Decisions.jl: Representing and Transforming Decision Problem Classes in Julia

Mel Krusniak
Vanderbilt University
Nashville, TN, United States
mel.krusniak@vanderbilt.edu

Ofer Dagan
University of Colorado Boulder
Boulder, CO, United States
ofer.dagan@colorado.edu

Himanshu Gupta
University of Colorado Boulder
Boulder, CO, United States
himanshu.gupta@colorado.edu

Benjamin Kraske
University of Colorado Boulder
Boulder, CO, United States
benjamin.kraske@colorado.edu

Kyle Hollins Wray
Northeastern University
Boston, MA, United States
k.wray@northeastern.edu

Zachary Sunberg
University of Colorado Boulder
Boulder, CO, United States
zachary.sunberg@colorado.edu

ABSTRACT

Decisions.jl is an open-source Julia ecosystem for standardized representations of general sequential decision problems. Uniquely, it explicitly represents the dynamic decision networks that underlie problem classes, and therefore supports everything from the most basic Markovian problems to problems with intricate variations on multiagency, observability, constraints, and continuity (among other characteristics). By leveraging the Julia language's just-in-time compilation, Decisions.jl delivers high-performance interfaces in the intuitive style of similar single-model frameworks. In this work, we explore how Decisions.jl can be used to navigate the space of sequential decision problem classes. Decisions.jl enables the transformation of problem formulations into simpler or more complex variants, allowing users to rigorously evaluate tradeoffs between generality and performance in decision-making algorithms. Additionally, Decisions.jl streamlines the usage of multiagent problem formulations that would otherwise be too cumbersome to replicate consistently.

KEYWORDS

Decision making under uncertainty; sequential decision making; dynamic decision networks; middleware

ACM Reference Format:
Mel Krusniak, Ofer Dagan, Himanshu Gupta, Benjamin Kraske, Kyle Hollins Wray, and Zachary Sunberg. 2026. Decisions.jl: Representing and Transforming Decision Problem Classes in Julia. In *Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2026), Paphos, Cyprus, May 25 – 29, 2026*, IFAAMAS, 8 pages. https://doi.org/10.65109/XRMX3337

1 INTRODUCTION

When modeling sequential decision problems, decision theorists and practitioners often begin by identifying key characteristics like multiagency, partial observability, decentralization, and then use software tailored to the corresponding formulation. But writing decision-making algorithms, drawing comparisons, and extending domains often require interacting with *multiple* such formulations.

For example, imagine, having developed a problem as a partially observable Markov game (POMG), attempting to (a) draw comparisons with decentralized, cooperative variants; (b) impose constraints, or (c) use belief-based operations from the single-agent literature. Currently, any one of these tasks would require manually altering the POMG software architecture to support a new problem class - a time-consuming, often unrigorous process. However, many decision problems can be represented using dynamic decision networks (DDNs), and the extension relationships between problems can be standardized as operations on those networks. Using such a standardization, we can generalize and automate away the tedious task of modifying problem frameworks.

Decisions.jl is an open-source sequential decision modeling ecosystem that represents *all* decision problems that can be expressed with a DDN. It represents DDNs explicitly at runtime, compiled just-in-time to provide first-class support even for nonstandard problem classes. As an example, Decisions.jl supports the problem class represented by Figure 1 - a centralized, partially observable, multi-agent game with unusual dependencies between agents on each node. No existing single sequential decision-making framework can represent this problem class out of the box. (We discuss this particular class further in Section 2.5.)

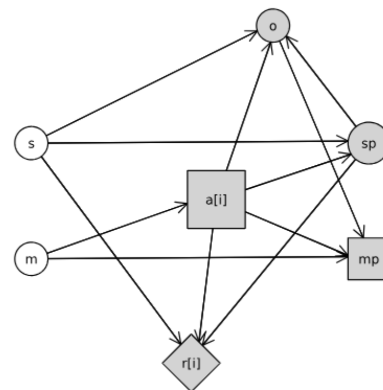

Figure 1: The decision network underlying centralized, partially observable, multi-agent games - one of many unusual problem classes supported by Decisions.jl.

This work is licensed under a Creative Commons Attribution International 4.0 License.

Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2026), C. Amato, L. Dennis, V. Mascardi, J. Thangarajah (eds.), May 25 – 29, 2026, Paphos, Cyprus. © 2026 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). https://doi.org/10.65109/XRMX3337

Furthermore, by specifying the right sequence of transformations, any problem in a particular problem class can be converted into a different problem class. In doing so, users can carry out many tedious framework alterations in one line. We discuss these transformations in Section 3.1.

Decisions.jl is primarily designed for model-aware settings. It is neither a solver library nor a collection of environments, though some simple domains and algorithms are provided for prototyping. It is implemented in the Julia programming language [2].

In this paper, we focus on two core functionalities of Decisions.jl:

(1) Since Decisions.jl supports any problem classes backed by a dynamic decision network, arbitrary problem classes can be represented. This includes standard problem classes (like Markov decision processes (MDPs), partially observable Markov decision processes (POMDPs), and Markov games (MGs)) as well as problems with novel definitions or with unusual combinations of characteristics.
(2) Decisions.jl also provides a number of *transformations* between problem classes, adding or removing problem components in a rigorous manner to enable different solver techniques and improve rapid model prototyping.

After an overview of related work, we dedicate a section to each of these functionalities, followed by a brief concluding discussion.

1.1 Related Software

Decisions.jl is primarily designed with model-aware decision theory in mind. For the purposes of this section we ignore the many model-free reinforcement learning frameworks available.

The Julia language [2] has active model-aware sequential decision making and mathematical optimization communities, so we point out some direct inspirations:

- The POMDPs.jl ecosystem [3] "provides a common programming vocabulary" for POMDPs to facilitate solver and environment development. But POMDPs.jl only supports POMDPs (and closely related problems) — specifically, there is no multiagent support, which is first-class in Decisions.jl.
- DecisionProgramming.jl [13] takes a similar approach to Decisions.jl in representing a broad array of problems using underlying decision networks, but primarily for the purpose of transforming decision networks into mixed-integer linear programs. Decisions.jl is more involved with the problems these decision networks represent, and allows for both continuous and discrete spaces.

Of course, the majority of sequential decision-making frameworks exist outside Julia, many oriented towards reinforcement learning and others towards specific decisions and operations research. Among these, we note two in particular. First, Openspiel [10] is widely used for reinforcement learning on extensive form games (EFGs). Unfortunately, the gap between EFGs and more standard sequential decision making problems makes it difficult to apply algorithms from one formulation to the other [1]. Decisions.jl is designed to allow for the extension of single-agent decision making into the multiagent space.

Second, the PRISM model checker [8] works on a set of decision problems comparable to that of Decisions.jl, particularly through PRISM-Games [9], which allows modeling stochastic games. However, while some of the formal modeling is similar, PRISM and Decisions.jl come from very different traditions. PRISM is primarily a software verification tool (though strategy synthesis is available). Decisions.jl is only for decision making, but is more involved in manipulating abstract formulation of sequential decision problems. (PRISM does not manipulate DDNs explicitly.)

1.2 Related Theory

Dynamic decision networks (DDNs) are the core of Decisions.jl. We define them in Section 2.1, and typically follow the patterns described by Kochenderfer et al. [6] in using them to represent problems.

Work which involves manipulating DDNs to solve the represented problems focuses heavily on Markov decision processes (MDPs). Of this, the work of Venturato et al. [15] is of some relevance for its focus on compiling somewhat general DDNs into more useful forms (in that case, into propositional logic formulae, among other steps). This broadly follows the compile-evaluate paradigm Decisions.jl uses to generate problem utilities, but to a different end (and using only boolean random variables). In the multiagent space, decentralized partially observable MDPs (Dec-POMDPs) appear to be the most general problem class formulated with DDNs found commonly in the literature. In particular, Kumar et al. [7] manipulate the DDN for Dec-POMDPs to improve scalability in a way well aligned with Decisions.jl transformations.

Decisions.jl is also inspired by the theory of game abstraction [14]. Game abstraction involves mapping a complicated game to a simpler space, solving it there, and mapping the solution back to the original space. Decisions.jl takes this a step farther, allowing for simplifications to problem classes outside of game-theoretic formulation entirely. What we refer to here as "problem class transformations" are a generalization of the "game view operations" discussed by Wellman and Mayo [16]. In particular, the implementation of fictitious play via transformations given in Section 3.3 is directly based upon their discussion of iterative best response via game views.

2 GENERATING PROBLEM FRAMEWORKS

In this section, we describe how Decisions.jl represents general decision problem classes.

2.1 Decision Networks

We begin with some formal definitions of the structures used in Decisions.jl.

A **decision network** (DN) is a Bayesian network with finite, labeled nodes V, edges E, and conditional probability distributions P, with the following modifications:

- Each node is a *decision* (or *action*) node, a *chance* node, or an *outcome* (or *utility*) node, with the following restrictions:
  - *Decision* nodes' conditional distributions have no defined PDF or sampling function. They may, however, have a well-defined support.
  - *Outcome* nodes have no children.
  - Nodes which are not *outcome* or *decision* nodes are said to be *chance* nodes.

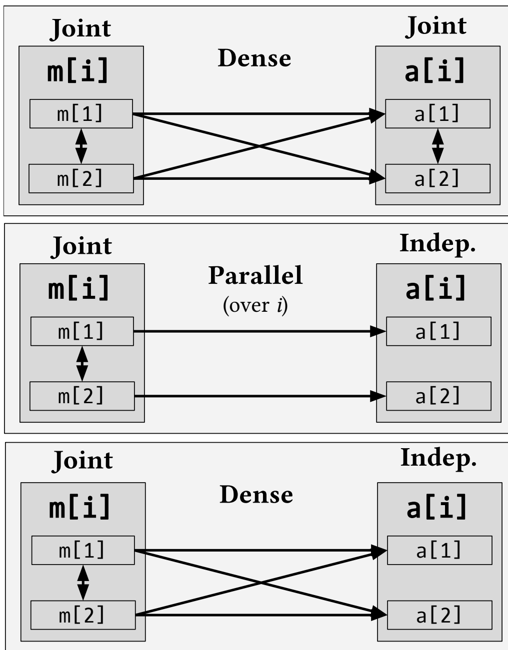

Figure 2: Three possible relationships between and within two indexed nodes (m and a) in a DDN.

- A decision network may be *dynamic*: representing a single time step in an iterative process, where specified nodes are *temporally linked*. For temporally linked nodes a, a', the output of node a at iterate t is the output of node a' at iterate $t-1$. Temporal links are specified with the *temporal mapping* $s : V_t \to V_{t+1}$, where $V_t, V_{t+1} \subset V$ and $V_t \cap V_{t+1} = \emptyset$.

(See Figure 1 for an example DDN visualized by Decisions.jl).

For our purposes, a decision network is a tuple (V, E, P, s). A *dynamic* decision network is any decision network where s does not map $\emptyset \to \emptyset$. For further formal discussion of decision networks, we refer the reader to Jensen and Nielsen [5].

For performance, Decisions.jl includes three other pieces of information alongside networks:

- Nodes are further labeled with the names of the random variables they represent. Homomorphic graphs with different labels are considered unique decision networks, regardless of the underlying distributions.
- Nodes may represent tensors of random variables with any number of axes (that is, "plates" are allowed). Along each axis, a node is marked as either *jointly* sampled along that axis, or *independently* sampled along that axis. We consider the axes as part of the node labels.
- Edges from and to plates are marked as either *parallel* or *dense* relationships along each axis.

Plates allow for parallelizable multiagent support that can be applied to extend any decision network into multiagent space. Figure 2 shows three possible plate relationships representable in Decisions.jl. The corresponding conditional probability distributions for node a, from top to bottom, are $P(a \mid m)$, $P(a_i \mid m_i)$, and $P(a_i \mid m)$.

Note that since decision networks need not be dynamic, there are some "sequential" decision problems represented by Decisions.jl that are not *strictly* sequential (that is, they are iterated only once). For instance, normal form games are supported.

2.2 Problem Classes and Instances

We borrow the language of object-oriented programming in formally defining problem "classes" (like "POMDP" and 'Markov game") and problem "instances."

A **sequential decision problem instance** q ("problem instance" informally) is a tuple (V, E, P, s, f, ρ). Specifically, q includes a decision network (V, E, P, s), an objective f, and objective parameters ρ. For performance, Decisions.jl uses objectives $f_\rho : K, X \to K$ for some arbitrary set K, where X is the space of $(x_1, x_2, ...)$ tuples and $x_1, x_2, ...$ are the random variables represented by the nodes of D. (That is, random variables are aggregated by f, step by step, into some value $\in K$.)

A **sequential decision problem class** C ("problem class" informally) is the set of all problem instances represented by (V, E, P, s, f, ρ) for some (V, E, s, f) and any (P, ρ). All problem instances lie in exactly one class (which we denote $\text{class}(q)$).

We denote the set of all problem classes as $\mathcal{D}$ and the set of all problem instances as $\mathcal{Q}$. Note that $\mathcal{D}$ is not simply the power set of $\mathcal{Q}$, since problem classes must be a set of *related* problem instances (sharing (V, E, s, f)), there are sets of problem instances that do not correspond to a problem class.

Decisions.jl primarily operates on problem classes rather than problem instances.

2.3 Standard Markov Problems

While all problems that can be represented with a decision network are supported, Decisions.jl explicitly provides names for problem classes widely used in the literature. One family of such problem classes is the *standard Markov family*, the set of problem classes where V is a subset of the following eight nodes:

- state (s) and successor state (sp), temporally linked
- memory (m) and successor memory (mp), temporally linked
- action (a)
- reward (r)
- observation (o), and
- sojourn time (τ).

As one would expect, the standard Markov family contains many common problems related to Markov decision processes and Markov games. However, not all "Markov" problem classes supported by Decisions.jl are in the standard Markov family - for instance, mixed observability POMDPs [11] are not a member.

Three of the nodes listed above merit further explanation:

- The *memory* and *successor memory* nodes appear in partially observable problem classes. They are temporally linked decision nodes that represent a belief update; that is, the memory tracks all information persisted by an agent from timestep to timestep. Explicit memory modeling has several benefits:
  - Policies, like all probability distributions in Decisions.jl, are built from pure functions. They do not have side effects, which would make it very difficult to guarantee that the

DNN structure is actually representative of the data flow in the problem.
  - The epistemics (that is, beliefs) of different agents can be easily compared.
  - In multi-agent settings, the individual memories of agents are explicitly tracked, allowing for rigorous modeling of hierarchical beliefs and epistemic observations.
- The *sojourn time* node appears in semi-Markov problem classes. Its inclusion in Decisions.jl allows the system to generalize semi-Markovianness from semi-MDPs to any standard Markov problem. For more information on formal modeling of semi-MDPs, we refer the reader to Puterman [12].

2.4 Mapping Semantic Traits to Problem Classes

To facilitate arbitrary composition of problem characteristics, Decisions.jl maps from "traits" (like "partially observable" or "decentralized") into the DAG (directed acyclic graph) underlying a problem class, given a problem class family (like "standard Markov"). As a result, in general, only the traits, distributions, and objective parameters need to be defined for any problem.

Figure 3 shows a graphical representation of the effects of these traits on the underlying decision network. Here (and in Decisions.jl in general), infixed indices like `i` are used to denote parallelization over an edge. Each of the eight listed traits has a distinct effect on the graph, with the possible[1] values listed.

Decisions.jl not only supports each defined problem trait individually, but also supports intersections of traits. This allows Decisions.jl to represent a wide variety of "nonstandard" or less-studied problem classes which, far from being "edge cases," represent some intricate problems in novel ways. Next, we explore how Decisions.jl represents one such problem class, the *centralized, partially observable Markov games*.

2.5 Case Study 1: Centralized POMGs

This case study illustrates the flexibility of Decisions.jl in representing "nonstandard" problems.

Decentralized POMDPs (Dec-POMDPs), *partially observable Markov games*, and *multi-agent* (centralized) *POMDPs* are three closely related and frequently studied partial-information scenarios (all of which are representable in Decisions.jl). In denoting these frameworks, we typically assume that when agents share a belief, they are cooperative.

However, belief centralization and cooperation are orthogonal traits: it is possible for agents to share a belief, but interact adversarially. In fact, the agent or agents controlling the belief (that is, the memory node) need not have any direct control whatsoever over the action. We call this scenario a *centralized partially observable Markov game*, or Cen-POMG, and it can be modeled with Decisions.jl.

As a motivating example[2], consider the following "adversarial information aggregation" game. A set of humans, $\mathbf{A_i}$, interacts with an information aggregation system, $\mathbf{B}$. We assume $\mathbf{A_i}$ are effectively memoryless: they rely on $\mathbf{B}$ to store data, as humans often do. However, $\mathbf{B}$ does not necessarily share an objective with all $\mathbf{A_i}$. Only $\mathbf{A_i}$ can affect the world state; $\mathbf{B}$ can only affect the belief used by $\mathbf{A_i}$. This is a Cen-POMG.

In Decisions.jl, a modeler would first identify the salient traits as being `Multiagent`, `Competitive`, `PartiallyObservable`, and `Centralized`. Assuming a Markov family structure, Decisions.jl uses these traits to create the decision network depicted in Fig. 1, where $\mathbf{A_i}$ controls a_i and $\mathbf{B}$ controls m. Next, the modeler provides implementations for `o`, `sp`, and `r_i` according to the generated network, as well as any necessary parameters for the objective (typically taken to be discounted sum-of-rewards in standard Markov family problems). This defines the sequential decision problem instance which models the adversarial information aggregation game (as well as the problem class, Cen-POMG, in which it lies).

Normally, this would be followed by the chore of writing efficient simulation and utility code for a Cen-POMG before any further progress could be made. In Decisions.jl, **this is done automatically.** Specifically, all information about problem class is stored in type space, so utility functions can be compiled just-in-time, tailored to that problem class. For instance, Decisions.jl automatically generates the following rollout loop for a Cen-POMG with N agents:

Require: state s, initial memory m, objective $f_\rho(\ldots)$
loop
  for $i \in 1..N$ **do**
    $a_i \sim P_{a_i}(\cdot \mid m)$
  end for
  $sp \sim P_{s_p}(\cdot \mid a, s)$
  if $sp = \text{TERMINAL}$ **then**
    return (s, sp, m, mp, a, r, o)
  end if
  $o \sim P_o(\cdot \mid a, s, sp)$
  for $i \in 1..N$ **do**
    $r_i \sim P_{r_i}(\cdot \mid a, s, sp)$
  end for
  $mp \sim P_{mp}(\cdot \mid a, m, o)$
  $m \leftarrow mp$
  $s \leftarrow sp$
  if $f_\rho(a, mp, o, r, sp, m, s)$ **then**
    return (a, mp, o, r, sp, m, s)
  end if
end loop

Decisions.jl generates this loop verbatim. Using this general approach, code optimizations can be applied to *all* problem classes (when they might otherwise be overlooked in an ad-hoc framework or problem extension). For instance, Decisions.jl can optionally sample nodes in parallel and in place, or enable/disable terminality checking on any particular node. (Terminal is a special value in Decisions.jl; any node can indicate Terminal conditions.) Relatedly, for performance, the objective f_ρ outputs a Boolean value to allow early stopping (the actual aggregated objective value is statically stored elsewhere).

[1] Reward style is a special case. For instance, problems with reward functions $r(s)$, $r(s,a)$, and $r(s,a,s')$ are all, at times, considered to be MDPs, despite the inconsistency this causes in the DDN. As such the `RewardStyle` trait lists exactly what the reward edges are expected to be.

[2] This example is based on the *partially observable assistance games* of Emmons et al. [4], with an adversarial twist.

Multiagency
(No agent, single agent, multiagent)
State structure
(Stateful, agent factored, stateless)
Observability
(Partial, full)
Reward style*
(None, single, multiple)
Memory presence
(Present, absent)
Semi-Markovness
(Semi-Markov, fixed timestep)
Cooperation
(Cooperative, competitive, independent)
Agent centralization
(Centralized, decentralized)

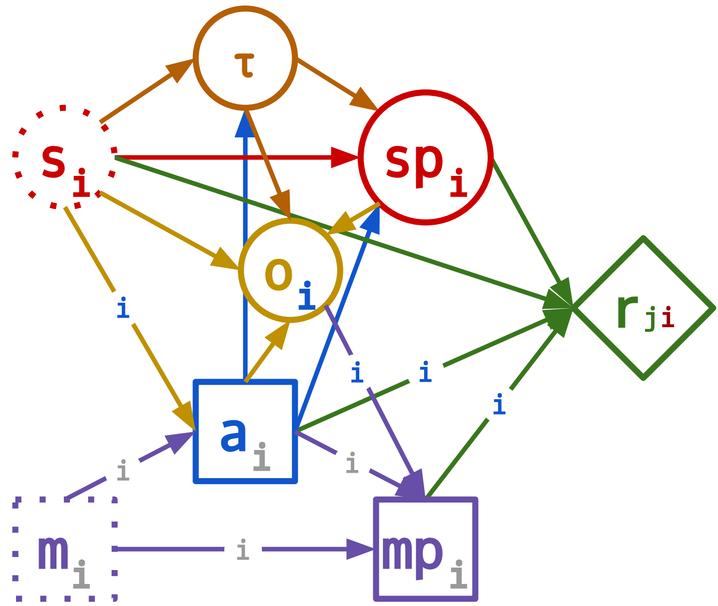

Figure 3: The effect of several Decisions.jl problem traits on nodes and edges in a standard Markov problem network. Each node, edge, and index is colored according to the last listed trait that affects its presence. (Colorblind accessible version available on demand).

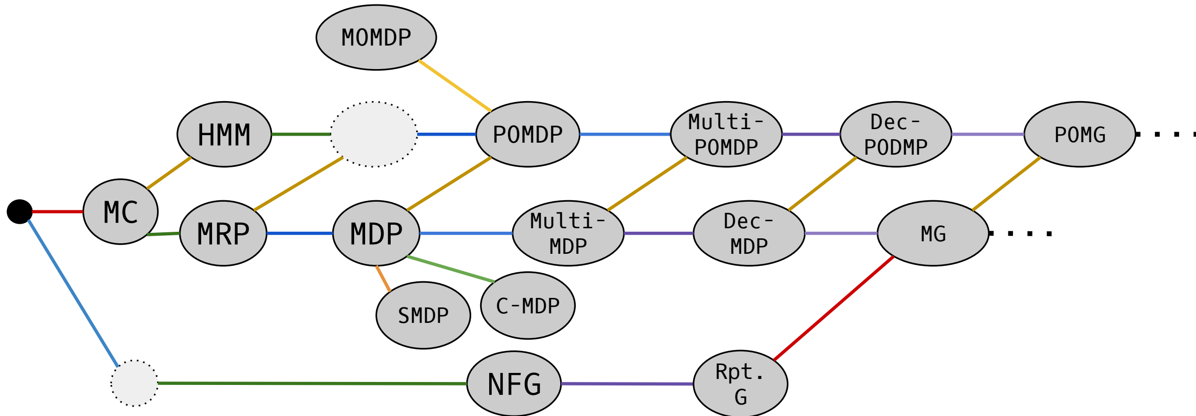

Figure 4: Relationships between several sequential decision problem classes. Each edge represents a transformation (or sequence of transformations); edges sharing a color represent the same transformation. Dotted nodes represent problem classes which have no commonly used name.

With this interface in place, the modeler is free to implement the desired operations or algorithms on all problems in the Cen-POMG class, including the adversarial information aggregation game.

3 NAVIGATING PROBLEM CLASSES

In this section, we detail how Decisions.jl can be used to navigate the space of problem classes. Its transformations add, remove, and modify problem assumptions in one line of code, significantly speeding up the prototyping process.

3.1 Problem Transformations

Transformations in Decisions.jl are its implementation of *game view operations* [16] (though not limited strictly to games).

A **sequential decision problem transformation** is a mapping $T : Q_{\text{in}} \to Q_{\text{out}}$, where $Q_{\text{in}}, Q_{\text{out}} \subseteq Q$. Furthermore, $\text{class}(q) = \text{class}(q') \implies \text{class}(T(q)) = \text{class}(T(q'))$ for problem instances q and q'. (That is, all problems sharing a class still share a class after the transformation). For convenience, we say a transformation maps problem class C to problem class C' if problem instances in C map to C'.

Transformation	New class	Notes
Original problem	**MDP**	
Insert((Dense(:s), Dense(:a), Dense(:sp)) => Joint(:o))		Add (o | s, a, sp).
Insert((Dense(:m), Dense(:a), Dense(:o)) => Joint(:mp))		Add (mp | m, a, o) - memory node to track belief.
InsertDynamic(:m => :mp)		Temporally link m and mp.
Implement(; :o=dist)		Set observation distribution to dist.
Recondition(; a = (Dense(:m),))	**POMDP**	Set action to be conditioned on belief, not state.
AddAxis((:a, :mp, :o, :r), :i, N)		Repeat action, memory, observation, and reward for N agents.
Recondition(; a = (Parallel(:m, :i),))		Indicate that each agent depends only on its own memory.
Recondition(; mp = (Parallel(:m, :i), Parallel(:o, :i), Parallel(:a, :i)))		Indicate that memory is conditioned separately for each agent.
WithIndep((:a, :r, :mp))	**POMG**	Specify that most nodes are sampled independently for each agent.

Table 1: A sequence of Decisions.jl transformations that generalizes an MDP into a POMG.

Transformation	New class	Notes
Original problem	**POMG**	
IndexExplode(:i)		Split agent-indexed nodes into separate nodes for each agent.
Implement(; a2 = dist)		Set opponent policy to dist.
Implement(; mp2 = dist)		Set opponent belief update to dist.
MergeForward(:r2, :a2, :mp2, <...>)		Treat opponent action and memory as part of the state.
Rename(; a1=:a, r1=:r, m1=:m)	**POMDP**	Treat agent 1 as ego agent.
Unimplement(:r, :o,:mp)		Prepare to modify distributions for some nodes.
MergeForward(:s, :sp, :o)		Wrap s, o into memory node (as environment model).
Implement(; mp = <...>)		Define belief update and belief sampling rules.
Recondition(; r=<...>)		Set reward to be belief-dependent.
Implement(; r = <...>)		Specify belief-dependent implementation of reward.
Rename(; m = :s, mp = :sp)	**(Belief) MDP**	Treat belief as state.

Table 2: A sequence of Decisions.jl transformations that transforms a POMG into a MDP by assuming opponent behavior as part of the state and transforming into a belief MDP. Problem is assumed to be two-agent.

There are some important consequences of this formulation:

- There is no "direction of complexity" for transformations. The output problems can be simpler, more complicated, or equivalent to the original.
- Transformations are general: a single transformation can transform multiple problem classes.
- There may be multiple transformations that map from any particular problem class to another.

Figure 4 shows the relationship of many common problem classes in terms of transformations that relate them. Unique transformations are indicated by color. In practice, Decisions.jl modularizes transformations as much as possible, so each transformation shown in in Figure 4 is in fact a sequence of lower-level transformations.

In the remainder of this section, we consider two use cases of Decisions.jl's transformations. In subsection 3.2, we quickly complicate and simplify an initial problem in a modular way, ideal for prototyping. Then, in subsection 3.3, we demonstrate an elegant formulation of a multi-agent decision making algorithm.

3.2 Case Study 2: Complicating and Simplifying Gridworld

Gridworld is a ubiquitous sequential decision making environment in which one or more agents navigate on a discrete grid for some arbitrary reward. When exploring a particular problem class, Gridworld is often among the first domains considered. Implementing Gridworld is easy, but doing so for multiple new problem classes (especially when prototyping problem formulations) is tedious and unnecessary. Through Decisions.jl's transformations, this can be done flexibly and automatically.

3.2.1 From MDP to POMG. Beginning with a MDP Gridworld instantiation, Table 1 shows a sequence of Decisions.jl transformations that change it into a partially observable Markov game by way of several intermediate problem classes - some named, some not. As in the prior case study, optimized simulation, sampling, and evaluation is available for every intermediate problem class.

Note that in this case, the only additional information provided to Decisions.jl is the distribution for o in the single Implement transformation. In particular, since the AddAxis transformation simply

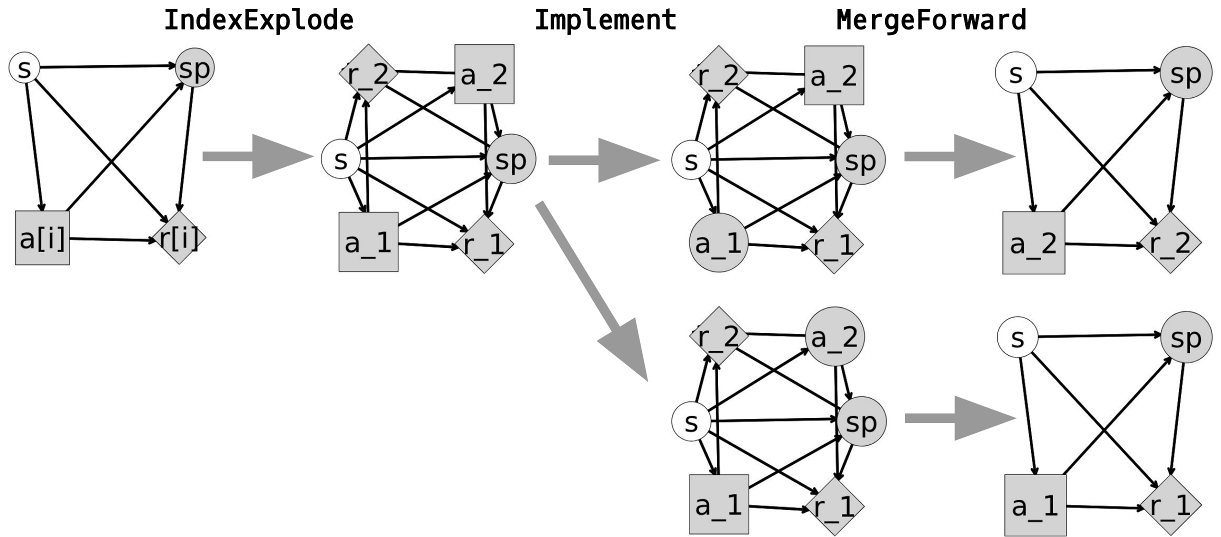

Figure 5: A sequence of Decisions.jl transformations, and the intermediate DDNs, that can be used to solve a Markov game with fictitious play.

extrudes the single-agent behavior onto n agents, no information is provided about agent interaction. It can be added post-hoc by `Implementing` it for the corresponding node.

3.2.2 From POMG to MDP. Many transformations, like those just demonstrated act trivially on the problem class by adding or altering problem characteristics. However, many decision making algorithms rely on transforming problems into a simpler class while maintaining aspects of the original problem description.

To demonstrate this, Table 2 shows a sequence of transformations that transforms the POMG-GRIDWORD we just created back into an MDP. Of course, one such sequence simply undoes the previous transformations and returns the original MDP, but this would result in a loss of generality: the additions of partial observability and multiagent interaction would be lost. Instead, we demonstrate a different sequence of transformations that *adds* problem information: first by assuming opponent behavior, then by indicating a belief update and transforming the problem into a belief MDP.

3.3 Case Study 3: Fictitious Play via Game Views

Many algorithms for decision making can be concisely expressed as a sequence of problem transformations. In this case study, we express fictitious play via problem transformations. We operate on a Markov game - a simple two-player setting in a GRIDWORLD - and make heavy use of the sequence of transformations described in Sec. 3.2.2 which treats opponent agents as part of the state.

The broad approach is as follows. We initialize both players to random policies and begin by transforming the game by splitting the `a` (action) and `r` (reward) nodes into two distinct nodes, one for each player. Then, iteratively, for each player, we assume the opponent will play the empirical average of past policies. We assign that policy to the opponent action node, merge it into the state, and apply an MDP solver, as described by the following snippet (where K is an arbitrary number of iterations):

Require: MG m
  $\pi_0^{(1)}$ = <random policy>
  $\pi_0^{(2)}$ = <random policy>
  m_{exp} = IndexExplode(m, ‘i’)
  for $t \in 1..K$ **do**
    $m_{\text{1p}}^{(1)}$ = Implement(m_{exp}, ‘a2’ $\to$ empirical_avg($\pi_{1..t}^{(2)}$))
    $m_{\text{merge}}^{(1)}$ = MergeForward($m_{\text{1p}}^{(1)}$, ‘r2’, ‘a2’)
    $m_{\text{mdp}}^{(1)}$ = Rename($m_{\text{merge}}^{(1)}$, ‘a1’ $\to$ ‘a’, ‘r1’ $\to$ ‘r’)
    $\pi_{t+1}^{(1)}$ = solve_mdp($m_{\text{md}}^{(1)}$)

    $m_{\text{1p}}^{(2)}$ = Implement(m_{exp}, ‘a1’ $\to$ empirical_avg($\pi_{1..t}^{(21)}$))
    $m_{\text{merge}}^{(2)}$ = MergeForward($m_{\text{1p}}^{(2)}$, ‘r1’, ‘a1’)
    $m_{\text{mdp}}^{(2)}$ = Rename($m_{\text{merge}}^{(2)}$, ‘a2’ $\to$ ‘a’, ‘r2’ $\to$ ‘r’)
    $\pi_{t+1}^{(2)}$ = solve_mdp($m_{\text{md}}^{(2)}$)
  end for

Figure 5 shows the sequence of decision networks underlying the transformed problem, transformation by transformation (excluding the final `Rename`).

4 CONCLUSION

In this paper, we present Decisions.jl, a software framework with the goal of supporting all sequential decision making problems representable with a dynamic decision network.

We first discuss the architecture of Decisions.jl, which allows for flexible and expressive problem definitions. We provide a motivating multi-agent example which demonstrates the applicability of this architecture to problem classes with complex combinations of

traits that are not readily supported in other architectures. Second, we provided an in-depth description and examples of problem class transformations in Decisions.jl, which allow researchers to rapidly prototype new test domains, precisely indicate "game view operations," and elegantly implement algorithms which rely on multiple problem representations. We exercised this prototyping power in two additional case studies: one generalizing and simplifying a plain single-agent Gridworld environment, and the other demonstrating how Decisions.jl can readily be used to implement fictitious play via repeated transformations.

4.1 Future Work

As a closing thought, we classify Decisions.jl as "mathematical middleware": a type of software aimed at relating the mathematical frameworks used by other software. The purpose of mathematical middleware is simultaneously functional (helping automate tedious modeling and problem-alteration tasks), and theoretical (enforcing rigor and connection to mathematical formulation). These two goals are sometimes at odds, and we have primarily discussed the former while disregarding contributions to the latter.

More specifically, Decisions.jl enforces precise (DDN-backed) problem descriptions only at the lowest modeling level, and the interactions agents have with the environment and with each other on the episode-to-episode level are not modeled. These meta-interactions are iterative, probabilistic interactions, and can be described as DDNs with (training) objectives - that is, as decision problem instances. As such, we see a possibility in extending Decisions.jl to formally model training loops, decentralized execution, and environment interactions. This would further improve Decisions.jl's capabilities in providing a common language for decision-making algorithms.

ACKNOWLEDGMENTS

The authors would like to thank Dr. Forrest Laine for insights provided during the development of Decisions.jl.

REFERENCES

[1] Tyler Becker and Zachary Sunberg. 2024. Bridging the Gap between Partially Observable Stochastic Games and Sparse POMDP Methods. *arXiv preprint arXiv:2405.18703* (2024).
[2] Jeff Bezanson, Stefan Karpinski, Viral B Shah, and Alan Edelman. 2012. Julia: A fast dynamic language for technical computing. *arXiv preprint arXiv:1209.5145* (2012).
[3] Maxim Egorov, Zachary N. Sunberg, Edward Balaban, Tim A. Wheeler, Jayesh K. Gupta, and Mykel J. Kochenderfer. 2017. POMDPs.jl: A Framework for Sequential Decision Making under Uncertainty. *Journal of Machine Learning Research* 18, 26 (2017), 1–5. http://jmlr.org/papers/v18/16-300.html
[4] Scott Emmons, Caspar Oesterheld, Vincent Conitzer, and Stuart Russell. [n.d.]. Observation Interference in Partially Observable Assistance Games. In *Forty-second International Conference on Machine Learning*.
[5] Finn V Jensen and Thomas D Nielsen. 2007. *Bayesian Networks and Decision Graphs: February 8, 2007*. Springer.
[6] Mykel J Kochenderfer, Tim A Wheeler, and Kyle H Wray. 2022. *Algorithms for decision making*. MIT press.
[7] Akshat Kumar, Shlomo Zilberstein, and Marc Toussaint. 2015. Probabilistic inference techniques for scalable multiagent decision making. *Journal of Artificial Intelligence Research* 53 (2015), 223–270.
[8] Marta Kwiatkowska, Gethin Norman, and David Parker. 2011. PRISM 4.0: Verification of probabilistic real-time systems. In *International conference on computer aided verification*. Springer, 585–591.
[9] Marta Kwiatkowska, Gethin Norman, David Parker, and Gabriel Santos. 2020. PRISM-games 3.0: Stochastic game verification with concurrency, equilibria and time. In *International Conference on Computer Aided Verification*. Springer, 475–487.
[10] Marc Lanctot, Edward Lockhart, Jean-Baptiste Lespiau, Vinicius Zambaldi, Satyaki Upadhyay, Julien Pérolat, Sriram Srinivasan, Finbarr Timbers, Karl Tuyls, Shayegan Omidshafiei, Daniel Hennes, Dustin Morrill, Paul Muller, Timo Ewalds, Ryan Faulkner, János Kramár, Bart De Vylder, Brennan Saeta, James Bradbury, David Ding, Sebastian Borgeaud, Matthew Lai, Julian Schrittwieser, Thomas Anthony, Edward Hughes, Ivo Danihelka, and Jonah Ryan-Davis. 2019. OpenSpiel: A Framework for Reinforcement Learning in Games. *CoRR* abs/1908.09453 (2019). arXiv:1908.09453 [cs.LG] http://arxiv.org/abs/1908.09453
[11] Sylvie CW Ong, Shao Wei Png, David Hsu, and Wee Sun Lee. 2010. Planning under uncertainty for robotic tasks with mixed observability. *The International Journal of Robotics Research* 29, 8 (2010), 1053–1068.
[12] Martin L Puterman. 2014. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons.
[13] Ahti Salo, Juho Andelmin, and Fabricio Oliveira. 2022. Decision programming for mixed-integer multi-stage optimization under uncertainty. *European Journal of Operational Research* 299, 2 (2022), 550–565. https://doi.org/10.1016/j.ejor.2021.12.013
[14] Tuomas Sandholm. 2015. Abstraction for solving large incomplete-information games. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 29.
[15] Gabriele Venturato, Vincent Derkinderen, Pedro Zuidberg Dos Martires, and Luc De Raedt. 2024. Inference and learning in dynamic decision networks using knowledge compilation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 20567–20576.
[16] Michael P Wellman and Katherine Mayo. 2024. Navigating in a space of game views. *Autonomous Agents and Multi-Agent Systems* 38, 2 (2024), 31.