PyCID: A Python Library for Causal Influence Diagrams

James Fox[‡*], Tom Everitt[§], Ryan Carey[‡], Eric Langlois[¶], Alessandro Abate[‡], Michael Wooldridge[‡]

Abstract—Why did a decision maker select a certain decision? What behaviour does a certain objective incentivise? How can we improve this behaviour and ensure that a decision-maker chooses decisions with safer or fairer consequences? This paper introduces the Python package *PyCID*, built upon *pgmpy*, that implements (causal) influence diagrams, a widely used graphical modelling framework for decision-making problems. By providing a range of methods to solve and analyse (causal) influence diagrams, *PyCID* helps answer questions about behaviour and incentives in both single-agent and multi-agent settings.

Index Terms—Influence Diagrams, Causal Models, Probabilistic Graphical Models, Game Theory, Decision Theory

Introduction

Influence-diagrams (IDs) are used to represent and analyse decision making situations under uncertainty [HM05], [MIMH+76]. Like Bayesian Networks, IDs have at their core a directed acyclic graph (DAG), but IDs also specify decision and utility nodes. Relationships between variables are given by conditional probability distributions. When these are specified, we call it an influence model (IM). In an IM, a decision-maker selects a distribution over its available actions at a decision (a decision rule) based on what it knows (the values of its parents in the ID) to maximise its expected utility. To demonstrate, consider the following example:

Grade Prediction: To decide who to admit, a university uses a model to predict the grades of applicants based on information in their application forms.

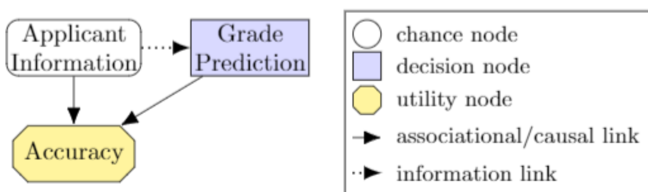

Fig. 1: *A (C)ID for the* ***Grade Prediction*** *example.*

Figure 1 shows the DAG for this example, which displays clearly the structure of the decision situation. The decision being made by an agent, the model, is the grade prediction (decision node). The agent selects a decision rule for this decision, based on information about the applicant (chance node), in order to optimise their prediction accuracy (utility node). The edges denote associational relationships in the case of a statistical IM, but denote causal links in causal influence models (CIMs). This difference in semantics [ECL+21] allows one to use CIMs to query the effect of causal interventions and provides a setting to ask counterfactual questions [Pea09]. (C)IMs have also been extended to multi-agent settings by [KM03], [HFE+21], and [HFE+].

Statistical and causal IDs have shown promise for a wide variety of applications. In business and medical decision making, statistical IDs provide a simple yet powerful model for optimising decisions by making assumptions explicit and revealing what information is relevant [Góm04], [KM08]. Moreover, for the design of safe and fair AI systems, causal IDs have been used to help predict the behaviour of agents arising due to their incentives in an environment [ECL+21], [CLEL20], [EHKK21], [Hol20], [EKKL19], [LE21], and [CVH20]. Nevertheless, although Python libraries exist for Bayesian networks, perhaps most prominently *pgmpy* [AP15], these libraries lack specific support for IDs. We found two Python wrappers of C++ influence diagram libraries: *pyAgrum* [DBDSMW20] and *PySMILE* [Bay]. These were limited by usability (hard to install), maintainability (using multiple languages) and versatility (they did not cover multi-agent or causal IDs). A Python library that focuses on implementing statistical and causal IDs is therefore needed to ensure their potential application can be explored, probed, and fully realised.

Consequently, this paper introduces *PyCID*[1], a Python library built upon *pgmpy* [AP15] and *NetworkX* [HSS08], which implements IDs and IMs (including their causal and multi-agent variants) and provides researchers and practitioners with convenient methods for analysing decision-making situations. *PyCID* can solve single-agent (C)IMs, find Nash equilibria in multi-agent (C)IMs, and compute the effect of causal interventions in CIMs (e.g., fixing the prediction model in Figure 1 to always predict a high grade regardless of the applicant's information). *PyCID* can also find which variables in an ID admit incentives. For example, positive value of information [How66] and value of control [Sha86] tell us when an agent can benefit from observing or controlling a variable. Meanwhile, other incentives concepts, recently proposed in [ECL+21], reveal which variables it can be instrumentally useful to control and when a decision-maker benefits from responding to a variable. Reasoning patterns are a related concept in multi-agent IDs: they analyze why a decision-maker would care about a decision [PG07], and these can also be

* Corresponding author: james.fox@cs.ox.ac.uk
‡ University of Oxford
§ DeepMind
¶ University of Toronto

Copyright © 2021 James Fox et al. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

computed in *PyCID*.

The first two sections of this paper provide the necessary background on (C)IDs and describe the architecture of the *PyCID* library. We then move to showcasing some of *PyCID*'s features through applications for discovering agent incentives and analysing games. In the * Instantiating Causal Influence Diagrams* section, we demonstrate how to instantiate a (C)ID for the **Grade Prediction** example in *PyCID*. In the *Analysing Incentives* section, we demonstrate how to find the nodes which admit value of information, response, value of control, or instrumental control incentives for more complex (C)IDs. We then turn to multi-agent (C)IDs (MA(C)IDs) and show how to use *PyCID* to compute Nash equilibria. Next, we explain how *PyCID* can construct random (MA)CIDs. Finally, we discuss the future of *PyCID*.

Background

Notation

Throughout this paper, we will use capital letters, X, for random variables and let $dom(X)$ denote their domain. An assignment $x \in dom(X)$ to X is an instantiation of X denoted by $X = x$. $\mathbf{X} = \{X_1, \dots, X_n\}$ is a set of variables with instantiation $\mathbf{x} = \{x_1, \dots, x_n\}$. We also let $\mathbf{Pa}_V$ denote the parents of a node V in a (MA)CID and $\mathbf{pa}_V$ be the instantiation of $\mathbf{Pa}_V$. Moreover, we define $\mathbf{Desc}_V$ and $\mathbf{Fa}_V := \mathbf{Pa}_V \cup \{V\}$ to be the descendants and family of V. We use subscripts to index the elements of a set and, in a multi-agent setting, superscripts to indicate a player $i \in \mathbf{N}$; e.g., the set of decisions belonging to player i is $\mathbf{D}^i = \{D^i_1, ..., D^i_n\}$.

Causal Influence Diagrams

A *Bayesian network* is a model consisting of a directed acyclic graph (DAG) and a joint distribution that is Markov compatible with that graph [Pea09]. The nodes in the DAG denote random variables and the directed edges represent the associational relationships between them. To parameterise the DAG and encode the joint distribution, each random variable, V, in the DAG is assigned a conditional probability distribution (CPD), $P(V|\mathbf{Pa}_V)$, dependent on its set of graphical parents, $\mathbf{Pa}_V$. Taken together, these CPDs define the Bayesian network's joint distribution.

A *causal Bayesian network* is a Bayesian network where the directed edges in the DAG now represent every causal relationship between the Bayesian network's variables. This enables the model the ability to answer questions about the effect of causal interventions from outside of the system.

Causal Influence Diagrams (CIDs) are DAGs where the nodes are partitioned into chance, decision, and utility nodes and the edges adopt the same causal semantics as causal Bayesian networks [ECL+21]. Causal Influence models (CIMs) are parameterised CIDs where, at the outset, the CPDs for chance and utility nodes are defined, but only the domains for the decision variables are fixed.

Definition 1 [ECL+21] A **Causal influence Diagram (CID)** is a directed acyclic graph $(\mathbf{V}, \mathbf{E})$ where the set of vertices $(\mathbf{V})$ connected by directed edges $(\mathbf{E} \subseteq \mathbf{V} \times \mathbf{V})$ are partitioned into chance $(\mathbf{X})$, decision $(\mathbf{D})$, and utility $(\mathbf{U})$ nodes. Utility nodes lack children.

Definition 2 [ECL+21] A **Causal influence Model (CIM)** is a tuple $(\mathbf{V}, \mathbf{E}, \theta)$ where $(\mathbf{V}, \mathbf{E})$ is a CID and $\theta \in \Theta$ is a particular parametrisation over the nodes in the graph specifying for each node $V \in \mathbf{V}$ a finite domain $dom(V)$, for each utility node $U \in \mathbf{U}$ a real-valued domain $dom(U) \subseteq \mathbb{R}$, and for every chance and utility node a conditional probability distribution (CPD) $P(V \mid \mathbf{Pa}_V)$.

Multi-agent Causal Influence Diagrams (MACIDs) partition decision and utility nodes further into sets associated with each agent. In a (MA)CID, a decision rule, $\pi_D(D|\mathbf{Pa}_D)$, is a probability distribution over the actions available at decision node D conditional on the value of its parents in the graph, $\mathbf{Pa}_D$. A policy, π^i, assigns decision rules to all of agent i's decision nodes, and, in a MACIM, a policy profile, π, assigns policies to every agent. In a (MA)CID, each agent i's expected utility, $\mathcal{U}^i_{\mathcal{M}}(\pi)$, under a policy (profile) π is the sum of the expected values of their utility nodes.

Package Architecture

In this section, we outline the structure (Figure 2) and describe the key classes of the *PyCID* library[2].

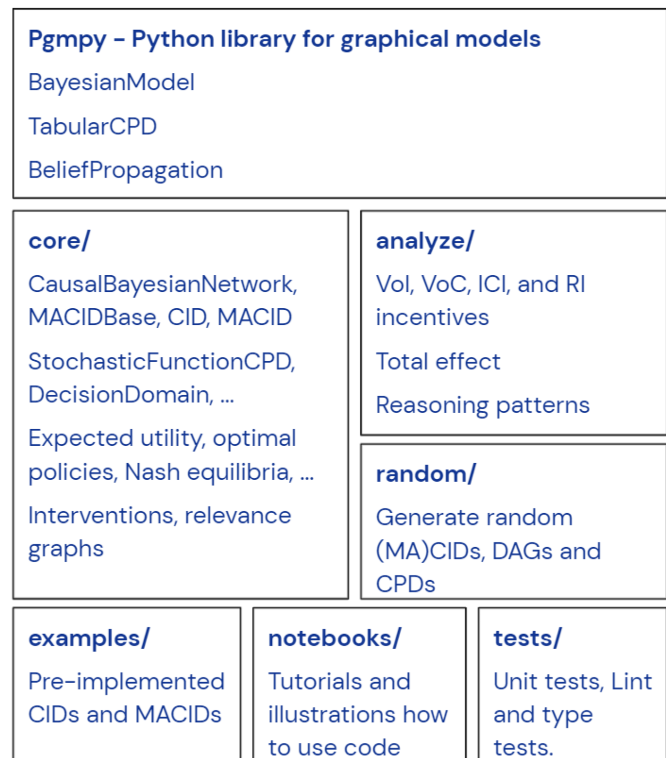

***Fig. 2:** An overview of* PyCID*'s file structure.*

Installation

PyCID is released under the *Apache License 2.0*. It requires *Python 3.7* or above, but only depends on *Matplotlib* [Hun07], *NetworkX* [HSS08], *NumPy* [HMvdW+20], and *pgmpy* [AP15]. It can be downloaded and installed in a Python virtual environment or in a Conda environment using:

```
python3 -m pip install pycid
```

PyCID is under continual development and so one can install the latest developmental package using a git checkout from the PyCID repository on GitHub: https://github.com/causalincentives/pycid.

1. This paper describes *PyCID* version 0.2.6.

2. *PyCID* is under continued development, so more features will be added over time. Any updated documentation may be found in the repository's README file.

Classes Inherited from pgmpy

PyCID's key classes inherit from *pgmpy*'s `BayesianModel`, `TabularCPD`, and `BeliefPropagation` classes [AP15]. The `BayesianModel` class represents a *Bayesian network* and CPDs are assigned to each random variable in the model using instances of the `TabularCPD` class. These CPDs define the *Bayesian Network*'s joint distribution and the `BeliefPropagation` class is then used to perform probabilistic inference on a `BayesianModel` object; for instance, one can query the probability that node V takes value v given some instantiation of other variables in the DAG (known as a *context*).

The pycid.core module

PyCID's base class is `CausalBayesianNetwork`. This class inherits from *pgmpy*'s `BayesianModel` and represents a *causal Bayesian network*. In particular, it extends `BayesianModel` by adding the ability to query the effect of *causal interventions*. It also adds methods for determining the expected value of a variable for a given *context* (again under an optional *causal intervention*) and for plotting the DAG of the *Causal Bayesian Network* using *NetworkX* [HSS08]. CPDs for a `CausalBayesianNetwork` object can be defined using *pgmpy*'s `TabularCPD` class, but we also allow relationships to be specified more directly with stochastic functions (under the hood, these are implemented via a `StochasticFunctionCPD` class). This can be used to specify relationships between variables with a stochastic function, rather than just with a probability matrix (see the **Instantiating Causal Influence Diagrams** section). `CausalBayesianNetwork` also has an inner class, `Model`, which keeps track of CPDs and domains for all `CausalBayesianNetwork` objects' variables in the form of a dictionary.

The `MACIDBase` class, which inherits from `CausalBayesianNetwork`, provides the underlying methods necessary for single-agent and multi-agent causal influence diagrams. The class includes methods for determining the expected utility of an agent, for finding optimal decision rules and policies, and for finding various new graphical criteria defined in influence diagrams (e.g. r-relevance).

`CID` and `MACID` are classes, inheriting from `MACIDBase`, that represent single-agent and multi-agent (C)IDs and are the models of most concern in *PyCID*. They include methods for finding the optimal policy for an agent in a (C)IM and for finding Nash equilibria [N+50] and subgame perfect Nash equilibria [Sel65] in a MA(C)IM. It is important to highlight here that statistical (i.e., non-causal) single-agent and multi-agent influence diagrams can also be defined as `CID` and `MACID` objects using *PyCID*. In their case, all class methods are permitted except those that involve causal interventions.

The *pycid.core* module also contains functions that exploit relationships between the (MA)(C)ID's variables such as finding all (active) (directed) paths between variables and classes that find the relevance graphs [KM03] associated with `MACIDBase` objects.

PyCID's other modules

The *pycid.analyse* module includes functions for determining incentives in (C)IDs [ECL+21], reasoning patterns in MA(C)IDs [PG07], and a function for computing the *total effect* of intervening on a variable with different values. *pycid.examples* contains pre-implemented (C)IDs and MA(C)IDs, whilst *pycid.random* contains functions for generating random (C)IDs and MA(C)IDs. *pycid.notebooks* contains *jupyter notebooks* with demonstrations of how to use the codebase; these can also be run directly as *Colab notebooks*. Finally, *pycid.tests* houses unit tests for all functions and public class methods.

Instantiating Causal Influence Diagrams

Having covered *PyCID*'s basic library structure, the remaining sections will demonstrate some use cases. We begin, in this section, by instantiating the structure of the simple (C)ID given in the introduction (Figure 1). For many purposes, including finding incentives, the graph is enough for analysis.

A (C)ID for the **Grade Prediction** example is created as an instance of our `CID` class. Its initializer takes a list of edges as its first argument and then two more lists specifying the (C)ID's decision and utility nodes. All other nodes introduced in the edge pairs, which are not decision or utility nodes, are chance nodes. For conciseness, we abbreviate and use P to denote the prediction model's decision node, A for the applicant's information, and Ac to denote the accuracy of the predictions:

```
import pycid
cid = pycid.CID(
    [("A", "P"), ("A", "Ac"), ("P", "Ac")],
    decisions=["P"],
    utilities=["Ac"],
)
cid.draw()
```

The `CID` class method, *draw*, plots this (C)ID (Figure 3) with a node colour and shape convention that matches what is given in Figure 1's legend.

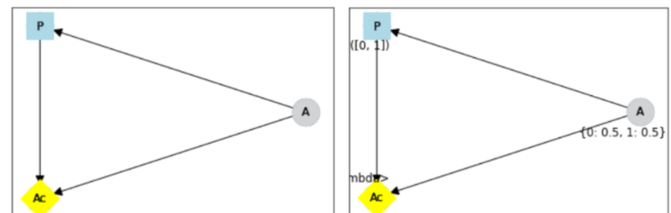

Fig. 3: *A simple (C)ID (Left) and corresponding CIM (Right) plotted using* PyCID.

To then parameterise this (C)ID as a (C)IM by adding a domain for P and CPDs for A and Ac, we pass keyword arguments to the *add_cpds* method:

```
1 cid.add_cpds(
2     A=pycid.discrete_uniform([0, 1]),
3     P=[0, 1],
4     Ac=lambda a, p: int(a == p),
5 )
```

CPDs in *PyCID* can be instantiated directly as `TabularCPD` objects, but more often *PyCID*'s `StochasticFunctionCPD` subclass is used. This provides multiple ways to easily specify how a chance or utility variable's CPD depends on its parents or follows some distribution; it then converts that expression into a `TabularCPD` object under the hood. On line 2 above, we assign variable A a discrete uniform distribution over its domain, $dom(A) = \{0,1\}$; on line 3, we specify $dom(P) = \{0,1\}$; and on the final line, we specify how the value of Ac depends on the values of its parents, A and P. Within the lambda function, other variables are referred to by their lower case form to denote that variable's instantiation. Using a `CID` class method, *solve*, we can

now solve this (C)IM by finding the agent's optimal decision rule for P. This returns the following output, saying that the optimal decision rule for P is to choose action 0 (low grade prediction) when the value of A is 0 (the quality of the application is poor), and action 1 (high grade prediction) when the value of A is 1 (the quality of the application is high):

```
{'P': StochasticFunctionCPD<D>
   {'a': 0}  ->  0
   {'a': 1}  ->  1}
```

If the agent behaves according to this optimal decision rule, we find that their expected utility is 1 using the code below; *expected_utility* accepts optional dictionaries for specifying contexts and causal interventions:

```
solution = cid.solve()
optimal_d_cpd = solution['P']
cid.add_cpds(optimal_d_cpd)
cid.expected_utility(context={}, intervention={})
```

There are several other ways to specify CPDs for variables. For example, on line 1 below, the CPD for A is updated to now follow a Bernoulli(0.8) distribution and line 2 specifies that now Ac just copies the value of P with probability 0.7:

```
1    cid.add_cpds(A=pycid.bernoulli(0.8))
2    cid.add_cpds(Ac=lambda a, p: pycid.noisy_copy(p,
3                       probability=0.7, domain=[0, 1]))
```

Analysing Incentives

In this section, we demonstrate how to use *PyCID* to find which nodes in a single-decision CID admit different types of incentives using their graphical criterion [ECL+21]. In general, a graphical criterion tells you what properties influence models can have based on the influence diagram (i.e, the graph) alone. A graphical criterion takes a graph and several nodes as arguments and returns whether or not the property (in this case the incentive) can occur for those nodes. Incentives are helpful for applications in safety and fairness ([ECL+21], [Hol20]), understanding the behaviour of RL algorithms ([LE21], [EHKK21]), and comparing the promise of different AGI safety frameworks [EKKL19]. We believe that *PyCID* can further mature these enquiries.

PyCID currently finds the following incentives in single-decision CIDs using their graphical criteria:

- Value of Information (VoI)
- Response Incentives (RI)
- Value of Control (VoC)[3]
- Instrumental Control Incentives (ICI)

Value of Information (VoI)

Intuitively, a variable has positive value of information (VoI) if a decision-maker would benefit (get more utility) from observing its value before making a decision:

VoI Definition: For a CIM[4] $\mathscr{M}$, and a node $X \in \mathbf{V} \setminus \mathbf{Desc}_D$, let $\mathscr{M}_{X \not\to D}$ and $\mathscr{M}_{X \to D}$ be $\mathscr{M}$ modified by respectively removing and adding the edge $X \to D$. The **value of information** for X is then $\max_\pi \mathscr{U}^i_{\mathscr{M}_{X\to D}}(\pi) - \max_\pi \mathscr{U}^i_{\mathscr{M}_{X\not\to D}}(\pi)$.

VoI has been applied to a wide array of problems in economics and computer science [BP16]. Although *PyCID*'s function *quantitatative_voi* returns the quantitative VoI of a variable in a CIM, for the remainder of this section we shall focus on its graphical criterion, which depends upon which nodes are **requisite** observations in the CID.

Requisite Observation Graphical Criterion: Let $\mathbf{U}_D \in \mathbf{U} \cap \mathbf{Desc}_D$ be the utility nodes downstream of D. An observation $X \in \mathbf{Pa}_D$ in a single-decision CID is **requisite** if $X \not\perp_{\mathscr{G}} \mathbf{U}_D | (\mathbf{Pa}_D \cup \{D\} \setminus \{X\})$[5].

VoI Graphical Criterion: A single decision CID, $\mathscr{G}$, admits **VoI** for $X \in \mathbf{V} \setminus \mathbf{Desc}_D$ if and only if X is a requisite observation in $\mathscr{G}_{X\to D}$, the graph obtained by adding $X \to D$ to $\mathscr{G}$.

To demonstrate how to find nodes that admit VoI using *PyCID*, we extend the **Grade Prediction** example given in the introduction:

Extended Grade Prediction: [ECL+21] The university wants to admit the brightest students using their grade prediction model, but doesn't want to treat students differently based on their gender (Ge) or race (R). The model uses the gender of the student and the high school (HS) they attended to make its grade prediction. We make the following assumptions:

- Performance at university is evaluated by a student's grades (Gr) and this depends on the quality of education (E) the student received before university (which depends on the high school they attended).
- A student's high school is assumed to be impacted by their race, but not by their gender.

We want to know whether the predictor is incentivised to behave in a discriminatory manner with respect to the students' gender or race. A CID for this example is defined below:

```
cid = pycid.CID(
   [
      ("R", "HS"),
      ("HS", "E"),
      ("HS", "P"),
      ("E", "Gr"),
      ("Gr", "Ac"),
      ("Ge", "P"),
      ("P", "Ac"),
   ],
   decisions=["P"],
   utilities=["Ac"],
)
```

PyCID finds that HS, E, and Gr can all have positive VoI for the predictor model (line 1). We can also display this visually (Figure 4) by passing, as an argument, a lambda function into `CID`'s *draw_property* method (line 2):

```
1    pycid.admits_voi_list(cid, 'P')
2    cid.draw_property(lambda node:
3            pycid.admits_voi(cid, 'P', node))
```

Our implementation of this example in *PyCID* has revealed that there exists a parameterisation of this setup (i.e., a CIM with the given CID) where the model would benefit from knowing the value of one or more of 'High School', 'Education', or the student's true 'Grade' before making a grade prediction.

Response Incentives (RI)

Response incentives (RI) are a related type of incentive and we explain how implementing them in *PyCID* can help improve the

3. Nodes can be specified further as admitting indirect or direct Value of Control.

4. This definition is also valid in (non-causal) statistical influence models.

5. $X \not\perp_{\mathscr{G}} Y \mid \mathbf{W}$ denotes that X is d-connected to Y conditional on the set of nodes in $\mathbf{W}$ and $X \perp_{\mathscr{G}} Y \mid \mathbf{W}$ would denote that X is d-separated from Y conditional on $\mathbf{W}$ [Pea09].

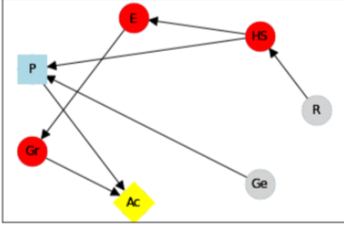

Fig. 4: A CID for the ***Extended Grade Prediction*** *example with the variables that admit VoI in a darker colour, red (plotted using* PyCID*).*

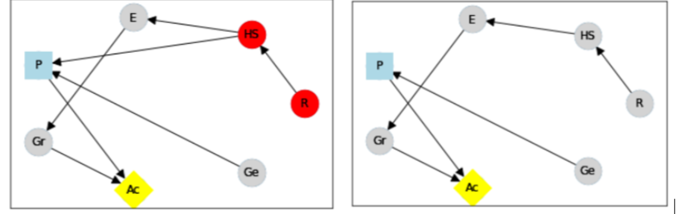

Fig. 5: (Left) The original CID for the ***Extended Grade Prediction*** *example with the variables that admit an RI in a darker colour, red, and (Right) the modified CID in which now no node admits an RI (plotted using* PyCID*).*

fairness of AI systems. A variable admits an (RI) if a decision-maker benefits from making its decision causally responsive to the variable [ECL+21][6].

RI Graphical Criterion: A single decision CID, $\mathcal{G}$, admits a **response incentive** on $X \in \mathbf{X}$ if and only if there is a directed path $X \dashrightarrow D$ in the requisite graph[7] $\mathcal{G}_{req}$ where $\mathcal{G}_{req}$ is the result of removing from $\mathcal{G}$ all information links from non-requisite observations.

To demonstrate how to find the nodes which admit RIs, we will again consider the **Extended Grade Prediction** example. As we did with VoI, we can list all of the nodes that admit RIs in the CID (line 1) or we can display the result visually (line 2) with the result shown in Figure 5 (Left):

```
1    pycid.admits_ri_list(cid, 'P')
2    cid.draw_property(lambda node:
3            pycid.admits_ri(cid, 'P', node))
```

Implementing CIDs in *PyCID* can help suggest how to improve the fairness of AI systems because [ECL+21] argue that an RI on a sensitive attribute can be interpreted as problematic from a fairness perspective. A decision is considered counterfactually unfair if a change to a sensitive attribute, such as race or gender, would change the decision [KLRS17]. Therefore, an RI on a sensitive attribute indicates that counterfactual unfairness is incentivised; specifically, it implies that all optimal policies are counterfactually unfair. To mitigate this, [ECL+21] propose redesigning the grade-predictor. By removing the predictor's access to knowledge about the student's high school (i.e., the edge $HS \rightarrow P$), there will no longer be an RI on a sensitive attribute. The following code trims the edge and shows that now no node admits an RI in the modified CID (Figure 5 (Right)):

```
cid.remove_edge('HS', 'P')
cid.draw_property(lambda node: \
      pycid.admits_ri(cid, 'P', node))
```

Value of Control (VoC) and Instrumental Control Incentives (ICI)

We now turn to Value of Control (VoC) and Instrumental Control Incentives (ICI) and show that implementing the latter in *PyCID* can help design safer AI systems. Intuitively, a variable has *positive value of control (VoC)* if a decision-maker could benefit from choosing that variable's value.

VoC Definition: For a CIM $\mathcal{M}$, the **value of control** for a non-decision node $X \in \mathbf{V} \setminus \mathbf{D}$ is $\max_{\pi} \max_{g^X} \mathcal{U}^i_{\mathcal{M}_{g^X}}(\pi) - \max_{\pi} \mathcal{U}^i_{\mathcal{M}}(\pi)$. $\mathcal{M}_{g^X}$ denotes the CIM $\mathcal{M}$ after intervening on X with any CPD, g^X, that respects the graph.

[6]. For a formal definition, we refer the reader to [ECL+21].

[7]. A requisite graph is also known as a minimal reduction, trimmed_graph, or d-reduction.

VoC Graphical Criterion: A single decision CID, $\mathcal{G}$, admits **positive value of control** for a node $X \in \mathbf{V} \setminus \{D\}$ if and only if there is a directed path $X \dashrightarrow U$ in the requisite graph $\mathcal{G}_{req}$.

Although VoC is a useful concept, it does not consider whether it is actually possible for an agent to control that variable. Therefore, [ECL+21] introduce Instrumental Control Incentives, which can be intuitively understood as follows: if the agent got to choose D to influence X independently of how D influences other aspects of the environment, would that choice matter? In other words, is controlling X instrumentally useful for maximising utility? The graphical criteria for ICI in a single-decision CID is:

ICI Graphical Criterion: A single decision CID, $\mathcal{G}$, admits an **instrumental control incentive** on $X \in \mathbf{V}$ if and only if $\mathcal{G}$ has a directed path from the decision D to a utility node $U \in \mathbf{U}$ that passes through X.

To demonstrate how to find these incentives in *PyCID*, we introduce another example from [ECL+21].

Content recommendation: An AI algorithm has the task of choosing posts (P) to show a user, to maximise the user's click rate (C). The designers want the algorithm to present content adapted to each user's original opinions (O) to optimize clicks; the algorithm does not know the user's true original opinions, so it instead relies on an approximate model (M). However, the designers are worried that the algorithm will use polarising content to influence user opinions (I) so that the user clicks more predictably:

```
cid = pycid.CID(
    [
        ("O", "M"),
        ("O", "I"),
        ("M", "P"),
        ("P", "I"),
        ("I", "C"),
        ("P", "C"),
    ],
    decisions=["P"],
    utilities=["C"],
)

cid.draw_property(lambda node: \
      pycid.admits_ici(cid, 'P', node))
```

With RI, we showed that implementing CIDs in *PyCID* can aid the design of fairer systems; with ICI, we demonstrate how *PyCID* can be used to help design safer AI systems. First, we can use analogous functions to what we used for VoI and RI - *pycid.admits_voc_list(cid)* and *pycid.admits_ici_list(cid, 'P')* - to find that O, M, I, and C can have positive VoC whilst I, P, and C admit ICI. From this, because I (influenced user opinions) admits an instrumental control incentive, we discover that the content recommender may seek to influence that variable to attain utility.

[ECL+21] offer an alternative content recommender design that avoids this undesirable behaviour. Instead of being rewarded for the true click-through rate, the content recommender is rewarded for the clicks it would be predicted to have, based on a separately trained model of the user's preferences. The modified CID for this changed model is shown in Figure 6 c) where the old utility node *C* (actual clicks) has become *PC* (predicted clicks):

```
cid = pycid.CID(
    [
        ("O", "M"),
        ("O", "I"),
        ("M", "P"),
        ("M", "PC"),
        ("P", "I"),
        ("P", "PC"),
    ],
    decisions=["P"],
    utilities=["PC"],
)

cid.draw_property(lambda node: \
        pycid.admits_ici(cid, 'P', node))
```

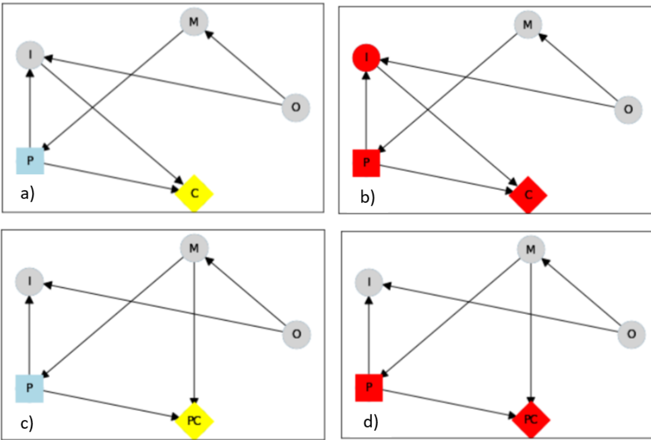

Fig. 6: The original CID for the ***Content recommendation*** *example in (a) with (b) the variables that admit ICI in a darker colour, red, and (c) the modified content recommender's CID in which (d) I no longer admits an ICI (plotted using* PyCID*).*

Multi-agent (Causal) Influence Diagrams

In this section, we will show how to instantiate MA(C)IDs/MA(C)IMs in *PyCID* and demonstrate a selection of methods for analysing games (strategic interactions between self-interested players) including strategic relevance [KM03] and finding Nash equilibria (NE) [N+50].

Recall from the *Background* section that a Multi-agent Causal Influence Diagram/Model (MACID/MACIM) is a simple multi-agent extension of a CID/CIM [HFE+]. For our purpose, all that's important is that there is now a set of $\mathbf{N}$ agents and so the decision and utility nodes are partitioned into $\{\mathbf{D}^i\}_{i\in\mathbf{N}}$ and $\{\mathbf{U}^i\}_{i\in\mathbf{N}}$ to correspond to their association with a particular agent $i\in\mathbf{N}$. We also again underline that the only difference between statistical multi-agent influence diagrams/models (MAIDs/MAIMs) and MACIDs/MACIMs is that the edges represent every causal relationship between the random variables chosen to be endogenous variables in the model, as opposed to just associational relationships. Nevertheless, because MACIDs subsume MAIDs (in the sense of Pearl's *causal hierarchy* [Pea09]), everything we can do in a MAID, we can also do in a MACID. Therefore, for the two examples we present here, MAIDs and MACIDs can be viewed as the same.

To serve as our example, we shall use the Prisoner's Dilemma, which is probably the best known simultaneous and symmetric two-player game:

Prisoner's Dilemma: Two prisoners, suspected of committing a robbery together, are isolated and urged to confess. Each is concerned only with getting the shortest possible prison sentence for himself and must decide whether to confess without knowing his partner's decision. Both prisoners, however, know the consequences of their decisions. Each year spent in prison can be represented as -1 utility and so the payoff matrix for this game (or Normal form) is given in Figure 7.

		D^2	
		cooperate	defect
D^1	cooperate	-1,-1	-3,0
	defect	0,-3	-2,-2

Fig. 7: Normal form game giving the payoffs for each player in the ***Prisoner's Dilemma****. Player 1 (2) is the row (column) player.*

MA(C)IDs and MA(C)IMs are instantiated as `MACID` objects with identical syntax to `CID` objects except for there being multiple agents and so we can draw them in the same way. Figure 8 (Left) shows that in *PyCID*, consistent with (C)IDs, decision nodes are drawn as rectangles and utility nodes are drawn as diamonds; however, because we now have more than one player, we reserve colouring to denote agent membership: each agent is assigned a unique colour. Chance nodes remain as grey circle (Figure 11):

```
macid = pycid.MACID(
    [
        ("D1", "U1"),
        ("D1", "U2"),
        ("D2", "U1"),
        ("D2", "U2"),
    ],
    # specifies each agent's decision and utility nodes.
    agent_decisions={1: ['D1'], 2: ['D2']},
    agent_utilities={1: ['U1'], 2: ['U2']},
)

d1_dom = ['c', 'd']
d2_dom = ['c', 'd']

agent1_payoff = np.array([[-1, -3], [0, -2]])
agent2_payoff = np.transpose(agent1_payoff)

macid.add_cpds(
    D1=d1_dom,
    D2=d2_dom,
    U1=lambda d1, d2: agent1_payoff[d1_dom.index(d1),
                                    d2_dom.index(d2)],
    U2=lambda d1, d2: agent2_payoff[d1_dom.index(d1),
                                    d2_dom.index(d2)]
)
macid.draw()
```

The following command tells us that the second player (agent) receives expected utility = -3 (i.e., they will spend 3 years in prison) given that player 1 decides to defect and player 2 decides to cooperate. This agrees with the payoff matrix in Figure 7:

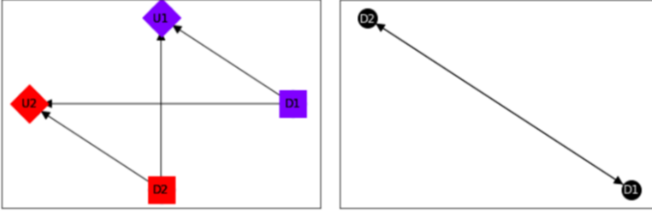

*Fig. 8: A MACID for the **Prisoner's Dilemma** (Left) and its corresponding relevance graph (Right) (plotted using* PyCID*).*

```
macid.expected_utility(context={'D1':'d', 'D2': 'c'},
                                agent=2)
```

Strategic relevance is a useful concept for analysing decisions made in games; it asks which other decisions' decision rules need to be already be known before we can optimise a particular decision rule. [KM03] introduced the graphical criterion *s-reachability* for determining this from the graph:

S-reachability Graphical Criterion: Another decision node D' is **s-reachable** from a decision $D \in \mathbf{D}^i$ in a MA(C)ID, $\mathscr{M} = (\mathbf{N}, \mathbf{V}, \mathbf{E})$, if a newly added parent $\hat{D}'$ of D' satisfies $\hat{D}' \not\perp_{\mathscr{G}} \mathbf{U}^i \cap \mathbf{Desc}_D \mid \mathbf{Fa}_D$.

Using *PyCID*, lines 1 and 2 below evaluate to *True*, which tells us that each decision strategically relies on the other; each prisoner would be better off knowing the other prisoner's policy before deciding on their own action. To show this visually, line 3 plots the MACID's relevance graph [KM03] (Figure 8 Right):

```
1   macid.is_r_reachable('D1', 'D2')
2   macid.is_r_reachable('D2', 'D1')
3   pycid.RelevanceGraph(macid).draw()
```

We now turn to finding NE in games. We use $\pi_{\mathbf{A}}$ to denote player i's set of decision rules for decisions $\mathbf{A} \subseteq \mathbf{D}^i$, given a partial policy profile $\pi_{-\mathbf{A}}$ over all of the other decision nodes in a MA(C)ID, $\mathscr{M}$. We write $\mathscr{U}^i_{\mathscr{M}}(\pi_{\mathbf{A}}, \pi_{\text{-}\mathbf{A}})$ to denote the expected utility for player i under the policy profile $\pi = (\pi_{\mathbf{A}}, \pi_{\text{-}\mathbf{A}})$.

Definition: [KM03] A full policy profile π is a **Nash equilibrium (NE)** in a MA(C)IM $\mathscr{M}$ if, for every player $i \in \mathbf{N}$, $\mathscr{U}^i_{\mathscr{M}}(\pi^i, \pi^{-i}) \geq \mathscr{U}^i_{\mathscr{M}}(\hat{\pi}^i, \pi^{-i})$ for all $\hat{\pi}^i \in \Pi^i$.

To find all pure NE in the MA(C)IM corresponding to the **Prisoner's Dilemma**:

```
macid.get_all_pure_ne()
```

This method returns a list of all pure NE in the MA(C)ID. Each NE comes as a list of `StochasticFunctionCPD` objects, one for each decision node in the MA(C)ID:

```
[[StochasticFunctionCPD<D1>
   {}  ->  d,
StochasticFunctionCPD<D2>
   {}  ->  d]]
```

In the **Prisoner's Dilemma**, there is only one NE and this involves both players defecting. We can then find that the expected utility for each agent is -2 under this NE joint policy profile:

```
all_pure_ne = macid.get_all_pure_ne()
macid.add_cpds(*all_pure_ne[0])
macid.expected_utility({}, agent=1)
macid.expected_utility({}, agent=2)
```

PyCID can also be used to find subgame perfect equilibria (SPE) [Sel65]. A SPE is a NE where no player makes a *non-credible threat* - an action that, if the player is rational, they would never actually carry out.

Definition: [HFE+21] A full policy profile π is a **subgame perfect equilibrium (SPE)** in a MA(C)IM $\mathscr{M}$ if π is an NE in every MAIM subgame[8] of $\mathscr{M}$.

The **Prisoner's Dilemma** MAIM has no proper MAIM subgames and so the NE we found above is (trivially) also a SPE. Therefore, to demonstrate how *PyCID* distinguishes between NE and SPE, we use the following example:

Taxi Competition: Two autonomous taxis, operated by different companies, are driving along a road with two hotels located next to one another - one expensive and one cheap. Each taxi must decide (one first, then the other) which hotel to stop in front of, knowing that it will likely receive a higher tip from guests of the expensive hotel. However, if both taxis choose the same location, this will reduce each taxi's chance of being chosen by that hotel's guests. The payoffs for each player are shown in Figure 9 and the MACIM for this example is instantiated in *PyCID* below

		D^2	
	U^1	expensive	cheap
D^1	expensive	2	5
	cheap	3	1

		D^2	
	U^2	expensive	cheap
D^1	expensive	2	5
	cheap	3	1

*Fig. 9: Payoff matrices for taxi 1 (left) and taxi 2 (right) for the **Taxi Competition**.*

```
macid = MACID(
   [("D1", "D2"), ("D1", "U1"), ("D1", "U2"),
   ("D2", "U2"), ("D2", "U1")],
   agent_decisions={1: ["D1"], 2: ["D2"]},
   agent_utilities={1: ["U1"], 2: ["U2"]},
)

d1_dom = ["e", "c"]
d2_dom = ["e", "c"]
agent1_payoff = np.array([[2, 5], [3, 1]])
agent2_payoff = agent1_payoff.T

macid.add_cpds(
   D1=d1_dom,
   D2=d2_dom,
   U1=lambda d1, d2: agent1_payoff[d1_dom.index(d1),
                                   d2_dom.index(d2)],
   U2=lambda d1, d2: agent2_payoff[d1_dom.index(d1),
                                   d2_dom.index(d2)],
)
```

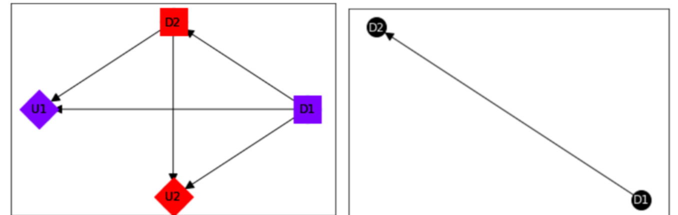

*Fig. 10: A MA(C)ID for the **Taxi Competition** and its corresponding relevance graph (plotted using* PyCID*).*

This MA(C)IM has three pure NE, which are found using *macid.get_all_pure_ne()*. We can also find the decision nodes in

8. We refer the interested reader to [HFE+21] for a definition of a MAIM subgame.

each MAID subgame (see [HFE+21]), the decision nodes that can be optimised independently from the rest:

```
macid.decs_in_each_maid_subgame()
[{'D2'}, {'D1', 'D2'}]
```

We can find the NE in the only proper subgame:

```
macid.get_all_pure_ne_in_sg(decisions_in_sg=['D2'])
```

and finally all SPE in the MA(C)IM. The **Taxi Competition**'s MACIM has only one pure SPE:

```
macid.get_all_pure_spe()

[[StochasticFunctionCPD<D2>
   {'d1': 'c'}  ->  e
   {'d1': 'e'}  ->  c,
StochasticFunctionCPD<D1>
   {}  ->  e]]
```

Random (C)IDs and MA(C)IDs

PyCID has other features that can be useful for researchers. In particular, the library contains functions for instantiating random (MA)(C)IDs. This is useful for estimating the average properties of graphs, or for finding a counterexample to some conjecture. The first example below finds and plots a random 10-node, single-agent (C)ID with two decision nodes and three utility nodes. The second example finds and plots a random 12-node MA(C)ID with two agents. The first agent has one decision and two utility nodes, the second agent has three decisions and two utility nodes. In both these examples, we set the *add_cpds* flag to *False* to create non-parameterised (MA)(C)IDs. If one sets this flag to *True*, each chance and utility node is assigned a random CPD, and each decision node a domain to instantiate a (MA)CIM. One can also force every agent in the (MA)(C)ID to have sufficient recall; an agent has sufficient recall if the relevance graph restricted to include just that agent's decision nodes is acyclic. This can be useful for certain incentives analyses [vMCE]. The *edge_density* and *max_in_degree* parameters set the density of edges in the (MA)(C)ID's DAG as a proportion of the maximum possible number ($n \times (n-1)/2$) and the maximum number of edges incident to a node in the DAG. To find a (MA)(C)ID that meets all of the specified constraints, *PyCID* uses rejection sampling and so *max_resampling_attempts* specifies the number of samples to try before timing out:

```
cid = pycid.random_cid(
   number_of_nodes=10,
   number_of_decisions=2,
   number_of_utilities=3,
   add_cpds=False,
   sufficient_recall=False,
   edge_density=0.4,
   max_in_degree=5,
   max_resampling_attempts=100,
)
cid.draw()

macid = pycid.random_macid(
   number_of_nodes=12,
   agent_decisions_num=(1, 3),
   agent_utilities_num=(2, 2),
   add_cpds=False,
   sufficient_recall=False,
   edge_density=0.4,
   max_in_degree=5,
   max_resampling_attempts=500,
)
macid.draw()
```

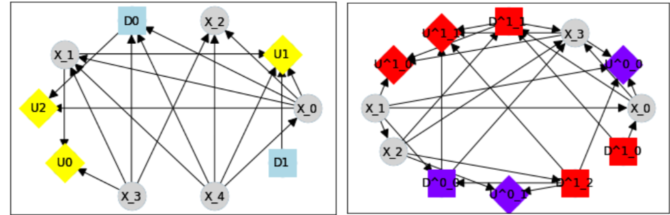

Fig. 11: *A random (C)ID and MA(C)ID created in* PyCID.

Conclusions and Future Directions

PyCID is a Python library for solving and analysing single-agent and multi-agent (causal) influence diagrams. Several key classes - `CausalBayesianNetwork`, `CID`, and `MACID` - enable decision problems to be solved and the effects of causal interventions to be studied whilst *PyCID*'s analysis functions can find graphical properties such as incentives in CIDs and reasoning patterns in MACIDs. This makes *PyCID* a customizable, but powerful library for testing research ideas and exploring applications. Moreover, implementing examples programmatically can substantiate the claims made by ID researchers about the benefit of their work; one can assess how different quantities vary over the parameter space or empirically verify complexity results [HFE+]. Single-agent and multi-agent (causal) influence diagrams are an area of active research, so as theory develops, the *PyCID* library will also grow. Extensions will likely include:

- Support for finding incentives in multi-decision CIDs [vMCE].
- Support for Structural Causal Models [Pea09] and therefore also quantitative RI and ICI.
- More game-theoretic concepts (e.g. more equilibrium concepts).
- Support for multi-agent incentives.

In this paper, we have demonstrated the usefulness of *PyCID* by focusing on causal influence diagrams; however, this library is also well suited for working with statistical influence diagrams. The development team would like to invite researchers from any domain to use *PyCID* to test the package for diverse applications, to contribute new methods and functions, and to join our Causal Incentives Working Group: https://causalincentives.com/. The *PyCID* repository is available on GitHub under our working group's organization: https://github.com/causalincentives/pycid.

Acknowledgements

Fox acknowledges the support of the EPSRC Centre for Doctoral Training in Autonomous Intelligent Machines and Systems (Reference: EP/S024050/1).

References

[AP15] Ankur Ankan and Abinash Panda. pgmpy: Probabilistic Graphical Models using Python. In Kathryn Huff and James Bergstra, editors, *Proceedings of the 14th Python in Science Conference*, pages 6–11, 2015. doi:10.25080/Majora-7b98e3ed-001.

[Bay] BayesFusionLLC. SMILE: Structural Modeling, Inference, and Learning Engine. URL: https://www.bayesfusion.com/smile.

[BP16] Emanuele Borgonovo and Elmar Plischke. Sensitivity analysis: a review of recent advances. *European Journal of Operational Research*, 248(3):869–887, 2016. doi:10.1016/j.ejor.2015.06.032.

[CLEL20] Ryan Carey, Eric Langlois, Tom Everitt, and Shane Legg. The incentives that shape behaviour. *arXiv preprint arXiv:2001.07118*, 2020.

[CVH20] Michael Cohen, Badri Vellambi, and Marcus Hutter. Asymptotically unambitious artificial general intelligence. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 2467–2476, 2020. doi:10.1609/aaai.v34i03.5628.

[DBDSMW20] Gaspard Ducamp, Philippe Bonnard, Christian De Sainte Marie, and Pierre-Henri Wuillemin. aGrUM/pyAgrum : a Toolbox to Build Models and Algorithms for Probabilistic Graphical Models in Python. In *10th International Conference on Probabilistic Graphical Models*, volume 138 of *Proceedings of Machine Learning Research*, pages 173–184, Skørping, Denmark, 2020. URL: https://hal.archives-ouvertes.fr/hal-02911619.

[ECL+21] Tom Everitt, Ryan Carey, Eric Langlois, Pedro Ortega, and Shane Legg. Agent incentives: A causal perspective. In *Proceedings of the Thirty-Fifth AAAI Conference on Artificial Intelligence,(AAAI-21). Virtual. Forthcoming*, 2021.

[EHKK21] Tom Everitt, Marcus Hutter, Ramana Kumar, and Victoria Krakovna. Reward tampering problems and solutions in reinforcement learning: A causal influence diagram perspective. *Synthese*, pages 1–33, 2021. doi:10.1007/s11229-021-03141-4.

[EKKL19] Tom Everitt, Ramana Kumar, Victoria Krakovna, and Shane Legg. Modeling AGI safety frameworks with causal influence diagrams. In *IJCAI AI Safety Workshop*, 2019. arXiv:1906.08663.

[Góm04] Manuel Gómez. Real-world applications of influence diagrams. In *Advances in Bayesian networks*, pages 161–180. Springer, 2004. doi:10.1007/978-3-540-39879-0_9.

[HFE+] Lewis Hammond, James Fox, Tom Everitt, Ryan Carey, Alessandro Abate, and Michael Wooldridge. Causality in games. *Forthcoming*.

[HFE+21] Lewis Hammond, James Fox, Tom Everitt, Alessandro Abate, and Michael Wooldridge. Equilibrium refinements for multi-agent influence diagrams: Theory and practice. In *Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems*, pages 574–582, 2021.

[HM05] Ronald A Howard and James E Matheson. Influence diagrams. *Decision Analysis*, 2(3):127–143, 2005. doi:10.1287/deca.1050.0020.

[HMvdW+20] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020. doi:10.1038/s41586-020-2649-2.

[Hol20] Koen Holtman. Towards AGI Agent Safety by Iteratively Improving the Utility Function. In Ben Goertzel, Aleksandr I Panov, Alexey Potapov, and Roman Yampolskiy, editors, *Artificial General Intelligence*, pages 205–215, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-52152-3_21.

[How66] Ronald A Howard. Information value theory. *IEEE Transactions on systems science and cybernetics*, 2(1):22–26, 1966.

[HSS08] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[Hun07] John D Hunter. Matplotlib: A 2D graphics environment. *IEEE Annals of the History of Computing*, 9(03):90–95, 2007. doi:10.1109/MCSE.2007.55.

[KLRS17] Matt Kusner, Joshua Loftus, Chris Russell, and Ricardo Silva. Counterfactual fairness. In *Advances in Neural Information Processing Systems*, volume 2017-Decem, pages 4067–4077. Neural information processing systems foundation, 2017. URL: http://arxiv.org/abs/1703.06856, arXiv:1703.06856.

[KM03] Daphne Koller and Brian Milch. Multi-agent influence diagrams for representing and solving games. *Games and economic behavior*, 45(1):181–221, 2003. doi:10.1016/s0899-8256(02)00544-4.

[KM08] Uffe B Kjaerulff and Anders L Madsen. Bayesian networks and influence diagrams. *Springer Science+ Business Media*, 200:114, 2008.

[LE21] Eric Langlois and Tom Everitt. How RL agents behave when their actions are modified. In *AAAI*, 2021. arXiv:2102.07716.

[MIMH+76] Allen C Miller III, Miley W Merkhofer, Ronald A Howard, James E Matheson, and Thomas R Rice. Development of automated aids for decision analysis. Technical report, STANFORD RESEARCH INST MENLO PARK CA, 1976. doi:10.21236/ada026379.

[N+50] John F Nash et al. Equilibrium points in n-person games. *Proceedings of the national academy of sciences*, 36(1):48–49, 1950. doi:10.2307/j.ctv173f1fh.6.

[Pea09] Judea Pearl. *Causality*. Cambridge university press, 2009.

[PG07] Avi Pfeffer and Yakov Gal. On the reasoning patterns of agents in games. In *Proceedings of the 22nd national conference on Artificial intelligence - Volume 1*, pages 102–109. AAAI Press, 2007. 22nd national conference on Artificial intelligence, AAAI-07 ; Conference date: 22-07-2007 Through 26-07-2007. URL: https://www.aaai.org/Conferences/AAAI/aaai07.php.

[Sel65] Reinhard Selten. Spieltheoretische behandlung eines oligopolmodells mit nachfrageträgheit: Teil i: Bestimmung des dynamischen preisgleichgewichts. *Zeitschrift für die gesamte Staatswissenschaft/Journal of Institutional and Theoretical Economics*, (H. 2):301–324, 1965.

[Sha86] Ross D Shachter. Evaluating influence diagrams. *Operations research*, 34(6):871–882, 1986.

[vMCE] Chris van Merwijk, Ryan Carey, and Tom Everitt. Techniques for analysing multi-decision influence diagrams. *Forthcoming*.