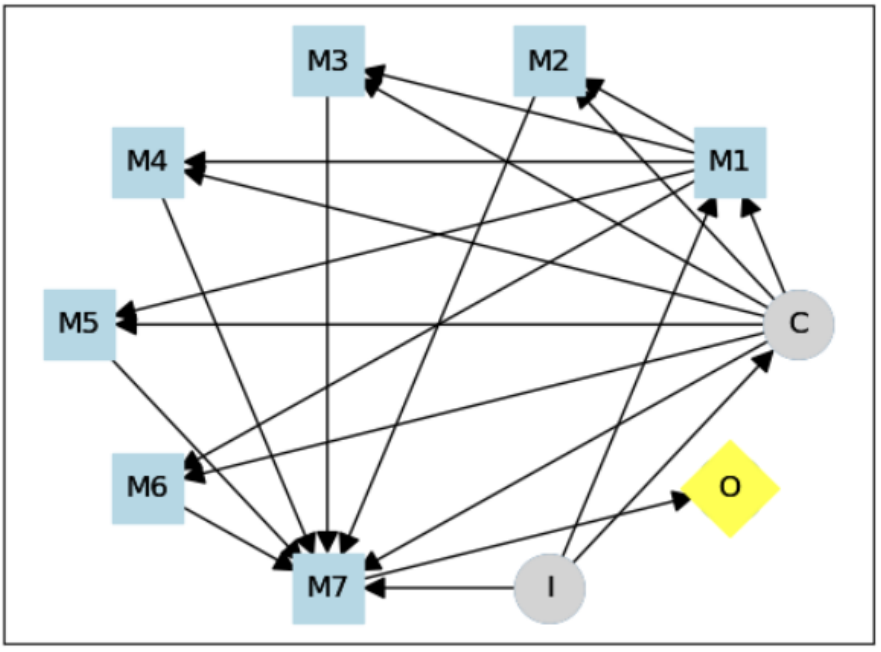
What follows are a few somewhat more complex scenarios, and more discussion of the safety relevance of this work.

7-Agent Parallel Scenarios

The second set first instantiates a single agent **M1** to generate 2-5 sub-questions based on user input **I** and the document context **C**. This instantiates 5 parallel sub-agents (**M2**-**M6**) to answer each of these sub-questions using the context, and a 7th agent **M7** to aggregate the sub-answers and context into a more detailed answer as output **O**.

(7-PV1) Parallel Variant 1 where the user's intent is omitted from the context of agents M2-M6.

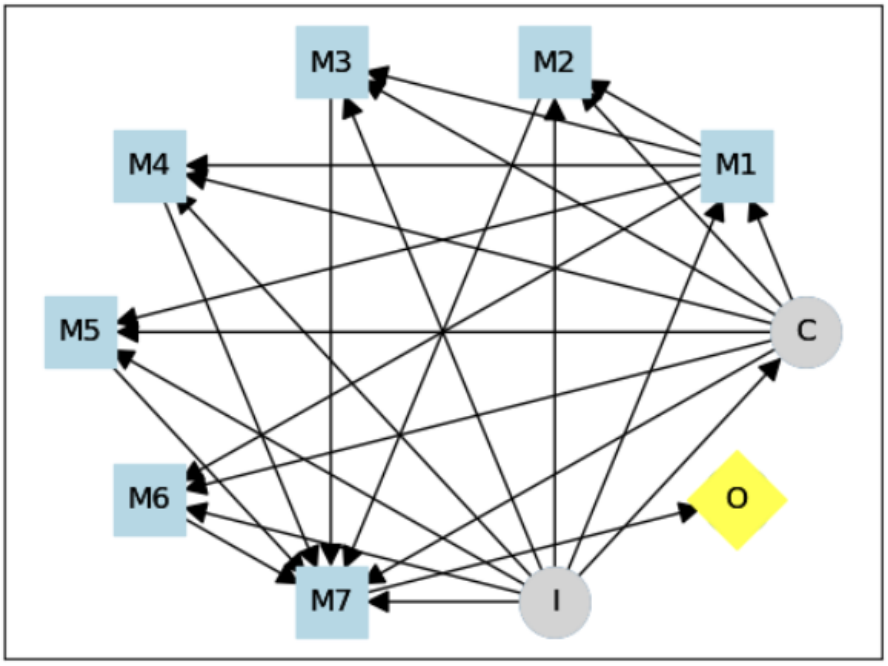
```
variant_1  = pycid.CID(
    [
        ("C", "M1"),
        ("C", "M2"),
        ("C", "M3"),
        ("C", "M4"),
        ("C", "M5"),
        ("C", "M6"),
        ("C", "M7"),
        ("I", "M1"),
        ("I", "M7"),
        ("M1", "M2"),
        ("M1", "M3"),
        ("M1", "M4"),
        ("M1", "M5"),
        ("M1", "M6"),
        ("M2", "M7"),
        ("M3", "M7"),
        ("M4", "M7"),
        ("M5", "M7"),
        ("M6", "M7"),
        ("M7", "O"),
    ],
    decisions=["M1", "M2", "M3", "M4", "M5", "M6", "M7"],
    utilities=["O"]
)
variant_1.draw()
```



My current hypothesis is that there are millions of these diagrams and also that there are easy ways to see possible failure modes in the designs of factored cognition schemes before they are implemented. For a concrete example of the parallel variants, see *A. Stuhlmüller and J. Reppert and L. Stebbing (2022)*[10].

(7-PV2) Parallel Variant 2 where the user's intent is included in the context of all agents M1-M7.

```
variant_2  = pycid.CID(
    [
        ("C", "M1"),
        ("C", "M2"),
        ("C", "M3"),
        ("C", "M4"),
        ("C", "M5"),
        ("C", "M6"),
        ("C", "M7"),
        ("I", "M1"),
        ("I", "M2"),
        ("I", "M3"),
        ("I", "M4"),
        ("I", "M5"),
        ("I", "M6"),
        ("I", "M7"),
        ("M1", "M2"),
        ("M1", "M3"),
        ("M1", "M4"),
        ("M1", "M5"),
        ("M1", "M6"),
        ("M2", "M7"),
        ("M3", "M7"),
        ("M4", "M7"),
        ("M5", "M7"),
        ("M6", "M7"),
        ("M7", "O"),
    ],
    decisions=["M1", "M2", "M3", "M4", "M5", "M6", "M7"],
    utilities=["O"]
)
variant_2.draw()
```



7-Agent Sequential Scenarios

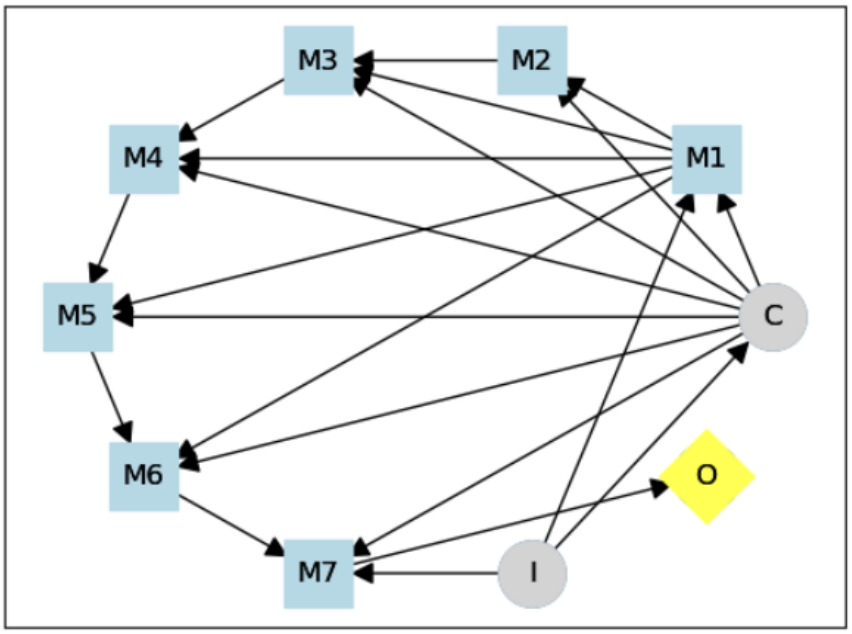
We clean up the two above variants to prevent data duplication in the prompt, and add a few more links to describe 2 more variants (**7-SV1** and **7-SV2**) that explore the idea of chaining in a more sequential way, where data flows directly from $\mathbf{M_n}$ to $\mathbf{M_{n+1}}$. So whatever **M1** starts to do, it may be amplified, verified, filtered out or somehow modified many times by the time **M7** encounters it. This all depends on the design of data flow. We use this formalism to note that as **decision nodes** with some level of independence and unreliability, factuality might be broken at any stage and we would want to detect it.

By intuition, my prediction is that in sequential scenarios errors would compound in subtle ways to more sub-agents faster than in parallel scenarios.

There are still more opportunities for either early stopping or self-healing in the sequential case since assuming the detection of a lack of factual grounding in e.g. node **M3** in the two Sequential variants, means that we may have an opportunity to remove that sub-answer and retry, or remove that node entirely by blanking out the text. **Qn:** So what would be a good way to deal with factuality issues in Parallel Agents **M2-M6**?

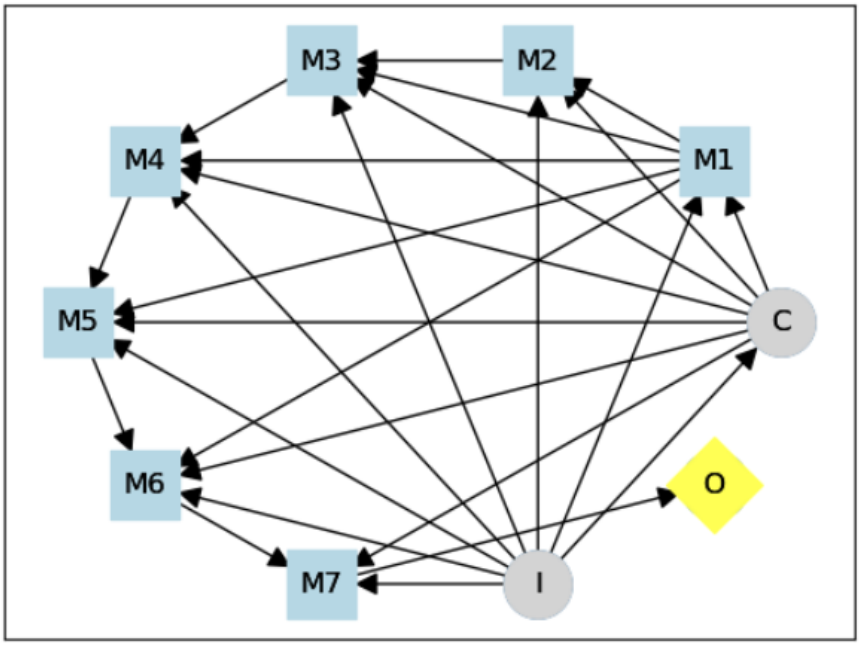
(7-SV1) Sequential Variant 1 where the user's intent is omitted from the context of agents M2-M6.

```
variant_3  = pycid.CID(
    [
        ("C", "M1"),
        ("C", "M2"),
        ("C", "M3"),
        ("C", "M4"),
        ("C", "M5"),
        ("C", "M6"),
        ("C", "M7"),
        ("C", "M7"),
        ("I", "C"),
        ("I", "M1"),
        ("I", "M7"),
        ("M1", "M2"),
        ("M1", "M3"),
        ("M1", "M4"),
        ("M1", "M5"),
        ("M1", "M6"),
        ("M2", "M3"),
        ("M3", "M4"),
        ("M4", "M5"),
        ("M5", "M6"),
        ("M6", "M7"),
        ("M7", "O"),
    ],
    decisions=["M1", "M2", "M3", "M4", "M5", "M6", "M7"],
    utilities=["O"]
)
variant_3.draw()
```



(7-SV2) Sequential Variant 2, where the user's intent is included in the context of all agents M1-M7.

```
variant_4  = pycid.CID(
    [
        ("C", "M1"),
        ("C", "M2"),
        ("C", "M3"),
        ("C", "M4"),
        ("C", "M5"),
        ("C", "M6"),
        ("C", "M7"),
        ("I", "M1"),
        ("I", "M2"),
        ("I", "M3"),
        ("I", "M4"),
        ("I", "M5"),
        ("I", "M6"),
        ("I", "M7"),
        ("M1", "M2"),
        ("M1", "M3"),
        ("M1", "M4"),
        ("M1", "M6"),
        ("M1", "M5"),
        ("M2", "M3"),
        ("M3", "M4"),
        ("M4", "M5"),
        ("M5", "M6"),
        ("M6", "M7"),
        ("M7", "O"),
    ],
    decisions=["M1", "M2", "M3", "M4", "M5", "M6", "M7"],
    utilities=["O"]
)
variant_4.draw()
```



Exercises

1. How would one filter the data flow to **M7** and prevent deceptive answers from the previous agents from causing **M7** to deceive the user, assuming each agent is equally unreliable.
2. Which of **7-PV1** vs **7-PV2** or **7-SV1** vs **7-SV2** are more aligned with the user's intent?
    a. When you consider all of them, together?
    b. Could we rank them in order of "alignment to user intent"?
    c. Which would be more likely to cause an accident?
    d. Could we tell just from analysing the drawing, i.e. before implementing them?

Hint: Count the arrows coming from the **I** nodes. Again, If there is an arrow from e.g. **(I, M4)**, we say the user's intent is included in **M4**'s context. We might assume that **M7** gets the same information in both (Sequential and Parallel) cases of **Variant 1**, but that variant either benefits, or fails creatively or catastrophically, from a slightly more independent search for answers that may or may not be faithful to the context, given its instructions and the sub-questions generated by **M1**. See the Appendix for an example of how we formalise these intuitions into an automated checker that relies on conventional SAT solvers.

Relationship to ML Safety

1. If we zoom in on one problem, like accidental inaccuracy, we can try to think about which group of architectures propagates errors faster, and which intervention would fix these issues. For example, a decision to include a re-ranker or filtering model would be demonstrated in one of these drawings, and a discussion could be had about data flow, and at which point. Preventing errors from accumulating until they reach **M7** would be a strong priority.
2. We consider the "prompt", or any incoming arrow in the diagrams, to be the goal. If **7-SV2** was a hierarchical planner, it would create approval-guided sub-agents, whereas **7-SV1** would not. We can tell if we only consider the goals, rather than the behaviour of the agents. **7-SV1** sub-agents could experience a drift in their goals and misrepresent what the user asked for. We would therefore not get the benefits of approval-guided action, or search, for those steps of the overall system's operation.
3. Assuming a system that is given a lot of power over its environment is coordinated in this way, we would want to characterise coordination failures from agents that are assumed to be equal to one another but each have the same failure mode that can compound across agents. A coordination failure in **7-SV1** might get much worse very quickly when left unchecked. The strategy of intervening on the data flow in each step to create an illusion of safety breaks down when the pillars themselves are unstable.
4. Formalising the ways in which multi-agent systems interact with their users is a purpose of the causal incentives program. My aim in this is to add more links between the three dense subgraphs of factored cognition, mechanistic interpretability and causal incentives.
5. Understanding the ways each multi-agent interaction intersects with questions related to introduced unfairness/bias/lack of generalizability through training methods and dataset imbalance.

6. In cases when we want to use human feedback to improve answer quality, the design of the structure of the reward modeling scheme is something people have to think very carefully about. Designing RLHF reward schemes is a complex problem primarily due to the problem of over optimization and reward hacking. Maybe we can illuminate this problem by using contributions to total factuality to explore different reward mechanisms that use this information.[9]
7. Representing debates as mechanised causal games would probably help us develop another level of analysis over debate scenarios.
8. Can we model more general failures in AGI safety scenarios, like treacherous turns?
9. In the **RANK** variant, the decision to make the **RANK** node blind to the input, or intent of the user, by removing the **(I, RANK)** node, might seem like a bad idea at first, but consider the scenario of a prompt injection attack. If the **RANK** node was blind to the input, and was already biased towards the contents of the data source supporting the context **C** (perhaps by being trained on that distribution of documents), and it was also biased towards shorter responses than majority of the contents of **C**, then in positive scenarios the right thing would happen, but during an attack the ranker would prioritise the contents of the context, being from the same distribution. Therefore it would be safer to make the ranker blind to input if it also matched the distribution of the context.
10. Scenarios where the first model in either a sequential or parallel scenario (**M1**) produces unfaithful Chain-of-Thought instructions or explanations for the next set of agents, leading to deceptive output, are demonstrated in the wild in (https://arxiv.org/abs//2305.04388). Claim: The CID framework can aid in discussion of flaws in systems that are already in the wild.

Acknowledgements

Thanks to Bhishmaraj S, Jonas Kgomo, Eliezer Yudkowsky, Ryan Carey and Joanny Raby for providing useful references, comments and suggestions.

References

1. Causal Incentives Working Group https://causalincentives.com
2. PyCID: A Python Library for Causal Influence Diagrams https://conference.scipy.org/proceedings/scipy2021/pdfs/james_fox.pdf
3. Agent Incentives: A Causal Perspective https://arxiv.org/pdf/2102.01685.pdf
4. Modeling AGI Safety Frameworks with Causal Influence Diagrams https://arxiv.org/pdf/1906.08663.pdf
5. Understanding Agent Incentives using Causal Influence Diagrams, Part 1: Single Action Settings https://arxiv.org/abs/1902.09980
6. Locating and Editing Factual Associations in GPT https://rome.baulab.info/
7. Language Models Don't Always Say What They Think: Unfaithful Explanations in Chain-of-Thought Prompting (https://arxiv.org/pdf/2305.04388.pdf)
8. Language Model Cascades (https://arxiv.org/abs/2207.10342)
9. Do the Rewards Justify the Means? Measuring Trade-Offs Between Rewards and Ethical Behavior in the MACHIAVELLI Benchmark (https://arxiv.org/abs/2304.03279)
10. Factored Cognition Primer: One-Step Amplification (https://primer.ought.org/chapters/amplification/one-step-amplification)
11. Anthropic - Attention: Theory, Info-Weighted Patterns, Attribution Patterns [rough early thoughts] (https://www.youtube.com/watch?v=etFCaFvt2Ks)
12. Answer Set Solving in Practice (https://ieeexplore.ieee.org/book/6813208)
13. Detecting inconsistencies in large biological networks with answer set programming (https://doi.org/10.1017/S1471068410000554)

Appendix

A: Answer Set Programming for Automated Verification of Intent Consistency

The causal influence diagrams introduced here, and the accompanying reasoning that favours certain diagrams over others based on links to the "I" node, are simple enough that we can devise automated rules that check if a diagram is correct or wrong. We call this property "intent consistency". Here we introduce three simple rules written in the Answer-Set Programming (ASP)[12] formalism that

1. find paths between any two nodes X and Y, then check if
2. a path exists from the input node "I" to any decision node, and
3. fails if there is no direct link from the node "I" to any decision node.

These rules encode our expectations and intuitions, and let us describe a framework for automatically deciding if a diagram satisfies them. We call these rules "intent consistency models" (ICM), after [13]. Here we show an example of this verification strategy for the first agent diagram on page 3.

```
%*
For a pycid CID:

pycid.CID(
  [("C", "M1"),
   ("I", "C"),
   ("I", "M1"),
   ("M1", "O")],
  decisions=["M1"],
  utilities=["O"]
)
*%

% ...after encoding the CID using ASP facts link(), decision(), chance() and utility(),
link("C", "M1").
link("I", "C").
link("I", "M1").
link("M1", "O").

decision("M1").
utility("O").
chance("I";"C").

% Recursive rule to find a path from X to Y
path(X, Y) :- link(X, Y).
path(X, Y) :- link(X, Z), path(Z, Y).

% Checks if a path exists from I to any decision node
check(Node) :- decision(Node), path("I", Node).

% Rule that fails if there is no direct link from I or C to any of the decision nodes.
direct_link(I, D) :- link(I, D), decision(D).
:- decision(D), not direct_link("I", D).
:- decision(D), not direct_link("C", D).


#show direct_link/2.
#show check/1.
```

This is a complete program that can be copied and checked without modification. ASP enables us to encode the graphs described here using facts, and run them through a conventional SAT solver like 'clingo' that checks for satisfiability. We can then describe unsatisfiable graphs as "incorrect". More examples of this proof strategy can be seen at this URL.