Information Set Generation in Partially Observable Games

Mark Richards and Eyal Amir
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801
{mdrichar,eyal}@illinois.edu

Abstract

We address the problem of making single-point decisions in large partially observable games, where players interleave observation, deliberation, and action. We present information set generation as a key operation needed to reason about games in this way. We show how this operation can be used to implement an existing decision-making algorithm. We develop a constraint satisfaction algorithm for performing information set generation and show that it scales better than the existing depth-first search approach on multiple non-trivial games.

Introduction

Many real-world decision problems can be framed as games, including securities trading, business acquisitions and mergers, and military operations. The focus of this paper is the extensive form game. Typically expressed in tree form, this formal construct can be used to reason about environments which involve multiple agents who make finite sequences of decisions in the presence of uncertainty about the state of the world and the decisions made by other agents.

A Nash equilibrium specifies an optimal choice for every decision point in the tree and, for two-player games, can be found in time polynomial in the number of nodes in the tree (Koller, Megiddo, and von Stengel 1994). However, many games are so large that it is not feasible to even enumerate a decision for every possible contingency.

In this work, we develop a system for using a list of observations and limited tree search to make single-point decisions. After making a single decision, the agent receives additional observations based on chance events or actions chosen by competing agents, and the process repeats. We express these games in our description language, POGDDL, which is an extension of the PDDL language commonly used in AI planning (Edelkamp and Hoffmann 2004).

In contrast to fully observable games like chess or checkers, players in partially observable games do not know exactly which node in the game tree corresponds to the current true state. Instead, players know only that the true state of the game is one of a subset of nodes in the tree, known as the player’s current information set.

Background

Kuhn formally described imperfect information games in terms of nodes and branches of a tree (Kuhn 1953). To describe a game using logical constructs, we adopt the following conventions. In a game tree $\Gamma$, each non-terminal node corresponds to a decision made by one of $K$ players $P_1, \ldots, P_K$ or a chance event (e.g., roll of a die) executed by a special player denoted $P_0$. The full game state $s$ at a node at depth $t$ comprises the true logical state $s.P$ and a list of observations $s.a_{1:t}^{(k)}$ for each player that constitute $P_k$’s knowledge base at that node. Nominally, $s.P$ is a subset of a finite set $P$ of propositional variables or ground predicates defined by a game description. The variables in $s.P$ are those that are $\text{TRUE}$ in $s$. If $\Gamma$ includes numerical values (e.g., a running score), $s$ encodes those values as well. Let $K(s)$ denote the player whose turn it is to move at $s$.

Let $a_{1:t}$ denote a sequence of actions $a_1, \ldots, a_t$, and let $a_{1:t}a_{t+1}$ denote the extension of that sequence by $a_{t+1}$. We
use $S(a_{1:t})$ to denote the state that is reached when action sequence $a_{1:t}$ is executed from the start state $s_0$.

Given a sequence of observations $o_{1:t}^{(k)}$ for $P_k$, the information set $I(o_{1:t}^{(k)})$ is the set of action sequences that lead to a state where $P_k$ would have those observations. That is, a legal sequence $a_{1:t} \in I(o_{1:t}^{(k)})$ iff $K(S(a_{1:t})) = P_k$, and $S(a_{1:t}).o_{1:t}^{(k)} = o_{1:t}$.

For a state $s$ corresponding to a terminal node, $U_k(s)$ denotes the numerical payoff (utility) for $P_k$ if that state is the outcome of the game.

Figure 1 shows a simple poker game for two players. The decision point for $P_0$, the dealer, is depicted as a circle. Decisions for $P_1$ and $P_2$ are shown as triangles and diamonds, respectively.

Both $P_1$ and $P_2$ initially hold two dollars and are required to put one dollar in the pot as an ante. The dealer, holding a King and a Jack ($K > J$), randomly deals one card to $P_1$ and the other to $P_2$, with $P_1$ holding the King in the left subtree and the Jack in the right subtree. $P_1$ learns the outcome of the deal; $P_2$ does not, $P_1$ and $P_2$, in turn, then choose to fold (left branch) or to bet their other dollar (right branch).

The payoffs shown at the terminals are for $P_1$; $P_2$’s payoffs are the negation of these values. If one player bets and the other passes, the bettor wins a net payoff of 1 (the other player’s ante). Otherwise, in the case of a showdown, the player with the stronger hand wins a net payoff of two dollars if both players bet or one dollar if both folded.

Because $P_2$ does not observe the outcome of the deal, its information is the same at $B.1$ and $B.2$. That is, $\{B.1, B.2\}$ is one of $P_2$’s information sets (These nodes are connected in Fig. 1 to indicate that they are in the same information set). When $P_1$ bets, $P_2$ knows only that the true state of the game is either $B.1$ or $B.2$; this encodes the fact that $P_2$ does not know whether $P_0$ selected branch $K_1J_2$ or $K_2J_1$.

A strategy $o_1$ for $P_k$ defines a probability for every outgoing branch from all members of $P_k$’s information sets. In Fig. 1, $P_1$ has two singleton information sets: $\{K\}$ and $\{J\}$. $P_1$ must assign probabilities such that $K_f + K_b = 1$ and $J_f + J_b = 1$. Likewise, $P_2$ must assign probabilities to outgoing branches from nodes in $\{F.1, F.2\}$ and $\{B.1, B.2\}$. Note that while $P_2$ would prefer $B_0 = 1$ in $B.1$ and $B_1 = 1$ in $B.2$, the uncertainty dictated by the rules of the game requires that this value be the same in both cases (i.e., $P_2$ views this as a single decision).

Under perfect information, the equilibrium strategy for both players would be to bet on the King and fold with the Jack; the expected value would be 0. Because of the partial observability, $P_1$ can guarantee an expected payoff of at least 1/6 by exploiting the uncertainty that it knows $P_2$ will have. The equilibrium solution requires that $P_1$ should bluff on the Jack with probability 1/3.

### Logic Description Language

The Game Description Language (GDL) can be used to describe games of perfect information (Love, Hinrichs, and Genesereth 2006). Edelkamp and Kissman have shown how GDL descriptions can be converted to a Planning Domain Description Language (PDDL) specification with only a minor extension to PDDL to support utility functions for each player at terminal nodes (Edelkamp and Kissmann 2007). Thieltscher has developed GDL-II, an extension to GDL that supports imperfect information games (Thieltscher 2010). In GDL, each player is notified of all the moves made by other players, allowing each player to completely update its internal representation of the true world state. In GDL-II, by contrast, the game description specifies an observation that each player receives for each move. Players must use their list of observations together with logical inference to deduce possible world states.

Thieltscher has shown that GDL-II can be embedded in the situation calculus, providing a reasoning system that is theoretically both sound and complete. Many practical challenges remain, however. A system might theoretically prove the formula $(\text{holds p1 king}) \lor (\text{holds p2 king})$, but it is not clear how a standard theorem prover would be able to manage analysis of the probability for each possibility. For our simple poker game, it would be dangerous for $P_2$ to assume that $(\text{holds p1 king})$ and $(\text{holds p2 king})$ are equally likely to be TRUE, given that $P_1$ bet. Rather than trying to reason about specific formulae, we propose to reason about the probability of each complete state in the information set or, equivalently, the probability of the corresponding action sequences. Also, for games with large information sets, it may be feasible only to analyze a sampled subset of the information set.

We have developed the Partially Observable Game Domain Description Language (POGDDL) as an extension of PDDL 2.2 (Edelkamp and Hoffmann 2004). Notably, this subset of the PDDL language offers support for derived predicates and numerical fluents.

We use the same extension described in (Edelkamp and Kissmann 2007) to specify payoffs for each player:

```
(:gain <typed list(variable)> <f-exp> <GD>)
```

The first parameter specifies a list of players. The second parameter is the numerical reward those players attain when
the goal condition, a logical formula given by the third parameter, is achieved in a terminal state.

Figure 2 shows a POGDDL encoding of the simplified poker game. (See Fig. 1.) Predicate declarations are omitted.

The key syntactic extension that we add to support partial observability is an observe construct, which appears in the (possibly conditional) effects clause of an action and specifies the observation each player receives for each action. Crucially, these observations may vary by player.

Figure 2 shows a POGDDL encoding of the simplified poker game from Figure 1. For $P_1$, the information set that corresponds to $o_{1:1} = \{(\text{stronger } p1)\}$, and for $o_{1:1} = \{(\text{stronger } p2)\}$, it is $\{(\text{deal-stronger } p2)\}$. These correspond to $\{K\}$ and $\{J\}$, respectively, in Fig 1.

For $P_2$, the information set for $o_{1:2} = \{(\text{stronger } ?), (\text{p1-choose pass})\}$ is $\{(\text{deal-stronger } p1), (\text{p1-choose pass})\}$ and for $o_{1:2} = \{(\text{stronger } ?), (\text{p1-choose bet})\}$, it is $\{(\text{deal-stronger } p1), (\text{p1-choose bet})\}$.

Note that in the POGDDL framework, action effects and associated observations are deterministic. To simulate stochastic effects or observations for a player’s actions, a game designer can implement a deterministic player action followed by a chance event.

Single-Point Decisions

Figure 3 shows a single-point decision procedure described in (Parker, Nau, and Subrahmanian 2005). In the original formulation, the algorithm takes a set of possible states as an input parameter. This set is computed in a game-specific way. We modify the algorithm to take instead only a list of observations; we then use our general information set algorithm to identify possible game histories.

By simulating a plausible sequence and checking which actions’ preconditions are satisfied in the resulting state, we identify possible legal moves. Then for each possible state and each legal action, we estimate the value of the successor state using some heuristic evaluation function. This may be computed as a function of the logical state variables (Kuhlmann, Dresner, and Stone 2006) or by using Monte Carlo Search methods (Kocsis and Szepesvári 2006).

The estimated values are weighted by the agent’s estimate of the probability of the corresponding game history. Some of the actions in $a_{1:t}$ may be opponent moves, so this process of estimating probabilities requires opponent modeling. A key advantage of our Information Set Generation (ISG) approach is that $P_k$ can partially simulate a possible sequence of moves up to an opponent’s decision point. Let $P_k$ be an opponent of $P_k$, and suppose that $K(S(a_{1:t-1})) = P_k$. By considering the observations that $P_k$ would have had at $S(a_{1:t-1})$, $P_k$ can generate $P_k$’s information set at that point, and then analyze that decision point from $P_k$’s perspective, in order to estimate $P(a_{1:t}|S(a_{1:t-1}))$. Note that this method provides a general approach for $P_k$ to reason about what $P_k$ knows. Implementations of this mode of reasoning has proven effective in systems designed to play specific games, such as Scrabble (Richards and Amir 2007).

Information Set Generation

Algorithm DFS-Infoset, shown in Figure 4, outlines a naive but generally applicable way for identifying sequences of actions that are consistent with a player’s observations. This straightforward approach is alluded to in (Parker, Nau, and Subrahmanian 2005) and (Russell and Wolfe 2005).

Given a list of observations $o_{1:k}$, the algorithm traverses the game tree in a depth-first manner, returning all sequences of actions for which the observations for $P_k$ would be exactly those specified by $o_{1:k}$.

At the $i$th level of recursion, the function checks that the simulated observation for $P_k$ at a node at the $i$th level of the tree is the same as the actual observation $o_{i:k}$ given in the input, and only makes a recursive call if they match.

While straightforward and generally applicable in theory, the algorithm is often prohibitively expensive.

Suppose the first move of the game is a random face-
down deal of one of \( n \) house cards, such that \( P_k \)'s observation is \( \text{deal} \) if the game tree is \( m \) levels deep, \( m \) moves later, that card is revealed to be \( c_1 \). Let the branching factor of the game tree be \( b \) for levels \( 2 \) to \( m + 1 \). Now suppose that the first branch searched at level 1 corresponds to \( \text{deal} \) at level 1. The DFS-Infoset function may potentially search a subtree of size \( b^m \) exhaustively, pruning only at level \( m + 1 \) each time it is discovered that the simulated observation (house \( c_0 \)) does not match the actual observation (house \( c_1 \)).

Such effort is both intractable and unnecessary. \( P_k \) should be able to infer from \( o_{m+1}^k \) (house \( c_1 \)) that \( a_1 = \text{deal} \), without having to search a large portion of the game tree up to depth \( m + 1 \). Systems targeted to one game tend to hard-code such domain-specific understanding.

Information Set Generation as Constraint Satisfaction

A better approach for the general case is to frame the information set problem as a constraint satisfaction problem that makes it possible to use all relevant observations to appropriately prune the search space. Algorithm CSP-InfosetSample (Figure 5) follows this approach.

```plaintext
1: function CSP-InfosetSample(SG, a^{(k)}_{1:T}, A[]) 2: if inconsistent(SG) then 3: return ∅ 4: else if complete(A) then 5: return A 6: else 7: \( t \leftarrow \text{RandomUnsetTimestep}(SG) \) 8: \( L \leftarrow \text{PlausiblyLegalMoves}(SG) \) 9: \( a \leftarrow \text{RandomElement}(L) \) 10: \( A[t] \leftarrow a \) 11: \( SG' = \text{Propagate}(SG, t, a, o^{(k)}_{1:T}) \) 12: return CSP-InfosetSample(SG', a^{(k)}_{1:T}, A)
```

Figure 5: Information Set Generation as a Constraint Satisfaction Problem.

This algorithm solves a general \( T \)-variable constraint satisfaction problem, where \( T \) is the total number of actions that have been executed by all players up to the current point in the game. For \( 1 \leq t \leq T \), the \( t \)th variable corresponds to the \( t \)th action taken, and the possible values are the possible actions in the game. A player’s information set is the set of all feasible solutions to the CSP. As written, the algorithm returns a single feasible action sequence (or \( \perp \)), if the algorithm’s non-deterministic choices lead to a conflict.) A sample of possible nodes from the information set can be obtained by running the algorithm repeatedly.\(^1\)

The basics of the backtracking search are implemented in CSP-InfosetSample. The inputs are an initialized special data structure which we call a stage graph (described in detail below), the observation sequence for the active player up to \( t \) and a list of actions \( A \) that have been executed by all players up to the current point in the game. For \( 1 \leq t \leq T \), the \( t \)th variable corresponds to the \( t \)th action taken, and the possible values are the possible actions in the game. A player’s information set is the set of all feasible solutions to the CSP. As written, the algorithm returns a single feasible action sequence (or \( \perp \)), if the algorithm’s non-deterministic choices lead to a conflict.) A sample of possible nodes from the information set can be obtained by running the algorithm repeatedly.\(^1\)

The basics of the backtracking search are implemented in CSP-InfosetSample. The inputs are an initialized special data structure which we call a stage graph (described in detail below), the observation sequence for the active player up to \( t \) and a list of actions \( A \) that have been executed by all players up to the current point in the game. For \( 1 \leq t \leq T \), the \( t \)th variable corresponds to the \( t \)th action taken, and the possible values are the possible actions in the game. A player’s information set is the set of all feasible solutions to the CSP. As written, the algorithm returns a single feasible action sequence (or \( \perp \)), if the algorithm’s non-deterministic choices lead to a conflict.) A sample of possible nodes from the information set can be obtained by running the algorithm repeatedly.\(^1\)

\(^1\)Alternatively, the algorithm can be modified to produce the full information set by changing the random selection at line 9 to iterate over all legal actions and then collect the sequences found at line 5 into a set. This may be prohibitively expensive in games with large information sets.
to the current point in the game, and an empty fixed-length vector \( A \) representing an action sequence.

At the initial invocation, each action in \( A \) is undefined. The result of the call to CSP-INFOSETSAMPLE is either a conflict, or an action sequence representing a node from the information set, or a recursive call with an additional slot in \( A \) fixed to a single concrete action. In the latter case, the time step selected is chosen at random (line 7) from among all the remaining time steps for which actions have not already been fixed by previous calls to the function. \(^2\)

Having selected a particular time step \( t \), the algorithm uses the game rules and information in the stage graph to identify all possible actions whose preconditions and effects are not inconsistent with the other actions in \( A \) that have been fixed by previous calls and whose resulting observation matches \( o_i^{(k)} \). From this set of possible actions, one is randomly selected (line 9). From the perspective of solving a CSP, this corresponds to assigning a value to a variable. Fixing a particular action \( a \) for \( A[t] \) (line 10) constrains the truth value of some predicates at time \( t-1 \) because of the preconditions of \( a \) and constrains some predicates at time \( t \) because of the effects of \( a \).\(^3\)

The **PROPAGATE** function returns an updated stage graph that encodes these ramifications (line 11). Then CSP-INFOSETSAMPLE is recursively called with the updated stage graph and action sequence. If a subset of actions selected in \( A \) lead to a situation where there are no legal actions possible for one or more other time steps, then a conflict is generated and the function returns (line 3). Otherwise, if an action is successfully assigned at every time step in \( A \), then the action sequence \( A \) corresponds to a node in the information set and is returned.

The **PROPAGATE**, **REEVALUATE**, and **INITGRAPH** functions manage the stage graph data structure that serves to reduce the branching factor of the search for solutions in the CSP by eliminating many actions from consideration. A stage graph is similar to the planning graph structures used in the AI planning (Blum and Furst 1997). For each time step, there is a stage that stores the set of potential actions that are plausible at that timestep and a vector with one of three values for each proposition in the resulting state: TRUE, FALSE, or UNKNOWN.

In function **INITGRAPH**, The 0th stage of the graph is initialized with the (known) start state; there is no action associated with the 0th stage (line 23). For timesteps \( t \) from 1 to \( T \), stage \( t \) of the graph is initialized (lines 24–36) by (1) computing the set of actions for which the resulting observation would match the observation \( o_i^{(k)} \) actually received and for which the preconditions do not conflict with the propositional values at stage \( t-1 \) (line 26); (2) setting known values for the propositions at \( t \) that are common effects of all the potential actions (line 32); (3) propagating the values from stage \( t \) for propositions that are unaffected by any of the potential actions (line 36); and (4) marking as UNKNOWN those propositions that appear in the effects of at least one plausible action but not in all (line 34). Additionally, if there are any propositions entailed by the disjunction of the preconditions of potential actions at \( t \) that were UNKNOWN at stage \( t-1 \), those values are updated and stage \( t-1 \) is marked to be re-evaluated. Specifically, the plausible action set at that stage must be recomputed. (lines 27–29).

The **PLAUSIBLEACTIONINFO** function identifies actions for a particular time step that are not inconsistent with information already stored in the stage graph. This is different than simply identifying provably legal actions because of the presence of UNKNOWN values in the stage graph.

Suppose we are considering plausible actions at time \( t \). If action \( a \) has preconditions \( p_1 \wedge p_2 \) and stage \( t \) holds that \( p_1 = \text{TRUE} \) and \( p_2 = \text{FALSE} \), then the action is definitely not plausible. However, if \( p_1 = \text{TRUE} \) and \( p_2 = \text{UNKNOWN} \), then \( a \) might be legal. We cannot prove that it is legal, but neither can we prove that it is not. We therefore say that it is “plausible” and that it does not conflict with the information in the stage graph.

Propositions (or their negations) that are entailed by the disjunction of the preconditions of all plausible actions **must** hold at the previous time step. These “definite preconditions” are returned from **PLAUSIBLEACTIONINFO** as **DefPre**. Similarly, the set of propositions that are entailed by the disjunction of the effects of possible actions **must** hold at \( t \). These “definite effects” are returned as **DefEff**. Both **DefPre** and **DefEff** include one set of known positive literals and one set of known negative literals.

Propositions that are entailed by the effects of one or more of the possible actions but not all of them are “possible effects” and are returned as **PossEff**. The **PLAUSIBLEACTIONINFO** function requires as parameters the observation at time \( t \) and the graph stages for time steps \( t-1 \) and \( t \). The function ensures that an action is excluded from the plausible set for \( t \) if and only if its precondition conflicts with known predicate truth values at \( t-1 \) or its effects conflict with known predicates at \( t \).

During the initialization of stage \( t \), it is possible to identify propositions whose truth values at \( t-1 \) are determined by **DefPre** but whose values were **UNKNOWN** during the initialization of stage \( t-1 \) (lines 27–29). Therefore, a vector of Boolean flags **Redo** is maintained that keeps track of stages whose set of plausible actions may need to be recomputed.

The **REEVALUATE** function (lines 8–20) runs a fixed point calculation that repeatedly computes plausible action sets until there are no additional propositions at any stage whose values can be changed from **UNKNOWN** to either **TRUE** or **FALSE**.

When a particular action \( a \) is fixed for time step \( t \) in CSP-INFOSETSAMPLE, the **PROPAGATE** function (lines 1–6) updates the stage graph to reflect the fact that the preconditions of \( a \) hold at \( t-1 \) and the effects of \( a \) hold at \( t \). As in stage graph initialization, the ramifications of these updates are computed by invoking **REEVALUATE**.
**Time and Space Complexity** The space taken by the stage graph is $O(|P|T)$, where $|P|$ is the number of ground propositions in the game description. The overall running time depends most crucially on the REEVALUATE function, and in particular on the number of iterations of the loop at lines 9–19. The number of iterations is bounded by the number of propositions marked as UNKNOWN during the stage graph initialization phase. Each iteration of the while loop in the REEVALUATE function (other than the last) reduces the number of UNKnOWNs by at least 1, so there are at most $|P|T$ iterations.

The PLausibleActionINFO subroutine (not explicitly shown) must identify plausible actions given the information currently in the stage graph for the current and next time steps. Identifying legal moves involves testing the satisfiability of arbitrary Boolean expressions and is therefore theoretically exponential in the size of the game description. In practice, any tree search algorithm will be practical only if the ability to identify legal moves is sufficiently computationally inexpensive so as to be used repeatedly as a subroutine. Many interesting games, including those studied in this work, satisfy this property.

The overall expense of information set generation depends on the expected number of times that CSP-InfosetSample must be called in order to obtain a viable action sequence. This efficiency varies greatly from game to game and will require additional analysis.

**Empirical Results**

We have encoded three non-trivial games in POGDDL. Racko and Battleship are popular table games; the Game of Pure Strategy comes from the game theory literature. For each game type, we varied complexity parameters (number of cards, size of game grid, etc.) to measure the performance and scalability of both DFS-Infoset and CSP-InfosetSample. For each game instance, we generated ten random positions and tested each algorithm’s ability to sample from the current player’s information set. For each position studied, each algorithm was allotted up to 30 seconds to search for nodes in the information set. (This cutoff was chosen arbitrarily, but we did not notice a significant variation in the relative strength of the algorithms when other cutoffs were used.) The decision-making procedure requested a maximum of 30 samples from each information set.

**Racko** Racko consists of a deck cards numbered 1 to $n$ and a rack for each player with $k$ ordered slots, with $n > 2k$. The cards are shuffled and then the first $2k$ cards are dealt into the players’ rack slots, such that players see only their own cards. The next card is placed face-up as the initial discard pile, and the remainder of the cards are placed face down as the draw pile. Each player’s goal is to get the cards in his rack in ascending order. On each turn, a player may exchange a card on his rack with the top card from the draw pile or the most recently discarded card. The card that is swapped out is placed face-up on the discard pile. The game ends when one player’s rack is totally ordered or when a set number of turns have been played.

Results for information set generation in Racko are shown in Table 1 (left). The table shows experimental data for instances of the game ranging in complexity from six slots per player and 30 total cards to the standard configuration of 10 slots per player and 60 total cards. Each entry in the table shows the number of information set nodes identified within the time limit. CSP-InfosetSample successfully samples information set nodes in each case, while DFS-Infoset is unable to find a single information set node in 19 of 20 games.

In the standard game, a player may have over ten billion nodes in its information set, so the ability to sample from the information set (rather than enumerate it in full) is crucial. As play proceeds and a player has opportunities to see what cards the opponent swaps out of its initial rack, the size of the information sets decreases. The propagation of the ramifications of these observations through the stage graph allows the CSP-InfosetSample algorithm to concretely identify the only plausible actions by the dealer during the rack initialization phase, which greatly improves the efficiency of search. This kind of inference would be hard-coded in a system designed specifically to play Racko. CSP-InfosetSample implements this reasoning in a generalized form.

**Battleship** In Battleship, players secretly deploy five ships on a 10x10 grid. Each ship occupies 2–5 consecutive horizontal or vertical grid spaces. Players then take turns calling out grid cells where they believe the opponent’s ships are. When a player calls out a cell, the opponent responds with ‘hit’ if the cell is occupied by a ship and ‘miss’ if it is not. The opponent is required to declare which ship was hit and must also declare ‘sunk’ if all other cells occupied by that ship have already been hit. The first player to sink all of her opponents ships wins.

We varied the number of ships in the game from one to five and the size of the grid (per side) from four cells to ten. In our experiments (Table 1, middle), CSP-InfosetSample is always successful in sampling information set nodes and in many cases produces the full information set. By contrast, the DFS algorithm finds fewer nodes in the information set and cannot find any information set nodes for many of the larger instances (more than three ships, or grids larger than 6x6).

**Game of Pure Strategy** The Game of Pure Strategy (GOPS) is described in (Luce and Raiffa 1957). A subset of a deck of cards is dealt such that each player has $n$ cards and $n$ additional cards lie in the middle. One by one, the cards in the middle are exposed. Players use the cards in their own hands to bid on the card shown. Players simultaneously declare their own bids, with the high bidder winning the card and scoring the number of points shown on the card. The player who has accumulated the most points at the end of the $n$ bidding rounds wins.

Our results for GOPS (Table 1, right) show that, once again, CSP-InfosetSample succeeds in every case, while the DFS-Infoset is unable to find any information...
Table 1:  (left) Racko. Variables: slots per player (row), total cards (column).  (middle) Battleship. Variables: ships per player (row), grid cells per side (column).  (right) Game of Pure Strategy (GOPS). Variables: bidding rounds (row), extra cards (column).

Conclusions and Future Work

The ability to reason about information sets at a logical level and to sample from them in a scalable way is necessary for general single-point decision-making. In our experiments, our constraint-based search methods far outperform existing depth-first search techniques on this task. The ability to identify game histories that are consistent with a sequence of observations makes it possible to recreate past positions from an opponent’s perspective and to reason about the opponent’s knowledge state at that position. Future work will elaborate on the opponent modeling and tree search aspects of game-play and how to combine these techniques to improve the quality of decision-making.

Acknowledgements  This work was supported by NSF IIS grant 09-17123 titled Scaling Up Inference in Dynamic Systems with Logical Structure.

References