MAIDs, NIDs and Hierarchical Reinforcement Learning

Mike Vlad Cora
mcora@cs.ubc.ca
University of British Columbia
CPSC532 - Multiagent Systems
Final Report

December 27, 2006

Abstract

This paper investigates two graphical modeling approaches used for multi-agent learning. Multi-agent influence diagrams (MAIDs) explicitly model the dependencies of decisions and utility on nature's random variables and other agents' decisions. Hierarchical RL on the other hand allows the decomposition of actions into sub-actions (think of actions as tasks composed of sub-tasks, or routines and sub-routines). This hierarchy constrains the policy/strategy space we have to search through. Combining the two approaches may result in a comprehensive graphical modeling package that gives the programmer explicit control over the agent, in a probabilistic, multi-agent learning framework.

1 Introduction

A real world environment for multi-agent interaction is, simply put, infinite. It is laden with uncertainties due to partial observability, limited computational and communication abilities, mistaken assumptions during opponent modeling, faulty updating of beliefs from observations, etc. All of these factors lead to bounded rationality[10], where the agents can only be expected to act rationally within the finite reasoning space they have available (if at all). This applies equally well when reasoning about both human and computer agents.

Therefore, the road from theory to application when building intelligent agents inevitably involves entering prior knowledge in the form of theoretical model choice, program architecture, parameter tuning, and so on. The argument for modeling a problem domain's structure is well grounded both theoretically and empirically. Any real-world plans that people make will be highly structured in state, action and policy space. This paper summarizes two main approaches for describing a planning problem that seem orthogonal to each other and thus may be combined for further expressivity gains.

Multi-agent influence diagrams (MAIDs)[6] have been recently introduced as a graphical modeling framework firmly grounded in both decision theory and game theory. At the core, a MAID is a Bayes net extended with agent-specific decision and reward nodes, and they are described further in section 2. The orthogonal representation is hierarchical reinforcement learning [1] which models repeated decision making by structuring the action/policy space into tasks (actions) composed of subtasks that last a various amount of time. The leaf nodes of the hierarchy are the primitive actions in the game (in this paper, the terms actions, tasks and subtasks are used interchangeably). This is further described in section 3, with section 4 discussing possible ways of bringing HRL and MAIDs together into an online model-based multi-agent learning algorithm. We finally conclude with section 5.

2 Probabilistic Graphical Models

Graphical models are like onions. They each add a layer of representation over the previous formalisms, to capture different aspects of the modeled domain. It is beneficial to stick with the formalisms that are based on vast amounts of sound theory and experimental evidence, such as Bayes nets.

2.1 From Bayes Nets to MAIDs

A Bayes net is defined as a graph where the nodes (ovals) are random variables that represent the world state, and the directed links between them represent conditional dependance relationships (see Figure 1). The entire Bayes net $B = (G, Pr)$ defines a joint probability distribution over the random variables $X_1, ...X_n$ given their parents Pa, using the chain rule:

$$P(X_1, ..., X_n) = \prod_{i=1}^{n} Pr(X_i|Pa(X_i)) \tag{1}$$

The modeling parameters that can be learned from data are the conditional probability distributions $Pr(X_i|Pa(X_i))$ and the graph structure G (which is obviously harder) [2]. Given these parameters, we can infer the value of any variable in the graph given some observations, reason about the model validity, and compare different model accuracies.

An influence diagram extends this formalism with decision (square) and utility (diamond) nodes in the single agent case. The reward nodes U_i are the simplest, representing a deterministic function that maps the instantiation of its parents $Pa(U_i)$ to a real number. They are simply random variables with all probability mass on a single value (per parent instantiation). A decision D is a function that maps each instantiation of its parents $Pa(D)$ to a probability distribution over its domain (or equivalently, a choice over actions). The agent is free to choose or learn this decision rule, leading to a strategy σ. Applying a strategy to an influence diagram means converting its decision nodes D_i to random variables with $\sigma(D_i)$ as their CPDs. Thus, an influence diagram with an instantiated strategy σ becomes a Bayes net. A strategy can be calculated by maximizing the expected value of all utility nodes:

$$EU(\sigma) = \sum_{U} \sum_{r \in U} P_\sigma(U = r) \cdot r \tag{2}$$

Koller and Mitch [6] have extended this powerful formalism into the multi-agent environment, where decision and utility nodes are now agent specific (see Figure 3). The dependencies between decision nodes of other agents are explicit, and so the model is expressive enough to represent all extended form games in a more compact way. The notion of *strategic relevance* between decision nodes is used to split the game into a series of smaller, independent sub-games that can be solved faster, and whose results are combined to form an overall solution. Solving a MAID means computing its *Nash equilibrium* using the expected utility defined above.

If a decision D' *strategically relies* on D, then the agent must consider D when optimizing D'. Using the graph criteria of *s-reachability* (similar to the Bayes-ball mechanism), strategic relevance can be computed in polynomial time of the graph size. The MAID formalism allows potentially exponential gains in both representation and computation of games with certain structural properties. If the game is non-symmetrical however (such as the centipede game), a naive MAID representation can actually be exponentially bigger than the equivalent game tree. Describing the algorithm for solving a MAID is beyond the scope of this report, so please refer to Koller and Mitch [6] for clear and detailed explanations. The main point is that

MAIDs can describe more complex games with less notation. Computing the *Nash equilibrium* can also be exponentially faster by taking advantage of the game's structure.

2.2 Network of Influence Diagrams

A further extension to MAIDs has been introduced recently by Gal and Pfeffer [4] [5]. A single MAID captures one particular model of a game that all the agents reason with. The NID formalism captures situations where agents have different beliefs about the game, or instead of playing their best response, they play as automatons or according to some heuristics (ie. they have bounded rationality). So when facing an opponent, we might have a particular root model that we consider rational, but we cannot assume the same of the other agents involved. They might not even know the game played. This is captured by allowing different MAIDs to represent a particular decision of an agent. The decision rule for a decision node then is a probability distribution of the possible MAID models the agent might have for optimizing that particular decision. By allowing a distribution over the opponent's models, better strategies are achievable that take advantage of possible weaknesses in their strategy. NIDs additionally allow modeling the effect of collusion and alliances to some degree, and can solve these games using the MAID algorithms.

To summarize briefly, the papers cited show that NIDs are a superset of Bayesian games, and thus much more expressive. In Bayesian games, the uncertainty over other agents is all captured by a probability distribution over their types, and associated utility function. In a NID, the uncertainty over an agent's type is modeled by the entire expressive power of a MAID. The (gross summary of the) algorithm for solving a NID involves solving the MAIDs at the leaf nodes, and then "aggregating" the solutions into the root MAID and solving it. This equilibrium is defined as rational according to the agent's self-beliefs (about the game played), and beliefs about what models the other agents hold.

3 Hierarchical Reinforcement Learning

Notice that so far we have reasoned only about decisions, and their utilities. However, between making a decision and receiving a payoff lies the great perilous gulf of executing actions that extend through time and whose outcome may be uncertain. Hierarchical RL focuses on this area by treating actions as tasks composed of other subtasks that focus on specific areas of the state and action space. As with most things, HRL follows the onion analogy, with each layer adding more expressive power.

3.1 MDPs and Semi-MDPs

Reinforcement learning models sequential decision making, where the agent observes the world state, performs an action and receives a reward. Proven convergent algorithms calculate an optimal policy that picks the best action to take in each state in order to maximize the reward. The generalized formulation, the *Markov Decision Process* is basically a chain of action, utility and world state nodes in an influence diagram. It is used to model sequential decisions in a dynamic environment, given the assumption that the future world state depends only on the current state and the actions taken. This simply means that any memory we may have about the past needs to be included into the joint state space S. An MPD is defined as a 4-tuple (S, A, P, R):

$$MDP = \begin{cases} S & \text{set of the joint world states} \\ A & \text{set of actions} \\ P(s'|a,s) & \text{prob. of transitioning into } s' \text{ having taken action } a \text{ in state } s \\ R(s,a) & \text{reward received when taking action } a \text{ in state } s \end{cases} \tag{3}$$

The process is defined by the following value and corresponding Q functions (called the Bellman equations):

$$V^*(s) = max_{a \in A} \left[R(s,a) + \beta \sum_{s',N} P(s'|s,a) V^*(s') \right] \tag{4}$$

$$Q^*(s,a) = R(s,a) + \beta \sum_{s',N} P(s'|s,a) V^*(s') \tag{5}$$

There exist several algorithms that can solve for the optimal policy $\pi^*(s) = \arg\max_{a \in A} Q^*(s,a)$ by taking actions and observing the new state and reward received, without ever having to compute the transition probability function P (thus the name model-free). The simplest algorithm, Q-learning, performs the following iterative update at every timestep (where t defines the timestep, α is the learning rate and β is the discount factor):

$$Q_{t+1}(s,a) \leftarrow (1-\alpha) Q_t(s,a) + \alpha [R(s,a) + \beta \max_{a' \in A} Q_t(s',a')] \tag{6}$$

This algorithm is guaranteed to converge to the optimal policy in the limit of infinite samples of states and actions (in practice, convergence is much quicker). A Semi-MDP also takes into account the number of timesteps N for an action to complete by modifying the transition probability distribution to $P(s', N|a, s)$. The algorithms for solving an SMDP are fairly straight forward extensions of the MDP algorithms and are left out for brevity. All hierarchical RL formulations are basically factored SMDPs, as described below.

3.2 Hierarchical Value Function Decomposition

Hierarchical RL methods allow actions composed of sub actions by representing an action as an SMDP composed of sub SMDPs, and including the callstack of the routines into the world state. The Q function of each action (or task or routine) has to take into account how long its child actions take, and their total aggregated reward. The main differences between HRL formulations is how the value function is decomposed, the task termination criteria, and the algorithmic details of updating the Q function.

Diettreich's MAXQ [3] two-part decomposition is simple to understand (see Figure 2). A subtask M_i is defined as a tuple (A_i, G_i, Z_i) where A_i is a set of subtasks, $G_i(s)$ is a termination predicate, and Z_i is a state abstraction function that returns a subset of the joint state relevant to the current subtask.

$$s = Z_i(S) \tag{7}$$

$$a = \pi_i(s) \tag{8}$$

$$Q^\pi(i,s,a) = V^\pi(a,s) + C^\pi(i,s,a) \tag{9}$$

$$C^\pi(i,s,a) = \sum_{s',N} P_i^\pi(s', N|s,a) \beta^N Q^\pi(i, s', \pi_i(s')) \tag{10}$$

$$V^{\pi}(i,s) = \begin{cases} Q^{\pi}(i,s,\pi_i(s)) & \text{if } i \text{ is composite} \\ \sum_{s'} P(s'|s,i)R(s'|s,i) & \text{if } i \text{ is primitive} \end{cases} \tag{11}$$

This looks more complicated than it is. Here, i is the current action, and a is a child action given that we are following policy π_i. The Q function is decomposed into two parts: the value of V^{π} being the expected one step reward, plus C^{π} which is the expected completion reward for the current action. Notice that N here is the number of timesteps that an action takes to complete, care of the SMDP formulation. Additionally, V is defined recursively, as the expected value of its child actions, or the expected reward itself if i is a primitive (atomic) action.

There are other hierarchical RL formulations with slightly different details, but the overall ideas are the same. The programmer specifies a task hierarchy, and the system learns which actions to pick at the specified choice points. One might ask what use is this if the analyst still has to enter the task hierarchy. Well in most cases breaking a plan into sub-plans is much easier than creating the rules for which tasks to execute when, which is exactly what the algorithm learns. The gains come from the fact that the policy space is constrained by the subtask hierarchy, and the state space is constrained at each subtask by the state abstraction function Z_i. This reduces both the amount of space required to store a full solution, as well as speeding up the learning process. Of course, care must be taken to create a proper task hierarchy with correct termination predicates and state abstraction functions. Needless to say, it is an open area of research.

An important consideration in hierarchical RL is whether the policy calculated is hierarchically or recursively optimal. Recursive optimality (satisfied by MAXQ [3]) means that each subtask is locally optimal, given the optimal policies of the descendants. This may result in a suboptimal overall policy. For example if there are two exits from a room, a recursively optimal policy would pick the closest exit for the task *ExitRoom*, regardless of the final destination. A hierarchically optimal policy (calculated by the ALisp [8] three-part value decomposition) would pick the exit to minimize total traveling time, given the destination. A recursively optimal learning algorithm however generalizes subtasks easier since they only depend on the local state, ignoring what would happen after the current task finishes. So both types of optimality are of value in different degrees for different cases.

4 Bringing the two together

As should be apparent, the two methods describe two different things. The MAID and NID graphical models allow the analyst to describe how the decisions and rewards depend on other decisions, world variables and agents involved. The decisions themselves however are instantaneous, with the reward received immediately. By taking advantage of the game structure, computing the *Nash equilibrium* of a MAID can be exponentially faster than an equivalent extended form game. Additionally, a NID allows modeling an agent's belief about themselves, the game played, and of other agents by tying together different MAIDs. The equilibrium solution is then a best-response given the current belief of the agent.

In hierarchical RL, actions take more than one timestep and can themselves be composed of other actions. For example a simple task of navigating from ones house to the grocery store might involve such detailed subtasks as finding the keys, locking up, playing the elevator mini-game, navigating a series of intersections from memory (or following a Google printout), while avoiding the pedestrians and playing Frogger across the street. Each of these tasks involves different parts of the world state, and different categories of features. For example continuous Euclidean positioning would be necessary to finely avoid another pedestrian, but navigating across a city grid requires only discrete intersections (and invoking the collision avoidance subtask). Whether or not we have the house keys does not matter, the navigation task will succeed regardless. We do not re-learn these tasks for every new plan we are faced with, but have generalized them into separate subtasks (or games) that we can invoke at the right time, across many instances of the world state. This is

exactly the sort of abstraction hierarchical RL strives to capture.

Clearly, a full-featured multi-agent graphical modeling language would allow us to model everything: procedural abstraction (tasks and subtasks), actions that last through time, state abstraction (rewards depend only on the relevant decisions and world state variables), and finally opponent modeling. So how do we bring hierarchical RL and NIDs together? This author does not know, but can make some semi-educated guesses of directions that may work, and pose more questions than answers.

4.1 Multi-agent RL: Minimax-Q and Belief-Based RL

The minimax-Q algorithm [11] defines the Q function the same as (6) however a is now the joint action over all agents. This iterative algorithm computes a strategy that maximizes the minimum payoff when playing against the best set of punishing agents (which we know is a *Nash equilibrium* for a zero-sum game). The value of a state at time t is defined as:

$$V_t(s) = \max_{\pi_i(s)} \min_{o} Q_{i,t}(s, \pi_i(s), o) \tag{12}$$

where π_i is agent i's policy, and o is the joint action of the opponents. For zero-sum games, the algorithm converges to the value of the game, following similar arguments as for the single-agent Q-learning algorithm. Such guarantees cannot be made of general-sum games however.

Q-learning (and R-max) are flat, model-free formulations that don't account for actions decomposed into subactions. The recent literature for extending hierarchical RL to the multi-agent setting, introduces algorithms that generally do not take a game-theoretic approach when solving for the optimal solution. They treat the system as stationary (ignoring other agents' changing strategies), and focus on faster learning by sharing action hierarchies [7], or have a central scheduler that pauses the agents for communication and optimization at a joint action choice [8]. So ideally, we would like to model the decision making process at each subtask of the plan hierarchy while taking into account the agent's beliefs about other agents, in a decentralized environment.

A possible bridge between HRL and NIDs is belief-based reinforcement learning:

$$Q_{t+1}(s, a) \leftarrow (1 - \alpha) Q_t(s, a) + \alpha [R(s, a) + \beta V_t(s')] \tag{13}$$

$$V_t(s) \leftarrow max_{a_i} \sum_{a_{-i} \subset A_{-i}} Q_t(s, (a_i, a_{-i})) P_i(a_{-i}) \tag{14}$$

Here, a is the joint action space of all agents, and $P_i(a_{-i})$ is the belief of agent i that the opponents will play actions a_{-i}. If we model the game as a NID, this probability distribution could be computed by solving the NID and optimizing the corresponding opponent's decision from the belief models available.

4.2 Fitting a Square into a Triangle

Glossing over the (unknown) details, let's assume that each subtask in a task hierarchy is a game with finite decisions modeled by a NID. How can the pieces fit together and what can be learned at each timestep? The NID structure implicitly represents the state abstraction function Z_i, since it contains all the state variable nodes that the rewards depend on. The function $Q(s, a)$ could in fact be seen as the maximum expected value of action a when instantiating and solving the NID with world state s.

In an MDP, the future state s' depends on the current state and the action taken $P(s'|s,a)$, modeling how an agent's actions will affect the world. This fact is completely ignored in MAIDs and NIDs, so they need an extension for the inter-timestep dependencies between states and actions. A sequence of dynamic MAID diagrams that represent each timestep may work. There is some theory of solving sequences of influence diagrams using LImited Memory Influence Diagrams (LIMIDs) [9], and the method may be extendable to MAIDs. It is unknown to this author how complexity is affected, how the *strategic relevance* property applies across timesteps, and what the exact definition of a best response would be.

For every timestep, the algorithm should update the conditional probability distributions (CPDs) of the active task's NID by observing the state and the opponents' actions, and running some variation of Maximum Likelihood Estimation or Expectation Maximization algorithms [2]. The reward nodes of the NID must also be updated with the observed reward from the environment, plus the expected total reward of the child tasks' NIDs, with $a = \pi_i(s)$ being the subtask returned by parent task i's policy:

$$EU(\sigma, i) = \sum_{a \in \pi_i(s)} \sum_{U \in NID_a} \sum_{r \in U} P_\sigma(U = r) \cdot r \tag{15}$$

This hierarchical expected utility aggregation is also an open question, but there seems to exist a similar formulation as the value function decomposition described by equations 7 through 11. The NID effectively becomes the model to the otherwise model-free Q function, which is more useful than a flat Q function (implemented as a table lookup for example), because it can answer more complicated questions, and can be more representationally compact. We already have at least two methods of aggregating MAIDs together: one using the *strategic relevance* which is applicable only for tasks with an independent utility function however (for example in a multi-tasking environment). The second method is how a NID aggregates many MAIDs to optimize a decision. The details are unknown, but there are still promising areas of literature left uninvestigated.

5 Conclusion

It is unreasonable to expect that an agent can learn how to behave intelligently from scratch, in the real-world or even in a slightly more complex game than the ones we have been studying (at least with our present day technology). This paper chiefly argues for a graphical modeling language that can handle all of the following aspects of multi-agent learning: model the dependencies of rewards and decisions on the world and other agents, break down a task into subtasks, and allow opponent modeling for optimal strategies. The two probabilistic formalisms summarized, MAIDs/NIDs and Hierarchical RL each handle a part of the whole package. It was hypothesized that combining these methods may result in a comprehensive language that would allow the analyst to model an agent in a probabilistic multi-agent framework that can learn strategies at many different levels of abstraction (care of the subtask hierarchy).

This paper has raised more questions than solutions; however, there are promising avenues of research for all the problems mentioned. The approach to the problem was roughly based on the process that a video game AI programmer follows when hard-coding intelligent agents. In a sports game for example, the hierarchical program starts at the team strategy level, composed of each player on the team, further broken down into high level actions such as "get the ball", all the way to the low-level animation stitching controller and physics engine. The glue holding it all together is scripted rules at each level of the behaviour hierarchy. The resulting agent appears to behave intelligently (bounded by the skill of the programmer); however, it has no way of learning new strategies.

One can sample the myriad algorithms and formalisms available to find those which best fit this process. It should be fairly clear that hierarchical RL matches the subtask decomposition requirement, the MAIDs

allow modeling the glue of decision making (replacing the hardcoded rules), and NIDs allow for opponent modeling to adapt to different agents' behaviors. The only obstacle left is assembling them into one comprehensive modeling framework.

References

[1] Andrew G. Barto and Sridhar Mahadevan, *Recent advances in hierarchical reinforcement learning*, Discrete Event Dynamic Systems **13** (2003), no. 1-2, 41–77.

[2] Wray L. Buntine, *Operations for learning with graphical models*, Journal of Artificial Intelligence Research **2** (1994), 159–225.

[3] Thomas G. Dietterich, *Hierarchical reinforcement learning with the MAXQ value function decomposition*, Journal of Artificial Intelligence Research **13** (2000), 227–303.

[4] Ya'akov Gal and Avi Pfeffer, *A language for modeling agents' decision making processes in games*, AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems (New York, NY, USA), ACM Press, 2003, pp. 265–272.

[5] ———, *Reasoning about rationality and beliefs*, AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (Washington, DC, USA), IEEE Computer Society, 2004, pp. 774–781.

[6] Daphne Koller and Brian Milch, *Multi-agent influence diagrams for representing and solving games*, IJCAI, 2001, pp. 1027–1036.

[7] Rajbala Makar, Sridhar Mahadevan, and Mohammad Ghavamzadeh, *Hierarchical multi-agent reinforcement learning*, Proceedings of the Fifth International Conference on Autonomous Agents (Montreal, Canada), 2001.

[8] Bhaskara Marthi, Stuart Russell, David Latham, and Carlos Guestrin, *Concurrent hierarchical reinforcement learning*, IJCAI-05 (Edinburgh, Scotland), ACM Press, 2005.

[9] Dennis Nilsson and Steffen L. Lauritzen, *Evaluating influence diagrams using limids*, UAI '00: Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence (San Francisco, CA, USA), Morgan Kaufmann Publishers Inc., 2000, pp. 436–445.

[10] Christos H. Papadimitriou and Mihalis Yannakakis, *On complexity as bounded rationality*, STOC '94: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing (New York, NY, USA), ACM Press, 1994, pp. 726–733.

[11] Yoav Shoham and Kevin Leyton-Brown, *Multi agent systems*, (To be published), 345–379.

6 Appendix - Sample Graphical Models

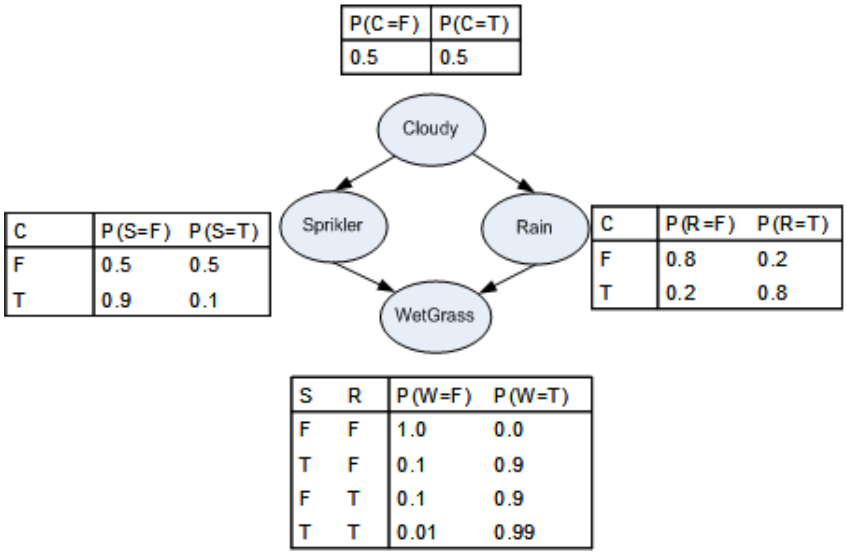

Figure 1: Typical sample of a Bayes net with associated conditional probability tables.

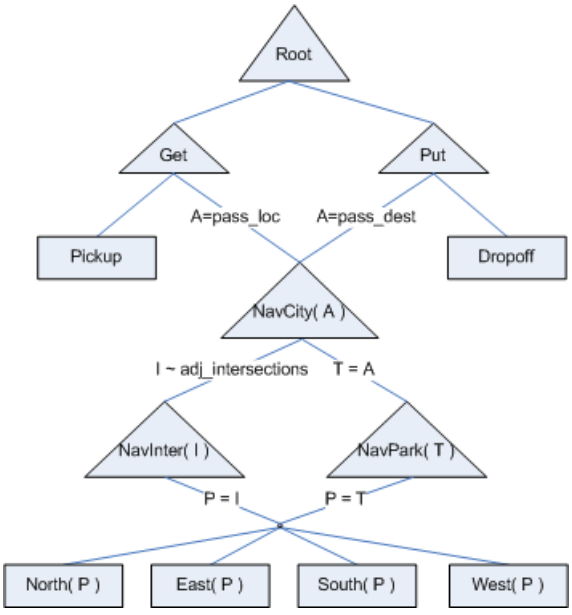

Figure 2: Sample MAXQ task hierarchy for a taxi that picks up a passenger, and then navigates to their destination. Navigating through the city depends only on the Address parameter passed in, and invokes NavInter to go from one adjacent intersection to another, and then NavPark to park the taxi beside the destination. The triangles are composite actions, and the squares are primitive actions taken by the taxi agent.

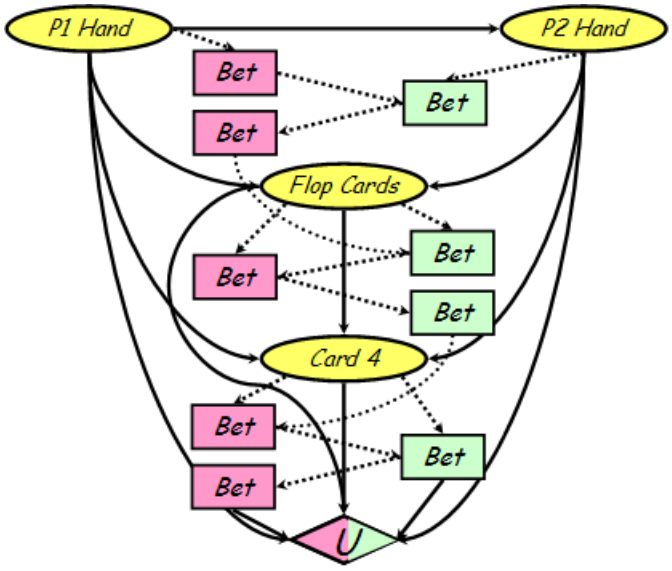

Figure 3: Sample MAID diagram of two player heads up poker, care of Koller and Mitch [6]. It models the sequence of decisions in a simplified Texas Hold'em deal for the two agents involved.