In *Artificial Intelligence* 94(1), July 1997, pages 167-215

Representations and Solutions for Game-Theoretic Problems

Daphne Koller Avi Pfeffer
Computer Science Department
Gates Building 1A
Stanford University
Stanford, CA 94305-9010
{koller,avi}@cs.stanford.edu

April 16, 1997

Abstract

A system with multiple interacting agents (whether artificial or human) is often best analyzed using *game-theoretic* tools. Unfortunately, while the formal foundations are well-established, standard computational techniques for game-theoretic reasoning are inadequate for dealing with realistic games. This paper describes the *Gala system*, an implemented system that allows the specification and efficient solution of large imperfect information games. The system contains the first implementation of a recent algorithm, due to Koller, Megiddo, and von Stengel. Experimental results from the system demonstrate that the algorithm is exponentially faster than the standard algorithm in practice, not just in theory. It therefore allows the solution of games that are orders of magnitude larger than were previously possible. The system also provides a new declarative language for compactly and naturally representing games by their rules. As a whole, the Gala system provides the capability for automated game-theoretic analysis of complex real-world situations.

1 Introduction

When designing or analyzing a situation with multiple interacting entities, it is important to consider the (often incompatible) goals of these entities, their possible actions, and the information available to them. *Game theory* provides us with the tools to formally model such a situation as a multi-player game, to analyze it, and to prescribe 'rational' strategies to the different players.

Since real life contains many situations involving multiple interactive agents with incompatible goals, game theory has played a role in a variety of different areas. Game theory has been fundamental in economics [Aumann and Hart, 1992], both in the theoretical foundations of microeconomic theory and in more practical examples (such as the design of the 1995/6 FCC auction of wavelengths [Norton, 1995]). Game theory has also been applied in the realm of government policy, law, politics, military analysis (both strategic and tactical), biology, and more.

Clearly, situations involving multiple agents also arise in computer science applications. In some applications, the agents are a mixture of computer and human users, e.g., computer game playing, interface design, or discourse understanding. Other applications involve two or more artificial agents, e.g., network routing, load sharing and resource allocation in a distributed system, coordination of multiple robots, and information or service transactions on the internet.

The applicability of game-theoretic analysis to computer science problems has not gone unnoticed. In recent years, several such problems have been analyzed using game theoretic tools, with interesting results. Examples include the work of Franklin, Galil, and Yung [1993] on computer security, the work of Shenker [1995] on network routing protocols, and the work of Parikh [1992] on discourse understanding. In artificial intelligence, more and more researchers are turning to game theory for the theoretical foundations of multi-agent systems (see, for example, [Rosenschein and Zlotkin, 1994] and the references therein).

Despite the growing popularity of game theory as an analytic tool, there has been little work on providing effective automated tools for game-theoretic analysis. The work in this area has focused almost exclusively on solution algorithms for games of *perfect information*: games where all players have full knowledge of the current state of the world. Unfortunately, such games form a very small fraction of the class of games that concern game theory. In real life, almost all situations contain some aspects that are hidden from the players.

It is important to distinguish the lack of information from the possibility of chance moves. The former involves uncertainty about the *current* state of the world, particulary situations where different players have access to different information. The latter involves only uncertainty about the future, uncertainty which is resolved as soon as the future materializes. Both perfect and imperfect information games may involve an element of chance; examples of games from all four categories are shown in Figure 1. Most of these are popular recreational games. The inspection game is used by game theorists to model arms control inspections; it is described in Section 3.5. The OPEC game models oil pricing by oil-producing countries; it is described in Rasmusen's game theory textbook [1989].

	Perfect information	Imperfect information
No chance	Chess Go	Inspection game Battleships
Chance	Backgammon Monopoly	OPEC game Poker

Figure 1: Examples of games of various types.

As it turns out, the presence of chance elements does not necessitate major changes to the computational techniques used to solve a game. In fact, the cost of solving a perfect information game with chance moves is not substantially greater than solving a game with no chance moves. By contrast, the introduction of imperfect information greatly increases the complexity of the problem. This increase materializes even in the one player case,[1] and is even more of a problem in the multi-player case, particularly when the different players may have access to different information.

Due to the complexity (both conceptual and algorithmic) of dealing with imperfect information games, this problem has been largely ignored at the computational level. The lack of a computational infrastructure for dealing with such games has had several unfortunate consequences:

- Since game-theoretic analysis must be done manually, only small simple games can be analyzed. When faced with a more complex situation (as most real-life situations are), the decision-maker must abstract the situation and simplify it until it becomes amenable to manual analysis. As a consequence, the results of the analysis, while providing insight, can rarely be applied directly to the original problem.

- Manual game-theoretic analysis is a subtle and complicated task, which can only be performed

[1] The problem of solving a Markov decision process (MDP) is much easier than the problem of solving a partially observable Markov decision process (POMDP). See [Kaelbling *et al.*, 1996] for a survey.

by experts. Therefore, the tools provided by game theory are only available to the general public via specialized consultants.

- The lack of practical game-theoretic algorithms has prevented the use of game-theoretic decision making directly by autonomous artificial agents.

In this paper, we take a first step towards addressing this lack. We describe an implemented system, called *Gala*, which automates game-theoretic analysis for a large class of games. The system takes a description of a game, analyzes it, and outputs strategies for the different players which are game-theoretically rational for the situation described. If desired, the system can also simulate the game (playing the role of one or more of the agents), providing a picture of the different scenarios that are likely to arise.

The Gala system is composed of two main interacting pieces. The first allows large complex games to be described clearly and concisely, using a special-purpose game specification language, which we also call Gala. The Gala language mimics the way in which a large game is typically described in natural language, by presenting its rules. For example, Figure 2 presents part of a Gala specification for the game of Poker.

Given a game specification such as this, the first component of the Gala system generates the *extensive form* of a game, a natural augmentation of a game tree utilized by game theorists. "Standard" game trees, as typically used in AI game-playing systems, are inadequate for modelling real-life games, since they do not represent the agents' information state. The extensive form addresses this problem by augmenting game trees with *information sets*.

The second main component of the Gala system automatically analyzes and finds optimal strategies for games in extensive form. For games of imperfect information, the use of randomized strategies is essential in order to achieve guaranteed reasonable performance. The basic insight is that deterministic strategies can be predictable, allowing the opponent(s) to gain additional information about the state of the game.

Once randomized strategies are allowed, the existence of "optimal strategies" in imperfect information games can be proved [Nash, 1951]. In particular, this means that there exists an optimal randomized strategy for poker, in much the same way as there exists an optimal deterministic strategy for chess. Indeed, Kuhn [1950] has shown for a simplified poker game that the optimal strategy does, indeed, use randomization.

The optimal strategy has several advantages: the player cannot do better than this strategy if playing against a good opponent, and furthermore the player does not do worse even if his strategy is revealed to his opponent, i.e., the opponent gains no advantage from figuring out the player's strategy. This last feature is particularly important in the context of automated game-analysis programs, since they are often vulnerable to this form of attack: sometimes the code is accessible, and in general, since they always play the same way, their strategy can be discovered by intensive testing. Given these important benefits of randomized strategies in imperfect information games, it is somewhat surprising that none of the (very few) AI papers that deal with these games (e.g., [Blair *et al.*, 1993; Gordon, 1993; Smith and Nau, 1993]) utilize such strategies.

Clearly, none of the minimax-based solution algorithms used for finding optimal strategies in standard game trees can be adapted to the task of finding randomized strategies in imperfect information games. The Gala system provides access to a variety of solution algorithms for imperfect information games.

Game-theoretic solution algorithms can be partitioned into two classes, based on the game representation they use. Traditionally, games in extensive form have been solved by converting them to an alternative representation, called the *normal form*. Standard linear optimization routines such

```
game(poker, [

        players : [dealer, gambler],

        ...

        flow : (play_round(ante),
                deal,
                bet),

        ante:   (money($player) gets $cash,
                 %% each player gets his/her initial allocation of cash
                 pay($ante, $player, pot)),
                 %% and pays the ante into the pot

        deal:   (choose(nature,
                        (Hand1, Hand2),
                        (deal(deck, $cards, Hand1),
                         deal(deck, $cards, Hand2))),
                 %% a pair of random hands is chosen and dealt from the deck
                 reveal(gambler, myhand(Hand1)),
                 reveal(dealer, myhand(Hand2)),
                 %% each player's hand is revealed only to him
                 if(beats(Hand1, Hand2),
                    betterhand gets gambler,
                    betterhand gets dealer)),
                 %% evaluate the hands immediately, so that we only do
                 %% it once per deal, rather than at every possible showdown

        bet:    (choose(gambler, InitialBet, between(0, $money(gambler), InitialBet)),
                 %% the player chooses his bet
                 reveal(dealer, bet(gambler, InitialBet)),
                 %% reveals it to the other player
                 debt := InitialBet,
                 pay(InitialBet, gambler, pot),
                 %% and pays it into the pot
                 take_turns(next_bet)),

        next_bet:
               (choose($player, Bet, ( % meet or raise
                                       between($debt, $money($player), Bet)
                                     ; % fold
                                       ($debt>0, Bet = 0))),
                if(($debt>0, Bet = 0),
                   wins($opponent),           % fold
                   (pay(Bet, $player, pot),    % meet or raise
                    if($debt=0,
                       wins($betterhand),      % showdown
                       (reveal($opponent, bet($player, Bet)),
                        debt gets Bet - $debt))))),

        ...
```

Figure 2: Abbreviated Gala description of poker

as linear programming and linear complementarity can then be used for finding optimal strategies. Gala provides access to these normal-form solution algorithms by interfacing with a state-of-the-art game-theoretic solving system called GAMBIT [McKelvey, 1992].

Unfortunately, the normal-form conversion process incurs exponential blowup in the size of the representation, rendering this approach impractical for large games. Gala therefore implements an alternative approach, due to Koller, Megiddo and von Stengel [1994], for solving games in extensive form. The approach is based on converting the game to a different representation, called the *sequence form*. This representation is much more compact than the normal form, but supports the use of similar linear optimization algorithms. The result is solution algorithms that are exponentially faster than the normal-form based algorithms.

One of the sequence-form algorithms is implemented as part of the Gala system (others will be added in future versions of the system). We provide the first experimental results for this algorithm, comparing it to the standard algorithm (based on the normal form). Our results show that this algorithm is exponentially faster not only in theory, but also in practice. It allows us to optimally solve complex games where the tree has tens of thousands of nodes. In comparison, normal-form algorithms can rarely deal with game trees larger than 30 nodes.

By combining the ability to easily specify complex games with the algorithms capable of solving them, the Gala system provides a complete automated game-theoretic analysis tool. For example, from a Gala specification of a Poker game, as in Figure 2, the system would generate a game tree, which would then be analyzed to produce optimal strategies for Poker. To illustrate this process, consider an instantiation of a general poker game to the case where the deck consists of 8 cards (6–K), each player gets one card, and each player only has one dollar to bet. Such a game would consist of three rounds. In the first round, the gambler can either bet his dollar or pass. After hearing the gambler's bet, the dealer must decide whether to bet or pass. If the gambler passed and the dealer decides to bet, the gambler still has his dollar, and therefore gets one more opportunity to decide whether or not to bet. At that point, the game ends.

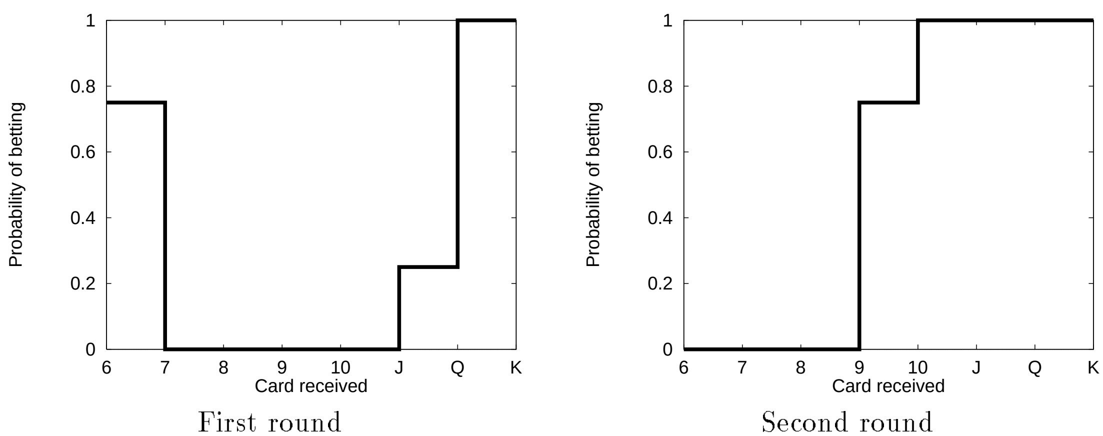


Figure 3: Gambler strategies for 8-card poker.

The optimal strategies for the gambler in this game, as obtained for the Gala system, are shown in Figure 3. They demonstrate an interesting phenomenon (first observed in a simpler game by Kuhn [1950]): Behaviors such as bluffing, that seem to arise from the psychological makeup of human players, are actually game-theoretically optimal.

These strategies were generated completely automatically by the Gala system, starting from the description of the rules of Poker, described in Figure 2. Thus, the system provides complete end-to-end functionality, where we start from a simple specification of the rules of a game, and end up with a clear and comprehensible description of optimal strategies for that game.

The remainder of this paper is structured as follows. In Section 2, we review the extensive form of a game, where game trees are augmented with information sets. In Section 3, we describe the Gala language, which supports a clear and concise specification of large and complex games by allowing a formal specification of the rules of a game. In Section 4, we review the basic solution concepts in imperfect information games, including the definition of the *Nash equilibrium* [Nash, 1951]. In Section 5 we survey both the standard game-theoretic solution algorithms based on the normal form of the game, as well as the more recent sequence form algorithms of [Koller *et al.*, 1994]. In Section 6, we show how these different ideas come together to form the Gala system, and present the first experimental results comparing the sequence form and the normal form algorithms. We conclude in Section 7 with some discussion and directions for future work. As some readers may be unfamiliar with game-theoretic concepts and techniques, the paper contains some tutorial sections that review well-known material. Section 2, Section 4 and Section 5.1 present standard concepts from game theory, while Section 5.2 reviews more recent work.

2 The extensive form

In this section we review the *extensive form* representation of a game. Readers familiar with game theory may wish to skip this section or to skim it rapidly so as to familiarize themselves with the notation used later on. The extensive form is similar to the traditional AI representation of a game as a tree. As usual, each node represents a possible state at some point in time, with the root representing the initial state. Each edge is an action which changes the state into a new one. In this paper, we restrict attention to situations that have a finite action set and end after a finite number of actions have been taken. Therefore, we consider only finite trees.

When modeling a multi-agent situation, each node is associated with a single agent, whose turn it is to choose an action. The set of edges leading out of a node are the choices available to that agent. The agent acting at a given node may also be chance or nature, in which case the edges represent random events. An agent's strategy dictates its moves at the different points in the game. In order to recommend a strategy which is rational for an agent, we must model the agent's preferences over the different possible outcomes of the situation. Therefore, we associate a vector of payoffs, one for each agent, with each leaf of the tree.

This representation of a multi-agent situation is the very familiar one, standardly used to represent game trees in AI game-playing applications. Unfortunately, it is inadequate as a model for some games, and for almost all games that model real-world situations. To understand why, consider the following simplified variant of poker, first described by Kuhn [1950]. The game has two players, each of whom initially has two dollars, and a deck containing the three cards J, Q, and K. Each player antes one dollar and is dealt one card.

Figure 4 shows half of the game tree for this game: the part of the tree corresponding to the deals (Q, J), (Q, K), and (J, K). The game consists of three rounds. In the first round, the gambler (shown in grey) can either bet an additional dollar or pass. After hearing the gambler's bet, the dealer (shown in dashed black) decides whether to bet or pass. If the gambler passed and the dealer decides to bet, the gambler still has a dollar, and therefore gets one more opportunity to decide whether or not to bet. At that point, the game ends. If both bet or both pass, the player with the highest card takes the pot; in the first case, the winning player wins two dollars, and in the second case, he or she wins one. If one player bet and the other passed, then the betting player takes the pot, thereby winning one dollar.

At first glance, Figure 4 appears to be an adequate, albeit partial, representation of this game. However, this is not the case. Note that, at the two points corresponding to the deals (Q, J) and (Q, K), the gambler has exactly the same information. So, even if the gambler would prefer to bet in the first case and pass in the second, this is not a strategy that he is capable of executing. Unfortunately, this information about the gambler's information state is encoded nowhere in this

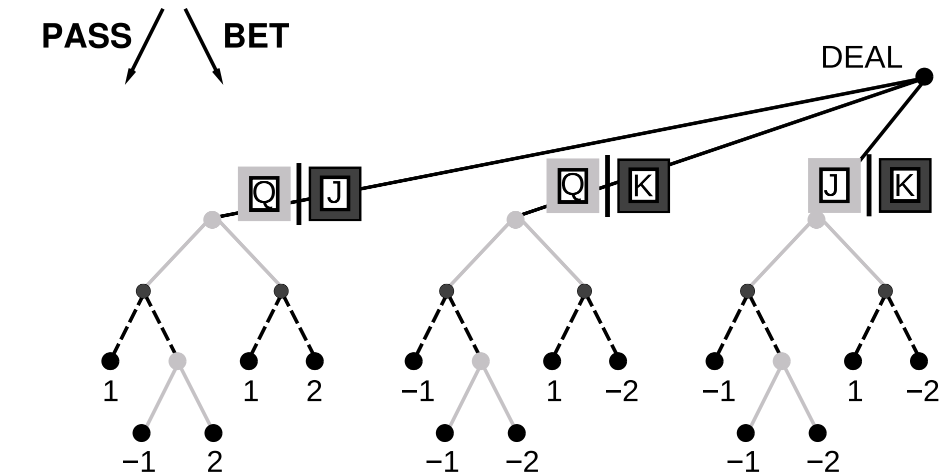


Figure 4: Naive attempt at a partial game tree for simplified poker, containing three of the six possible deals.

tree. Therefore, an algorithm that uses this tree as a model of the game cannot possibly determine that this "omniscient" strategy is not legitimate.

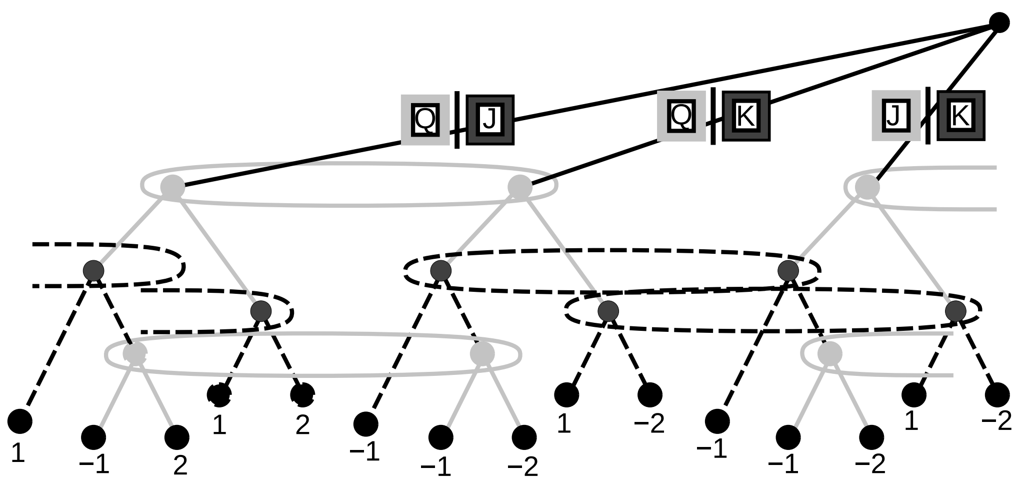


Figure 5: A partial game tree for simplified poker, containing three of the six possible deals. A move to the left corresponds to a pass, a move to the right to a bet. The information sets are drawn as ellipses; some of them extend into other parts of the tree.

Game theorists have long known that the representation of a game must encode the information states of the players. Therefore, the standard representation of a game as a tree, known as the *extensive form* of the game, also contains *information sets*. Each information set, represented graphically by an oval containing several nodes of the tree, aggregates those states of the game which are indistinguishable to the player whose turn it is to act. For example, as shown in Figure 5, the two nodes immediately following the deals (Q, J) and (Q, K) are in a single information set, which is associated with the gambler.

More precisely, let (c, d) denote the hands dealt to the two players. Initially, the gambler only knows his own card, so for each possible c, he has one information set u_c containing two nodes; each node corresponds to the two possibilities for the dealer's hand. In her turn, the dealer knows d as well as the gambler's action in the first round. Hence, she has two information sets for each d—v_d^p and v_d^b—corresponding to the gambler's previous action. Finally, the gambler has an information set u_c' in the third round. As we can see, Figure 5 is an accurate (although still partial) representation of this game.

We can now provide a formal definition of an extensive form game, as first described by Kuhn [1953]. The game is represented as a finite directed tree whose nodes denote game states. The internal nodes of the tree are of two types: decision nodes of some player k, for $k = 1, \ldots, n$, and *chance moves*. The outgoing edges at a decision node represent possible actions at that node, and have distinct labels called *choices*. A *play* denotes the path from the root to some leaf. A *move* is a choice taken on that path.

The *payoff* function h determines a payoff vector $h(p) \in \mathbb{R}^n$ for each leaf p. The kth component $h^k(p)$ of $h(p)$ is the payoff at p to player k. The relation between the payoffs to the different players is, in general, arbitrary. Thus, the interests of the players may coincide in some circumstances, and conflict in others. A *zero-sum* game models a situation where there are only two players, whose interests are strictly opposed, i.e., $h^2 = -h^1$.

The set of decision nodes is partitioned into *information sets*. Each information set u belongs to exactly one player k. Intuitively, the player cannot differentiate between different nodes in the same information set. This implies that at each node p in u, the player must have the same set C_u of *choices* (labels for the outgoing edges) at u. For simplicity, it is assumed that the choice sets C_u and C_v of any two information sets u and v are disjoint. For example, in Figure 5, the dealer might have the choices P_d^p and B_d^p in v_d^p and P_d^b and B_d^b in v_d^b, where P indicates a pass and B indicates a bet. In games with *perfect information*, where the players always know the current state of the game, the information sets of all players are always single nodes.

Throughout this paper, we restrict attention, as in most of the game-theory literature, to games where all of the players have *perfect recall*. Informally, a player has perfect recall if she never forgets her previous actions or any fact that she knows. Formally:

Definition 2.1: Player k is said to have perfect recall if for each of her information sets u, and any two nodes $p, q \in u$, the sequence of player k choices on the path to p is precisely the same as the sequence of player k choices on the path to q. ∎

Thus, p and q can only be in the same information set if the entire history of events leading to u and v is identical *according to player k's point of view*. Had there been some difference between the two histories, player k would have had to forget that difference in order to place p and q in the same information set. Recall that we assumed that actions at different information sets have different labels. For example, the first-round passing action when the gambler has a Queen would be labelled P_Q, while the same action when the gambler has a Jack would be labelled P_J. Thus, two histories that pass through different information sets must be different. Hence, the definition of perfect recall guarantees that two histories that passed through different information sets can never "converge" into the same information set; i.e., the player can never "forget" information she once had.

3 The Gala language

The extensive form captures the low-level dynamics of the game: the choice points, the choices available, and the sets of nodes among which a player cannot distinguish. However, it captures none of the underlying structure, e.g., the fact that the choices of a Poker player are determined by by the amount of money he has left. In fact, within the extensive form, we cannot even represent the amount of money the player has at various points in the game.

Since we cannot encode this information, we cannot utilize it to compactly specify the game tree (e.g., by observing that the available moves in the betting round of Poker are identical regardless of the hands the players are dealt). The game tree must be written down explicitly: every move and every information set. While it is possible, although far from trivial, to write down a correct game tree with 30 nodes, it is unrealistic to expect a person to manually construct a game tree with thousands of nodes. It is even less reasonable to expect the resulting game tree to be correct.

In this section, we present a language, called Gala (for GAme LAnguage),[2] that supports clear and concise specifications of large, complex games. Gala mimics the way in which a large game is typically described in natural language, by presenting its *rules*.

The idea of using a declarative language to specify games was proposed by Pell [1992]. He utilizes it to specify *symmetric chess-like games*—a class of two-player perfect-information board games. Our language is much more general, and can be used to represent a very wide class of games, in particular: one-player, two-player and multi-player games; games where the outcomes are arbitrary payoffs; and games with either perfect or imperfect information. As we will show, the expressive power of Gala allows for clear and concise game descriptions, that resemble, at a high-level, a natural language representation of the rules of the game.

Gala describes a game as a branching program, where each possible execution of the program corresponds to a possible play of the game, as determined by the players' actions and by the outcomes of the chance moves. The program first specifies an initial state, including the number of players. It then lists a sequence of game steps, whose execution causes the game state and the players' information state to change over the course of the game.

A Gala program consists of a set of declarations. These declarations fall into two categories. Declarations in the first category describe entities in the game. In a poker program they might say that the game involves two players named `dealer` and `gambler`, a deck of cards, and so on. The other category contains declarations that describe the sequence of events that take place during the game. In Figure 2, the declarations for `flow`, `ante`, `deal`, `bet` and `next_bet` all fall into this category. Each such declaration contains a sequence of Gala statements. Gala provides flow control statements similar to those found in many programming languages.

Three important Gala statements are `choose`, `reveal` and `outcome`. These are the basic building blocks for defining the course of a game. A `choose` statement defines a choice point in the game; it indicates the set of choices available at that point, and whether the choice is made by a player or at random. A `reveal` statement changes the information state of a player. The `payoff` statements determine the final outcome of the game.

In addition, Gala allows a game state to be explicitly specified, maintained through the game, and utilized in determining the available choices. For example, in the Poker program of Figure 2, we maintain the amount of money available to the players via the variables `$money(gambler)` and `$money(dealer)`. These are updated by Gala commands as the betting progresses, and are accessed to determine the set of possible amounts that a player can choose to bet.

The game manipulation statements range from simple variable manipulation, to ubiquitous concepts such as combinations of objects, to special-purpose libraries for dealing with certain types of objects (e.g., a rectilinear board). These statements do not directly affect the structure of the game; they do not induce choice points or specify information sets. Rather, they provide bookkeeping for the game state, which in turns supports the specification of the game tree.

[2]We use the name "Gala" to refer both to the Gala language and to the Gala system described in Section 6, which contains an implementation of the Gala language and algorithms for solving games specified in this language.

It is worth discussing the relationship between Gala and Prolog. Gala is embedded in Prolog, and uses Prolog syntax for many of its constructs. The Gala interpreter is written in Prolog, and uses the Prolog proof generator to discover all possible plays of a game. In addition, Prolog predicates can be called from within a Gala program, allowing conditions on the game state to be expressed as declarative queries. We do require that Prolog predicates called from within Gala be deterministic, so that all non-determinism in the game is defined by `choose` statements. In general, the code describing a game in Gala consists of two parts: the Gala program itself, and auxiliary Prolog predicates used by the Gala program.

We begin by discussing the underlying semantics of a Gala program. We then discuss the various types of Gala statements: basic primitives, flow control, and game state manipulation. Finally, we demonstrate how the various features of the language come together in specifying complete games.

3.1 Gala semantics

As we have discussed, game trees are not sufficiently expressive for ascribing precise semantics to a Gala program, since they do not allow us to discuss the state of the game and the information state of the players. We therefore consider a somewhat more expressive framework, based on the work of Fagin, Halpern, Moses and Vardi [1995] for reasoning about multi-agent systems.

We specify the behavior of a Gala program via its *execution tree*. Each node in the execution tree is a possible state that can occur in the game, with the root corresponding to the initial state of the program, and the children of each node corresponding to those states that can follow the state encoded by that node.

As in [Fagin *et al.*, 1995], a *global state* describes both the "external" state of the game and the internal states of the different players. In an N-player game, a global state is therefore an $(N+1)$-tuple of the form $\langle s_e, s_1, \ldots, s_N \rangle$ where s_e is the *environment state* and s_k is the *local state* of player k. The player's local state encodes all of the information available to the player, so that the player cannot distinguish between global states where her local state is the same.

The environment state corresponds to the program state of the Gala program, in the same sense as in any other programming language. It consists of two main parts:

- A *gameflow*, which is a Gala code fragment specifying the sequence of steps that remains to be executed. The gameflow serves the role of a program counter. In the description of a state, the gameflow is presented in quotes, and we use the letters φ and ψ to indicate an arbitrary (possibly empty) sequence of gameflow steps.
- A *gamestate*, consisting of a set of current assignments (or bindings) to the program variables. Gala allows both Prolog-style variables, whose bindings are progressively refined during the course of execution, and variables whose values can be changed at will, as in traditional programming languages. We use $\mathcal{V}$ to indicate a set of bindings for all the variables.

The local state for each player describes the agent's mental state. It consists of the player's memories, represented as an ordered list of facts (Prolog terms) known to the player. We use $\mathcal{F}_k$ to denote the list of facts known to player k.

The execution tree is a directed tree whose vertices are labeled with global states. The successors of a state are the possible states resulting from executing the first command in the gameflow at that state. Some transitions are deterministic, corresponding to the mechanics of the game (e.g., transferring the ante to the pot in the beginning of a poker game). In this case, the vertex will have only a single successor. In other cases, the transition corresponds to a choice point in the game, resulting in vertices with several successors. The choice point can be due either to a chance event (e.g., a roll of the dice) or to an action of one of the players (e.g., a decision to pass or bet). Different choices of action will induce different possible plays of the game.

3.2 Basic Gala

3.2.1 Choice points

The `choose` statement has the format `choose(Player, Template, Constraint)`, where `Template` is a Prolog expression containing unbound variables, and `Constraint` is a Prolog predicate. Any instantiation of `Template` satisfying `Constraint` is an action that can be taken in the current state.[3] For example, the `choose(gambler, InitialBet, between(0, $money(gambler), Bet))` statement in Figure 2 specifies the set of possible actions to be any numerical value for the variable `Bet` which is between 0 and the amount of money that the gambler has. (See the semantics of Gala variables in Section 3.3.1.) `Player` is either the name of a player, or `nature(`μ`)`, where μ is a probability distribution over the instantiations of `Template` satisfying `Constraint`.[4]

The power of `choose` lies in the fact that the available actions in a state are the answers to a query, and do not have to be encoded explicitly. The answers to the query depend on the current game state, and vary according to context. For example, the available bets in the statement above depend on the current value of the variable `$money(gambler)`, as described in the gamestate part of the global state. Thus, the same choose statement can be used throughout the betting phase.

More formally, let

$$\langle(\text{``}\texttt{choose(Player,Template,Constraint)},\ \psi\text{''}, \mathcal{V}), \mathcal{F}_1, \ldots, \mathcal{F}_k, \ldots, \mathcal{F}_N\rangle,$$

be the current state, and let $\mathcal{I}$ be the set of instantiations of `Template` that satisfy the prolog constraint `Constraint` and are consistent with $\mathcal{V}$. For each $I \in \mathcal{I}$, let $\mathcal{V}/I$ denote the refinement of the bindings of Prolog variables in $\mathcal{V}$ by the bindings in I.

We cannot transition directly from this state to the state after the choice is made, since, at this point, the player might not know the set of actions available to her. (The information may not be in her local state.) We therefore implement the statement using two transitions. Formally, when the interpretation of `Player` is some player $k \in \{1, \ldots, N\}$, then a deterministic transition is made to the state:

$$\langle(\text{``}\texttt{choosing(}k\texttt{, }\mathcal{I}\texttt{)},\ \psi\text{''}, \mathcal{V}), \mathcal{F}_1, \ldots, (\mathcal{F}_k \circ \texttt{choosing(}\mathcal{I}\texttt{)}), \ldots, \mathcal{F}_N\rangle\ .$$

The statement `choosing` is not a statement in the Gala language; it is just a placeholder for the intermediate state. Note that the addition of the fact `choosing(`$\mathcal{I}$`)` to $\mathcal{F}_k$ forces the player to distinguish between states where she had different choice sets.

This new state has multiple successor states, corresponding to all possible instantiations $I \in \mathcal{I}$. Each such state has the form

$$\langle(\text{``}\psi\text{''}, \mathcal{V}/I), \mathcal{F}_1, \ldots, (\mathcal{F}_k \circ \texttt{choosing(}\mathcal{I}\texttt{)} \circ \texttt{chose(I)}), \ldots, \mathcal{F}_n\rangle\ ,$$

where $\mathcal{V}/I$ is the modification of $\mathcal{V}$ according to the bindings in I. (See the specification of variables below.) Note that the fact `chose(`I`)` is added to the player's local state, enforcing the requirement (implied by the perfect recall assumption) that a player always remembers her own actions.

For the case of chance events, where `Player` is `nature(`μ`)`, then we first transition to the state

$$\langle(\text{``}\texttt{choosing(nature(}\mu\texttt{), }\mathcal{I}\texttt{)},\ \psi\text{''}, \mathcal{V}), \mathcal{F}_1, \ldots, \mathcal{F}_N\rangle\ .$$

[3] We assume that in a correct Gala program, this set of instantiations is finite and will be computed in Prolog in a finite amount of time.

[4] μ can be omitted, in which case it defaults to the uniform distribution.

The set of successors for this state are all those of the form

$$\langle (\text{"}\psi\text{"}, \mathcal{V}/I), \mathcal{F}_1, \ldots, \mathcal{F}_n \rangle \ ,$$

such that $I \in \mathcal{I}$. Each of these successor states is reached with probability $\mu(I)$.

The `choose` statement is the only way to represent choice points in the game, whether the choice is due to a chance move or to a move by one of the players. Thus, it is the only statement to allow multiple successor states in the execution tree.

3.2.2 Information states

The `reveal` statement has the format `reveal(Player, Fact)`, where `Player` is any player and `Fact` is any Prolog term. Intuitively, the fact `Fact` is added to the player's local state, allowing the player to distinguish runs where `Fact` was observed from runs where it was not. `Fact` is interpreted by the Prolog system, using the set of bindings in $\mathcal{V}$.

More formally, let

$$\langle (\text{"}\texttt{reveal(Player,Fact)},\ \psi\text{"}, \mathcal{V}), \mathcal{F}_1, \ldots, \mathcal{F}_k, \ldots, \mathcal{F}_N \rangle,$$

be the current state, and let k be the interpretation of `Player`. Then the next state is:

$$\langle (\text{"}\psi\text{"}, \mathcal{V}), \mathcal{F}_1, \ldots, \mathcal{F}_k \circ \texttt{Fact}_{\mathcal{V}}, \ldots, \mathcal{F}_N \rangle.$$

where $\texttt{Fact}_{\mathcal{V}}$ is the Prolog term resulting from interpreting `Fact` using the bindings in $\mathcal{V}$.[5]

For example, in the Gala program of Figure 2, each player's $\mathcal{F}_k$ starts out as the empty list. After the execution of the statements `reveal(gambler, Hand1)` and `reveal(dealer, Hand2)`, each $\mathcal{F}_k$ contains only the Prolog fact `myhand(Hand)`, for the appropriate value of `Hand`; for example, in games where each player gets one card, this may be `myhand([J])`. As the betting progresses, each $\mathcal{F}_k$ is augmented with facts that denote his own moves and the moves of the opponent. For example, after two rounds of betting, the gambler's local state may be the list: `[myhand([J]), choosing([0,1,2]), chose(1), bet(dealer, 0)]`.

Note that the facts known by a player are ordered, so a player can differentiate between runs in which the same facts were revealed in different orders. This is important, because perfect recall (see Definition 2.1) requires that a player distinguish between runs in which a fact was revealed before or after choosing an action. We fulfil this requirement by adding `chose(Action)` to the list of facts known to the player at the time the action is chosen. Since `choose` and `reveal` are the only Gala statements that change a player's local state, and they always append information to the facts already known, perfect recall is maintained.

3.2.3 Payoffs

The format of the `payoff` statement is `payoff(Player, Amount)`. It is interpreted to mean that the amount `Amount` is added to `Player`'s payoff to date. Thus, the player's actual payoff at the end of the game is the amount she has accumulated throughout the game. The cumulative nature of the `payoff` command allows it to be used in contexts where the game may be very long or infinite. In such games, we often want to consider a prefix of the game, and it is useful to be able to have a record of the player's payoffs to date.

[5] `Fact` may also contain uninstantiated variables, but their names are ignored in this interpretation process. Given any set of bindings $\mathcal{V}$, a Prolog term `Fact` has a unique structure $\texttt{Fact}_{\mathcal{V}}$ that ignores the names of variables, and it is this structure that is added to the player's information state.

To implement the `payoff` command, we maintain a special set of variables $h_1, \ldots, h_N$ in the global state. Formally, let the current state be

$$\langle(\text{“}\texttt{payoff(Player,Amount)},\ \psi\text{”}, \mathcal{V}), \mathcal{F}_1, \ldots, \mathcal{F}_k, \ldots, \mathcal{F}_N\rangle.$$

Let k be the interpretation of `Player`, and p be the interpretation of `Amount`. Then the next state is:

$$\langle(\text{“}\psi\text{”}, \mathcal{V}[h_k \leftarrow h_k + p]), \mathcal{F}_1, \ldots, \mathcal{F}_k, \ldots, \mathcal{F}_N\rangle.$$

Note that we do not assume that the payoff is revealed to the player.

3.2.4 Basic flow control

Gala provides several flow control statements that can be used to guide the progress of the game. The most basic ones are concatenation, termination and conditional statements.

Concatenation is straightforward. If φ and ψ are gameflows, then (φ, ψ) is simply the concatenation of the steps in φ and ψ. The semantics for concatenation is derived from the semantics for other statements, as demonstrated above.

The game terminates whenever the gameflow in a state is empty, or an `end` statement is reached. The payoff h_k accumulated up to that point is then allocated to player k. Since in many games a payoff is assigned to all the players when the game terminates, Gala provides an `outcome` statement that is defined in terms of `payoff` and `end`. The statement takes a vector V as its argument, assigns payoff V_k to the kth player, and terminates the game.

Conditionals have the form `if(Condition,` φ_1`,` φ_2`)` where `Condition` is a Prolog predicate and φ_1, φ_2 are gameflows. If `Condition` can be satisfied given the set of bindings $\mathcal{V}$ in a state,[6] the flow continues with φ_1, otherwise it continues with φ_2. More formally, let the current state be

$$((\text{“}\texttt{if(Condition,}\ \varphi_1\texttt{,}\ \varphi_2\texttt{)},\ \psi\text{”}, \mathcal{V}), \mathcal{F}_1, \ldots, \mathcal{F}_N).$$

If `Condition` can be satisfied given $\mathcal{V}$, then the next state is

$$((\text{“}\varphi_1, \psi\text{”}, \mathcal{V}), \mathcal{F}_1, \ldots, \mathcal{F}_N),$$

otherwise, it is

$$((\text{“}\varphi_2, \psi\text{”}, \mathcal{V}), \mathcal{F}_1, \ldots, \mathcal{F}_N),$$

As a shorthand one can use `if(Condition,` φ`)` instead of `if(Condition,` φ`,` ϵ`)`, where ϵ is the empty sequence of commands.

For example, in our Poker example, `beats(Hand1, Hand2)` is a Prolog predicate (defined in Figure 8) used in the argument of an `if` statement. The outcome of this predicate thereby determines the assignment to the variable `betterhand`.

3.2.5 Game trees

These primitives are the basic tools for specifying game trees within the Gala language. Any extensive form game can be written as a Gala program utilizing only the features described above. A leaf can be translated into an `outcome` statement with an appropriate vector of payoffs. For an internal node p, suppose player k has to choose one of the actions $c_1, \ldots, c_\ell$, and that p is in information set u. Assume (by induction) that we have already defined gameflows $\psi_1, \ldots, \psi_\ell$ for the subtrees rooted at the correspoding children of p. Then the subtree rooted at p can be translated into:[7]

[6]In a correct Gala program, the evaluation of `Condition` must always terminate.

[7]In this program, [...] is used as a set constructor, and MEMBER tests for membership in the resulting set.

```
reveal(k, u),
choose(k, C, member(C, [c1, ..., cℓ])),
if(C=c1,
   ψ1,
   if(C=c2,
      ...,
      if(C=cℓ−1,
         ψℓ−1,
         ψℓ)···)) .
```

Conversely, the basic primitives `choose`, `reveal`, and `payoff` are the only Gala commands that directly influence the construction of the game tree from a Gala program. We will discuss this further at the end of this section.

3.3 Maintaining game state

As we mentioned, the power of the `choose` statement derives from the fact that the player's set of choices does not have to be explicitly listed in the program. Rather, this set can depend on the current state of the game. Gala provides an extensive suite of commands for accessing and manipulating the game state.

3.3.1 Variables

As in a traditional programming language, the state is maintained via a set of variables. Some of these are standard Prolog variables, which can be accessed in any of the statements described above. The bindings for the Prolog variables are maintained as part of the set of bindings $\mathcal{V}$. They can be changed as a consequences of some of the statements described above (e.g., as a consequence of a particular choice in a `choose` statement).

It is often convenient to model the changing game state in the same way as the changing state of traditional programming languages, using variables whose values can be assigned and changed at will. For this reason, it helps to provide *Gala variables* in addition to the Prolog variables. Gala variables are similar to variables encountered in many programming languages: they can be assigned values at will, without requiring (as for Prolog variables) that the new value be consistent with the old value.

The two types of variables are normally used in different ways. Prolog variables are useful for storing local information and connecting statements within a gameflow, while Gala variables can maintain global information that is used throughout the course of the game. For example, consider the following code fragment:

```
choose(a, Number, between(1, 10, Number)),
Parity is Number mod 2,
reveal(b, Parity),
choose(b, Guess, between(1, 10, Guess)),
if(Guess = Number,
    score gets $score + Number)
```

This fragment specifies that player a chooses any number between 1 and 10 and reveals only its parity to player b. Player b then tries to guess the number, and gets the number's value added to his score if he guesses right. The Prolog variables `Number`, `Parity` and `Guess` are used to connect the statements together and relate the numbers to each other. The Gala variable `score`, in contrast, maintains the score that is accumulated throughout the course of the game.

Gala variables can be directly referenced within a gameflow by putting a "$" in front of them. They can be instantiated within a gameflow using the gala command `gets`, which is analogous to `is` in Prolog. Gala variables can be converted to and from Prolog variables using `gala_val(GalaVar, PrologVar)`, and `gala_set(GalaVar, Value)`. These statements allow Prolog predicates outside the Gala program to access and modify the game state. The semantics for setting and referencing Gala variables are obvious: setting a variable causes a transition to a state in which the new binding is appended to the list, and a reference to a variable is replaced by the value in its most recent binding.

In Figure 2, the only Prolog variables are `Hand1` , `Hand2`, and `Bet`. They serve precisely the role described above, of making a short-term connection between the outcome of the `choose` statement with some additional statements. For example, we use `Bet` in the statement revealing information to the opponent, and also to update the value of `debt`. The use of the Gala variable `debt` allows information to be passed between different calls to `next_bet`.

3.3.2 More complex game states

The different types of variables provide the basic facilities for storing, manipulating, and accessing various aspects of the game state. Using variables, Gala provides a shorthand notation for concepts that occur ubiquitously in games. These include locations and their contents, pieces and their movement patterns, and resources that change hands, such as money. For example, Gala allows the user to utilize statements of the form `move(queen(white), (d,1), (d,8))` or `pay(gambler, pot, Bet)`.

On a more abstract level, we have observed that certain structures and combinations appear in many different games. While these usually involve sets in one way or another, they come in many flavors. For example, a flush in poker is a set of five cards sharing a common property; a straight, on the other hand, is a sequence of cards in which successive elements bear a relation to one another; a full house is a partition into equivalence classes based on rank in which the classes are of a specific size.

The Prolog language provides a few predicates that describe sets and subsets. We have supplemented these with various predicates that make it easy to describe many of the combinations occuring in games. For example, `chain(Predicate, Set)` determines whether `Set` is a sequence of objects in which successive elements are related by `Predicate`, while `partition(Relation, Set, Classes)` partitions `Set` into equivalence `Classes` defined by the equivalence relation `Relation`.

The following example, using a more elaborate function provided in the same library, shows how we can concisely test for all types of poker hand except flushes and straights:

```
partition_profile(match_rank, Hand, Profile),
associate(Profile, HandType,
    [([4, 1], four_of_a_kind), ([3, 2], full_house),
     ([3, 1, 1], three_of_a_kind), ([2, 2, 1], two_pairs),
     ([2, 1, 1, 1], one_pair), ([1, 1, 1, 1, 1], nothing)])
```

The predicate `partition_profile` takes three arguments: a set—in this case `Hand`; an equivalence relation—in this case `match_rank`; and a list of numbers—`Profile`. The predicate is true precisely when the list of numbers is a profile of the partition defined by the equivalence relation, where a profile is a list of sizes of partition cells in non-increasing order. For example, if `Hand` is [9♡, 6♣, 9♠, 6♡, 6♢], then `Profile` must be $[3, 2]$. The profile $[3, 2]$ is then associated with the class `full_house`.

Building on this library of predicates for describing combinations, Gala also provides libraries with more specific functionality that is common to a certain class of games. These include games played on a grid (such as chess or tic-tac-toe), playing cards, dice, and so on. For example, if a game is declared to include a grid-board object, a range of predicates that apply specifically to a rectilinear board become available. One such predicate is `straight-line`, which tests for a sequence of squares in a straight line, all of which satisfy a certain property (such as an open file in chess or three-in-a-row in tic-tac-toe). It is defined in terms of the `chain` predicate. In general, high-level predicates are typically very easy to define in terms of the intermediate level concepts, so that adding a module for a new class of games requires little effort.

3.4 Advanced flow control

One of the primary advantages of representing a game as a Gala program is the ability to utilize the same code to encode structures that repeat again and again throughout the game. For example, as we discussed above, the rule for specifying the set of legal bets can be used in different subtrees of the poker game, as well as in different bets within the same subtree.

Gala provides a variety of repetition statements, the most basic of which is the `while`. It has the form `while(Condition,` φ`)`, and is interpreted by translation into an `if` statement. That is, if the current state is

$$\langle(\text{``}\texttt{while(Condition, }\varphi\texttt{), }\psi\text{''}, \mathcal{V}), \mathcal{F}_1, \ldots, \mathcal{F}_N\rangle,$$

then the following state would be:

$$\langle(\text{``}\texttt{if(Condition, (}\varphi\texttt{, while(Condition, }\varphi\texttt{))), }\psi\text{''}, \mathcal{V}), \mathcal{F}_1, \ldots, \mathcal{F}_N\rangle.$$

Gala also provides a `repeat` statement that is defined in terms of `while` and provides fancier functionality. Its form is `repeat(Flow, Condition`$_1$`, ..., Condition`$_n$`)`. Each `Condition`$_i$ can take one of two forms: `unless(Predicate)` or `until(Predicate)`. The set of statements in `Flow` is executed repeatedly until one of the predicates becomes true; `unless` conditions are tested before an iteration, while `until` conditions are tested after an iteration.

Another feature adapted from traditional programming languages is the *defined gameflow*, which is analogous to a defined procedure. As usual, it consists of a head and a body; the head consists of a name and a list of formal parameters, while the body is a gameflow. A defined gameflow is listed as a separate section in the specification of a Gala program. For example, in our poker example, `deal`, `bet`, `first_bet` and `next_bet` are all defined gameflows.

A call to a defined gameflow is interpreted in the obvious way, with the parameters passed by value. If the gameflow in a game state begins with a call to a defined gameflow, the values of the actual parameters are substituted for the formal parameters in the body of the defined gameflow, and this body is substituted for the call to the defined gameflow in the game state. All Prolog variables within the body of a defined gameflow are local, so different calls to the gameflow do not cause binding conflicts.

Finally, Gala provides functionality that allows for players to be treated uniformly. This allows for games where all the players (or a subgroup of players) take the same sequence of actions.

One such command is `play_round(Flow)`, where `Flow` is a gameflow. In addition to executing `Flow` once for each player, this also defines a special Gala variable "player" which takes as value each of the players in turn. Thus if `Flow` contains a step of the form `choose($player,Action, Constraint)`,[8] each player will make a choice in turn. For example, in the poker program, `play_round(ante)` has the `ante` flow called twice, with `$player` first instantiated to `dealer` and then to `gambler`.

[8]The dollar sign indicates that a Gala variable is being referenced; see Section 3.3.

In addition to the `player` variable, the variable `opponent` is also defined in a two player game, and `lho` and `rho` (for left- and right-hand opponent, respectively) are defined in multiplayer games. There is also a variant of `play_round` which allows a subset of the players to play the round, perhaps in a different order from the default order.

Gala also allows a temporally extended version of `play_round`, in which the players take turns performing a sequence of actions until some condition is true. The statement takes the form `take_turns(Flow, Condition`$_1$`, ..., Condition`$_n$`)`. It is essentially the same as `repeat(Flow, (Condition`$_1$`, ..., Condition`$_n$`)`, with the `player` variable and its relatives being defined in the same way as for `play_round`.[9] For example, in the poker program, the `take_turns` construct is used to describe repeated betting by the players in turn. This terminates either when `wins()` is called within the `next_bet` flow, or when someone meets, in which case `$debt=0`.

3.5 Two complete Gala programs

In this section, we describe how the various features of the Gala language are combined to form a Gala program. As an example, Figure 6 shows a complete listing of a simple *inspection game*, which has received significant attention in the game theory community as a model of on-site inspections for arms control treaties [Avenhaus *et al.*, 1995].

The inspection game involves two players: a violator, who wants to commit some treaty-violating act (such as nuclear testing), and an inspector who wants to prevent the act. The game takes place over n stages. In each period, the violator chooses whether or not to violate, and the inspector chooses whether or not to inspect. Unfortunately, by the terms of the treaty the inspector may only inspect l times over the n stages, where $l < n$. If at any stage the violator violates without being caught, the violator wins. If the violator is caught, or if he does not attempt a violation throughout the game, the inspector wins. If at any point the number of remaining inspections is 0, we assume that the violator will succesfully violate at the next stage. If the number of inspections remaining equals the number of stages remaining, we assume the inspector will inspect in every subsequent stage, thereby ensuring victory. The game terminates as soon as either of these assumptions becomes valid, because the winner is then known.

A Gala program consists of a statement `game(GameName, DeclarationList)`, together with a set of auxiliary Prolog predicates. Each declaration in `DeclarationList` has the form `Name : Definition`. Some declarations define gameflows, while others specify other aspects of the game.

The `players` declaration lists the players in the game, in the order used by default for constructs such as `play_round` and `take_turns`. The `variables` declaration declares Gala variables used in the program. The `params` declaration illustrates a powerful feature of Gala. A Gala program can be parameterized, so that it defines a family of games. The program shown defines the inspection game for any number of stages and inspections. In this program, the parameter values are used only to declare the initial values of Gala variables. In general, the parameters can also appear directly inside a gameflow, but they are not part of the game state, and cannot be modified during a game.

The `flow` declaration defines the gameflow for the entire game; it must appear in every game description. (It is analogous to the `main` function in a C program.) Two other gameflows are also defined in this program: `stage` and `determine_outcome`. The content of every gameflow is a sequence of statements, as defined in the previous sections. In this case, the primary construct in the main

[9] This is not the same as `repeat(play_round(Flow), Condition`$_1$`, ..., Condition`$_n$`)` since in this case, `Condition` is checked only between complete rounds, whereas in the `take_turns` statement, `Condition` is also checked between the turns of the different players.

```
game(inspection, [

        players : [inspector, violator],

        params : [stages, inspections],

        variables : [stages_remaining = $stages,
                     inspections_remaining = $inspections,
                     violate,
                     inspect],

        flow :  (repeat(stage, unless(no_more_inspections),
                              unless(no_more_stages),
                              until(violation)),
                 determine_outcome),

        stage :
                (choose(violator, X, (X = yes ; X = no)),
                 choose(inspector, Y, (Y = yes ; Y = no)),
                 reveal(violator, Y),
                 violate gets X,
                 inspect gets Y,
                 if($inspect = yes,
                    inspections_remaining gets $inspections_remaining - 1),
                 stages_remaining gets $stages_remaining - 1),

        determine_outcome :
                if(no_more_inspections,
                   outcome([-1, 1]),
                   if(no_more_stages,
                      outcome([1, -1]),
                      if($inspect = yes,
                         outcome([1, -1]),
                         outcome([-1, 1])))) ]).

no_more_inspections :-
        gala_val(inspections_remaining, 0).

no_more_stages :-
        gala_val(stages_remaining, S),
        gala_val(inspections_remaining, R),
        S =< R.

violation :-
        gala_val(violate, yes).
```

Figure 6: A Gala description of inspection games.

flow is the repeat statement: `stage` is repeated until one of the termination conditions is met. When that happens, the outcome of the game is determined. The termination conditions are defined in auxiliary Prolog predicates.

Each `stage` consists of a choice by both players. The violator chooses whether or not to violate, while the inspector chooses whether or not to inspect. After they make their choices, the inspector's decision is revealed to the violator, but the violator's decision is not revealed to the inspector. The remainder of the stage consists of bookkeeping, updating the values of the Gala variables `violate`, `inspect`, `inspections_remaining` and `stages_remaining`.

As a final example, Figures 7 and 8 show the complete code for the poker game from Figure 2. Approximately half the code is the Gala program, while the rest is Prolog code for evaluating and

```
game(poker, [

        players : [dealer, gambler],

        params : [suits, ranks, cards, cash, ante = 1],

        objects : [deck : $suits * $ranks,
                   money : [dealer, gambler, pot]],

        variables : [winner, debt],

        flow : (play_round(ante),
                deal,
                bet),

        ante:   (money($player) gets $cash,
                 pay($ante, $player, pot)),

        deal:   (choose(nature,
                        (Hand1, Hand2),
                        (deal(deck, $cards, Hand1),
                         deal(deck, $cards, Hand2))),
                 %% a pair of random hands is chosen and dealt from the deck
                 reveal(gambler, myhand(Hand1)),
                 reveal(dealer, myhand(Hand2)),
                 %% each player's hand is revealed only to him
                 if(beats(Hand1, Hand2),
                    betterhand gets gambler,
                    betterhand gets dealer)),
                 %% evaluate the hands immediately, so that we only do
                 %% it once per deal, rather than at every possible showdown

        bet:    (choose(gambler, InitialBet, between(0, $money(gambler), InitialBet)),
                 %% the player chooses his bet
                 reveal(dealer, bet(gambler, InitialBet)),
                 %% reveals it to the other player
                 debt := InitialBet,
                 pay(InitialBet, gambler, pot),
                 %% and pays it into the pot
                 take_turns(next_bet)),

        next_bet:
               (choose($player, Bet, ( % meet or raise
                                       between($debt, $money($player), Bet)
                                     ; % fold
                                       ($debt>0, Bet = 0))),
                if(($debt>0, Bet = 0),
                   wins($opponent),           % fold
                   (pay(Bet, $player, pot),   % meet or raise
                    if($debt=0,
                       wins($betterhand),     % showdown
                       (reveal($opponent, bet($player, Bet)),
                        debt gets Bet - $debt))))),

        wins(player):
               (Winnings is $money($player) + $money(pot) - $cash,
                Losings is - Winnings
                if($player = dealer,
                   outcome([Winnings, Losings]),
                   outcome([Losings, Winnings])) ]).
```

Figure 7: Gala description of poker: Gala program

```
beats(Hand1, Hand2) :-
        predsort(compare_ranks, Hand1, SortedHand1),
        predsort(compare_ranks, Hand2, SortedHand2),
        evaluate(SortedHand1, Type1, Details1),
        evaluate(SortedHand2, Type2, Details2),
        (Type1 = Type2 ->
            Details1 @> Details2
        ;   precedes(Type1, Type2, [straight_flush,
                                    four_of_a_kind,
                                    full_house,
                                    flush,
                                    straight,
                                    three_of_a_kind,
                                    two_pairs,
                                    one_pair,
                                    nothing])).

evaluate(Hand, Type, Details) :-
        (is_flush(Hand) ->
            (is_straight(Hand) ->
                (Type = straight_flush,
                 Hand = [(_, Rank) | _],
                 Details = [Rank])
             ;  (Type = flush,
                 maplist(rank, Hand, Details)))
         ;  (is_straight(Hand) ->
                (Type = straight,
          Hand = [(_, Rank) | _],
          Details = [Rank])
             ;  (partition_profile(match_rank, Hand, Profile),
                 associate(Profile, Type, [([4 | _], four_of_a_kind),
                                          ([3, X | _] : X>1, full_house),
                                          ([3 | _], three_of_a_kind),
                                          ([2, 2 | _], two_pairs),
                                          ([2 | _], one_pair),
                                          (_, nothing)])))).

is_flush(Hand) :-
        checklist(match_suit(_), Hand).

is_straight([_]).

is_straight([(_, Rank1), (_, Rank2) | Tail]) :-
        Rank2 is Rank1 - 1,
        is_straight([(_, Rank2) | Tail]).

compare_ranks((_, Rank1), (_, Rank2)) :-
        Rank2 =< Rank1.

match_rank((_, Rank), Rank).

match_suit(Suit, (Suit, _)).
```

Figure 8: Gala description of poker: Prolog predicates

comparing hands. Most of the code should be easy to understand based on the discussion above. The only new feature is the `objects` declaration, which declares that the game includes a deck of cards, and a money account for the two players and the pot. This declaration allows the program to use Gala functionality created for these types of objects. For example, the `deal` statement causes cards to be removed from the deck and added to a player's hand.

This Gala program takes five parameters: the suits and ranks of cards in the deck, the number of cards dealt to each player, the amount of cash initially given to each player, and the amount of the ante. Not only does the program describe many different poker games through the use of parameters, it can easily be modified to describe other variants of poker, in which cards are dealt one at a time, exchanged with the deck, individually revealed, and so on. For example, describing a game in which each player exchanges cards can be done by adding a gameflow describing the exchange process, and replacing the `bet` step in `flow` with `(take_turns(exchange), bet)`.

3.6 Gala programs and game trees

Above, we showed how we can specify any extensive form game as a Gala program. However, in order to apply standard solution algorithms to Gala-specified games, we need a process for converting a Gala program into a game tree. More precisely, given a Gala program and an assignment of values to the program parameters, we need to generate a game tree corresponding to the game.

To accomplish this task, we need to understand the relationship between an execution tree and a game tree. In some sense, an execution tree is an annotated game tree: just like in a game tree, each vertex in the execution tree corresponds to a state in the game.[10] However, the game tree contains nodes only for those states which represent a choice point in the game, one where different outcomes lead to different plays in the game. In contrast, an execution tree also contains vertices which correspond to deterministic manipulations of the gamestate, e.g., changing the values of variables, transforming a complex statement into simpler ones, etc. The states corresponding to these deterministic transformations would not be associated with nodes in the game tree.

Thus, the game tree is essentially a condensed version of the execution tree, where we eliminate all vertices that have only one successor. More precisely, the game tree has:

- An internal node for every vertex in the execution tree where the first command in the gameflow is the placeholder `choosing(···)`.
- A leaf node for every leaf vertex in the execution tree.
- An edge from one node to another, whenever there is a path consisting of deterministic edges between the corresponding vertices in the execution tree.

The information sets for the game tree can be derived from the local states of the associated vertices in the execution tree. An information set for player k corresponds to some local state s_k, and consists of all choice nodes for player k that correspond to execution-tree vertices where player k's local state is s_k. Note that the information set is determined only by the local state of the player whose turn it is to move.

4 Strategies and equilibrium

While defining a game and examining the possible scenarios is an interesting exercise, our goal is really to find good strategies for playing the game. In this section, we survey the game-theoretic definitions of a strategy, a minimax strategy, and a Nash equilibrium strategy. We describe these concepts in the framework of extensive form games.

The simplest of all strategies is a *pure (deterministic) strategy.* Like a conditional plan in AI, a pure strategy is a very explicit "how-to-play manual" that tells the player what to do at every

[10] To help disambiguate, we use "state" or "vertex" for nodes in the execution tree, and "node" for nodes in the game tree.

possible point in the game. In the poker example of Figure 5, such a manual for the gambler would contain an entry: "If I hold a King, and I passed in the first round, and the dealer bets, then bet 1." In general, a pure strategy π^k for player k specifies a choice at each of his or her information sets. (Since the player cannot distinguish between nodes in the same information set, the strategy cannot dictate different actions at those nodes.)

Fixing a strategy for each of the players does not completely determine the outcome of the game, since the game also contains moves representing nature. However, the behavior at these nodes, while random, is completely specified in the description of the game. Therefore, a tuple of strategies $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_N)$, where each π_k is a strategy for player k, determines a probability distribution over the leaves of the tree. We will denote the probability of reaching some node p in the tree by $\Pr_{\boldsymbol{\pi}}(p)$.

Definition 4.1: The *expected payoff* $H(\boldsymbol{\pi})$ is defined to be

$$H(\boldsymbol{\pi}) = \sum_{\text{leaves } p} \Pr_{\boldsymbol{\pi}}(p) h(p). \quad \blacksquare$$

Based on these definitions, we can now describe what we mean by 'solving' a game. A solution to a game is a recommendation to the various players of how to play the game, i.e., a tuple of strategies $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_N)$. Of course, not every tuple of strategies dictates behavior which is rational for the different players. At the very minimum, one would wish that each player's strategy be optimal *with respect to the current context*. That is, a player should not be able to do better by diverging from his strategy, provided the strategies of all other players remains constant. If a tuple of strategies satisfies this property for all players, the strategies are said to be in *equilibrium*. More formally, we have the following definition:

Definition 4.2: A strategy combination $(\pi_1, \ldots, \pi_N)$ is said to be in *equilibrium* if for every player k and every strategy π'_k for that player,

$$H_k(\pi_1, \ldots, \pi_k, \ldots, \pi_N) \geq H_k(\pi_1, \ldots, \pi'_k, \ldots, \pi_N). \quad \blacksquare$$

The equilibrium property is a highly desirable one. Without it, a player following the "recommended solution" may leave himself vulnerable to having his strategy predicted (since following the recommendation is the rational thing to do) and possibly taken advantage of. In that case, following the recommended solution is no longer rational, making the whole idea of solution somewhat dubious.

The ability to announce one's strategy without giving the advantage to the other player is particularly important in the context of an artificial agent. There, it is very difficult to keep one's strategy a secret. For one thing, it is always possible to break into the code. For another, it is often possible to subject the agent's program to intensive testing simply by playing the game over and over again.

In perfect information games, we can easily construct an equilibrium solution to the game via a process called *backward induction*: At the leaves of the tree, the payoffs for all players are known. At a state which is just before the end of the game, the player whose turn it is will choose the action that maximizes his or her own payoffs. Under the assumption that this player will act rationally, the player in the preceding node in the game tree can now determine the optimal action for him or her. This backward induction process reduces to the standard *minimax algorithm* (originally due to Zermelo [1913]) in the case of zero-sum games.

Unfortunately, in most real-life games, the players do not have perfect information. It is clear that this simple process cannot work for imperfect information games. Here, the decision as to the optimal move must be done for the entire information set, rather than for individual nodes. And a strategy which is optimal in one node in the information set may not be optimal in others. For example, in our simple poker game, there are two nodes in the information set corresponding to the situation where the dealer's hand is a Queen and the gambler bet. In one of these two nodes, the gambler has a Jack, and in the other, a King. Clearly, different moves are optimal in these two nodes.

How do we find an equilibrium in the far more complex case of imperfect information games? The answer is that we don't. In fact, as defined, an equilibrium might not even exist. To see this, consider the simple game of "scissors-paper-stone." There, any pure strategy is a losing one as soon as it is revealed to the other player. That is, we cannot achieve equilibrium with any pure strategy.

The problem is not with the notion of equilibrium, but with the use of pure strategies. Pure strategies are predictable, and predictable play gives the opponent information. The opponent can find a strategy calculated to take advantage of this information, thereby making the original strategy suboptimal. Unpredictable play, on the other hand, maintains the information gap inherent in imperfect information games. Unpredictability can only be guaranteed by using randomized strategies:

Definition 4.3: A *randomized strategy* μ_k for player k (called a *behavior strategy* in game theory) is a function that, for each information set u of player k, returns a probability distribution over the choices C_u at u. ∎

In our poker example, a randomized strategy for the gambler can be described by defining the probability of betting at each information set u_c and u'_c, $c = 1, 2, 3$.

Once we fix a tuple of randomized strategies $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_N)$, the behavior (albeit random) is specified at all points in the game. Therefore, just as for pure strategies, $\boldsymbol{\mu}$ determines a probability distribution $\Pr_{\boldsymbol{\mu}}$ over the nodes in the game tree. Thus, Definition 4.1 can be used (only substituting μ for π) to ascribe an expected payoff to a tuple of randomized strategies. Similarly, the notion of equilibrium presented in Definition 4.2 can be used, again substituting μ for π.

The use of randomized strategies allows us to find an adequate solution for the game of scissors-paper-stone. The strategy combination where each of the players assigns probability 1/3 to each of the three possible choices is clearly an equilibrium: Each player is guaranteed an expected payoff of 0, and can do no better with any other strategy so long as the other player sticks to the equilibrium.

However, it is far from clear that an equilibrium necessarily exists in complex games involving many moves. In his Nobel-prize winning theorem, Nash [1951] showed that the use of randomized strategies allows us to guarantee the existence of an equilibrium for imperfect information games.[11]

Theorem 4.4: [Nash, 1951] *Any extensive-form game with perfect recall has an equilibrium solution in randomized strategies.*

Just as in the case of perfect information games, the equilibrium strategies are particularly compelling when the game is zero-sum. Then, as shown by von Neumann [von Neumann and Morgenstern, 1947], any equilibrium strategy is optimal against a rational player. More precisely,

[11] Nash's result actually applies to mixed strategies and normal form games, both of which are described in the next section. The application of Nash's theorem to extensive form games and behavior strategies is based on a theorem by Kuhn [1953] asserting that, for extensive games of perfect recall, the two strategy representations are essentially equivalent.

the equilibrium pairs are those where each player plays the optimal defensive strategy: the one that provides the best worst-case payoff. This maximin behavior is reasonable for zero-sum games since it is, in fact, in the other player's best interests to be as harmful as possible. More formally:

Theorem 4.5: [von Neumann, 1947] *In a zero-sum game, a strategy pair μ_1^*, μ_2^* is in equilibrium iff μ_1^* maximizes $\max_{\mu_1} \min_{\mu_2} H_1(\mu_1, \mu_2)$ and μ_2^* maximizes $\max_{\mu_2} \min_{\mu_1} H_2(\mu_1, \mu_2)$,[12] where the maximization and minimization is over the space of randomized strategies.*

Thus, in zero-sum games, the best defensive strategies are optimal in a very strong sense: They are the best that can be achieved against a rational player, i.e., a player that also plays the defensive strategy. Furthermore, a player can publicly announce her intention to do so without adversely affecting her payoffs.

5 Solving games

Now that we have a clearly defined notion of a solution, how do we go about finding one? This is, in general, a very difficult problem. In the poker example of Figure 5, we would specify a strategy for each of the players using six numbers: the move probabilities at the various information sets (six information sets per player). When trying to solve a game, we need to *find* an appropriate set of numbers that satisfies the properties we want. That is, we want to treat the parameters of the strategy as variables, and solve for them. In the zero-sum case, for example, the general computational problem is:

Find $\boldsymbol{x}$ which achieves:

$$\begin{array}{ll} \max_{\boldsymbol{x}} & \min_{\boldsymbol{y}} H_1(\boldsymbol{x}, \boldsymbol{y}) \\ \text{subject to} & \boldsymbol{x} \text{ represents a strategy for player 1} \\ & \boldsymbol{y} \text{ represents a strategy for player 2} \end{array} \qquad (\star)$$

Given our definition of randomized strategies, the appropriate set of variables seems to be obvious: We simply use the different move probabilities in the game. In the poker example, we would have $\boldsymbol{x} = \{x_c, x'_c \;:\; c = 1,2,3\}$ representing the gambler's strategy, and $\boldsymbol{y} = \{y_d^p, y_d^b \;:\; d = 1,2,3\}$ representing the dealer's strategy.

Unfortunately, the space of legal assignments to these variables is a difficult one over which to search. It is continuous and high-dimensional. Furthermore, the payoff function H, which plays an important role in both the definition of equilibrium and the definition of maxmin strategy, is not a 'nice' linear function of the x's and y's.

We now survey the two main frameworks for solving games in extensive form. In Section 5.1 we survey the traditional normal form algorithms, and in Section 5.2 we review the more recent approach of [Koller *et al.*, 1994].

5.1 Normal form

The representation of a game in extensive form is rather complex, requiring many different components (nodes, edges, information sets, chance moves, ...) for representing the dynamics of the game and of the players' information state. By contrast, the goal of game theory is to provide a simple uniform framework for representing games, that will enable a mathematical analysis of their

[12] Since $H_2 = -H_1$, this is equivalent to minimizing $\min_{\mu_2} \max_{\mu_1} H_1(\mu_1, \mu_2)$.

properties. The complexity of the extensive form was viewed as detracting from its suitability as a primary representation, inducing game theorists to develop an alternative formulation of a game, called the normal form (also known as the *strategic form*). It turns out that the normal form also allows for clean and elegant solution algorithms, adding to its attraction.

The normal form abstracts away much of the structure of the game. It represents the game only via the list of pure strategies available to the players. More precisely, the normal form is a table, indexed by a tuple of pure strategies $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_N)$, one for every player. For each such tuple, it lists the tuple of payoffs $H(\boldsymbol{\pi}) = (H_1(\boldsymbol{\pi}), \ldots, H_N(\boldsymbol{\pi}))$.

Clearly, every extensive-form game can be converted into the normal form. We simply list all of the possible pure-strategy combinations $\boldsymbol{\pi}$, and compute the (expected) payoff tuple $H(\boldsymbol{\pi})$. While this tranformation loses most of the information encoded in the game tree, it does capture the basic components necessary for equilibrium analysis (Definition 4.2): the possible courses of actions and their outcomes.

In the case of general two-player games, also called bimatrix games, the normal form can be written as a pair of $m \times n$ matrices $\boldsymbol{A}$ and $\boldsymbol{B}$. A row represents a pure strategy π_1 of player 1, a column represents a simultaneously chosen pure strategy π_2 of player 2, and the corresponding entries in $\boldsymbol{A}$ and $\boldsymbol{B}$ are $H_1(\pi_1, \pi_2)$ and $H_2(\pi_1, \pi_2)$, respectively. In zero-sum games, $\boldsymbol{B} = -\boldsymbol{A}$, so the normal form of the game is completely specified by $\boldsymbol{A}$.

Figure 9, for example, shows part of the normal form of the simplified poker game of Figure 5. The entire normal form is a 27×64 matrix. The 27 rows correspond to the strategies of the gambler. Each strategy indicates what the gambler should do in the first round for each possible card, and whether the gambler should bet in the third round if he passes in the first round. Thus, for example, the triple pp, b, pb (in boldface in the figure) represents the strategy of passing on a Jack in both rounds, betting on a Queen, and passing on a King (in an attempt to bluff) and then betting if the dealer bets.

Each column corresponds to a dealer's strategy, indicating the action to be taken at v_d^p and v_d^b for each possible card d. For example, the triple pp, bp, bb (also in boldface) represents the strategy where the dealer passes on a Jack and bets on a King no matter what the gambler does, but bets on a Queen only if the gambler passed. The matrix only shows 16 of the 64 possible strategies.

The matrix entry for each strategy pair represents the expected payoff for this pair. In this case, it is simply the average of the six possible outcomes of the game arising from the six possible deals. For example, on a (Q, J) deal, the gambler would bet, and then the dealer would pass, leading to a payoff of 1. On a (K, Q) deal, the gambler would pass, and then the dealer would bet, giving the gambler the opportunity to bet after all, with an ensuing payoff of 2.

As we mentioned above, the simpler representation of a game also allows for simple solution algorithms. Recall that the main difficulty with solving games in the extensive form was the complexity of representing a randomized strategy. However, once we list all of a player's pure strategies, a randomized strategy can be viewed simply as a probability distribution over this set. Such a distribution is known as a *mixed strategy* in the game theory literature. Clearly, any randomized strategy generates such a distribution. For games of perfect recall, it is also the case that any such distribution (mixed strategy) is equivalent to a randomized strategy [Kuhn, 1953].

This representation of strategies allows us to construct a simple formulation of the problem of finding equilibrium strategies. The advantages are most tangible in the case of two-player games, so we will focus on those for the remainder of the section. However, much of the analysis holds unchanged in the general case.

Suppose player 1 has m pure strategies and player 2 has n pure strategies. Consider some mixed strategy μ_1 for player 1, i.e., some distribution over player 1's pure strategies $\pi_1^1, \ldots, \pi_1^m$.

	pp pp pp	pp pp pb	pp pp bp	pp pp bb	pp pb pp	pp pb pb	pp pb bp	pp pb bb	pp bp pp	pp bp pb	pp bp bp	**pp** **bp** **bb**	pp bb pp	pp bb pb	pp bb bp	pp bb bb
pp, pp, pp	0	0	0	0	0	0	0	0	$-\frac{1}{3}$	$-\frac{1}{3}$	$-\frac{1}{3}$	$-\frac{1}{3}$	$-\frac{1}{3}$	$-\frac{1}{3}$	$-\frac{1}{3}$	$-\frac{1}{3}$
pp, pp, pb	0	0	0	0	0	0	0	0	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$
pp, pp, b	0	0	0	0	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	0	0	0	0	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$
pp, pb, pp	0	0	$-\frac{1}{6}$	$-\frac{1}{6}$	0	0	$-\frac{1}{6}$	$-\frac{1}{6}$	$-\frac{1}{3}$	$-\frac{1}{3}$	$-\frac{1}{2}$	$-\frac{1}{2}$	$-\frac{1}{3}$	$-\frac{1}{3}$	$-\frac{1}{2}$	$-\frac{1}{2}$
pp, pb, pb	0	0	$-\frac{1}{6}$	$-\frac{1}{6}$	0	0	$-\frac{1}{6}$	$-\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	0	0	$\frac{1}{6}$	$\frac{1}{6}$	0	0
pp, pb, b	0	0	$-\frac{1}{6}$	$-\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	0	0	0	0	$-\frac{1}{6}$	$-\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	0	0
pp, b, pp	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{3}$	$-\frac{1}{6}$	0	$-\frac{1}{2}$	0	$-\frac{1}{2}$	0	$-\frac{1}{2}$	0	$-\frac{1}{2}$
pp, b, pb	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{2}$	0	$\frac{1}{2}$	0	$\frac{1}{2}$	0	$\frac{1}{2}$	0
pp, b, b	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{2}$	0	$\frac{1}{2}$	0	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{2}$	0	$\frac{1}{2}$	0
pb, pp, pp	0	0	$-\frac{1}{6}$	$-\frac{1}{6}$	0	0	$-\frac{1}{6}$	$-\frac{1}{6}$	$-\frac{1}{2}$	$-\frac{1}{2}$	$-\frac{2}{3}$	$-\frac{2}{3}$	$-\frac{1}{2}$	$-\frac{1}{2}$	$-\frac{2}{3}$	$-\frac{2}{3}$
pb, pp, pb	0	0	$-\frac{1}{6}$	$-\frac{1}{6}$	0	0	$-\frac{1}{6}$	$-\frac{1}{6}$	0	0	$-\frac{1}{6}$	$-\frac{1}{6}$	0	0	$-\frac{1}{6}$	$-\frac{1}{6}$
pb, pp, b	0	0	$-\frac{1}{6}$	$-\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	0	0	$-\frac{1}{6}$	$-\frac{1}{6}$	$-\frac{1}{3}$	$-\frac{1}{3}$	0	0	$-\frac{1}{6}$	$-\frac{1}{6}$
pb, pb, pp	0	0	$-\frac{1}{3}$	$-\frac{1}{3}$	0	0	$-\frac{1}{3}$	$-\frac{1}{3}$	$-\frac{1}{2}$	$-\frac{1}{2}$	$-\frac{5}{6}$	$-\frac{5}{6}$	$-\frac{1}{2}$	$-\frac{1}{2}$	$-\frac{5}{6}$	$-\frac{5}{6}$
pb, pb, pb	0	0	$-\frac{1}{3}$	$-\frac{1}{3}$	0	0	$-\frac{1}{3}$	$-\frac{1}{3}$	0	0	$-\frac{1}{3}$	$-\frac{1}{3}$	0	0	$-\frac{1}{3}$	$-\frac{1}{3}$
pb, pb, b	0	0	$-\frac{1}{3}$	$-\frac{1}{3}$	$\frac{1}{6}$	$\frac{1}{6}$	$-\frac{1}{6}$	$-\frac{1}{6}$	$-\frac{1}{6}$	$-\frac{1}{6}$	$-\frac{1}{2}$	$-\frac{1}{2}$	0	0	$-\frac{1}{3}$	$-\frac{1}{3}$
pb, b, pp	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{6}$	$-\frac{1}{3}$	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{6}$	$-\frac{1}{3}$	$-\frac{1}{6}$	$-\frac{2}{3}$	$-\frac{1}{3}$	$-\frac{5}{6}$	$-\frac{1}{6}$	$-\frac{2}{3}$	$-\frac{1}{3}$	$-\frac{5}{6}$
pb, b, pb	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{6}$	$-\frac{1}{3}$	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{6}$	$-\frac{1}{3}$	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{6}$	$-\frac{1}{3}$	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{6}$	$-\frac{1}{3}$
pb, b, b	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{6}$	$-\frac{1}{3}$	$\frac{1}{2}$	0	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{6}$	$-\frac{1}{3}$	0	$-\frac{1}{2}$	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{6}$	$-\frac{1}{3}$
b, pp, pp	$\frac{2}{3}$	$\frac{1}{6}$	$\frac{2}{3}$	$\frac{1}{6}$	$\frac{1}{6}$	$-\frac{1}{3}$	$\frac{1}{6}$	$-\frac{1}{3}$	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{3}$	$-\frac{1}{6}$	$-\frac{1}{6}$	$-\frac{2}{3}$	$-\frac{1}{6}$	$-\frac{2}{3}$
b, pp, pb	$\frac{2}{3}$	$\frac{1}{6}$	$\frac{2}{3}$	$\frac{1}{6}$	$\frac{1}{6}$	$-\frac{1}{3}$	$\frac{1}{6}$	$-\frac{1}{3}$	$\frac{5}{6}$	$\frac{1}{3}$	$\frac{5}{6}$	$\frac{1}{3}$	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{3}$	$-\frac{1}{6}$
b, pp, b	$\frac{2}{3}$	$\frac{1}{6}$	$\frac{2}{3}$	$\frac{1}{6}$	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{2}{3}$	$\frac{1}{6}$	$\frac{2}{3}$	$\frac{1}{6}$	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{3}$	$-\frac{1}{6}$
b, pb, pp	$\frac{2}{3}$	$\frac{1}{6}$	$\frac{1}{2}$	0	$\frac{1}{6}$	$-\frac{1}{3}$	0	$-\frac{1}{2}$	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{6}$	$-\frac{1}{3}$	$-\frac{1}{6}$	$-\frac{2}{3}$	$-\frac{1}{3}$	$-\frac{5}{6}$
b, pb, pb	$\frac{2}{3}$	$\frac{1}{6}$	$\frac{1}{2}$	0	$\frac{1}{6}$	$-\frac{1}{3}$	0	$-\frac{1}{2}$	$\frac{5}{6}$	$\frac{1}{3}$	$\frac{2}{3}$	$\frac{1}{6}$	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{6}$	$-\frac{1}{3}$
b, pb, b	$\frac{2}{3}$	$\frac{1}{6}$	$\frac{1}{2}$	0	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{6}$	$-\frac{1}{3}$	$\frac{2}{3}$	$\frac{1}{6}$	$\frac{1}{2}$	0	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{6}$	$-\frac{1}{3}$
b, b, pp	1	0	1	0	$\frac{1}{2}$	$-\frac{1}{2}$	$\frac{1}{2}$	$-\frac{1}{2}$	$\frac{2}{3}$	$-\frac{1}{3}$	$\frac{2}{3}$	$-\frac{1}{3}$	$\frac{1}{6}$	$-\frac{5}{6}$	$\frac{1}{6}$	$-\frac{5}{6}$
b, b, pb	1	0	1	0	$\frac{1}{2}$	$-\frac{1}{2}$	$\frac{1}{2}$	$-\frac{1}{2}$	$\frac{7}{6}$	$\frac{1}{6}$	$\frac{7}{6}$	$\frac{1}{6}$	$\frac{2}{3}$	$-\frac{1}{3}$	$\frac{2}{3}$	$-\frac{1}{3}$
b, b, b	1	0	1	0	$\frac{2}{3}$	$-\frac{1}{3}$	$\frac{2}{3}$	$-\frac{1}{3}$	1	0	1	0	$\frac{2}{3}$	$-\frac{1}{3}$	$\frac{2}{3}$	$-\frac{1}{3}$

Figure 9: Part of the normal form for three-card poker.

We can represent this distribution as a vector $\boldsymbol{x}$, with m components representing the probabilities $\mu_1(\pi_1^i)$ assigned by μ_1 to the pure strategies π_1^i of player 1. In fact, any nonnegative m-vector $\boldsymbol{x} = (x_1, \ldots, x_m)$ with $\sum_{i=1}^m x_i = 1$ describes a mixed strategy. Similarly, a mixed strategy μ_2 for player 2 can be represented by an n-vector $\boldsymbol{y}$. It is easy to see that the expected payoff $H_1(\mu_1, \mu_2)$ is equal to the matrix product $\boldsymbol{x}^T \boldsymbol{A}\boldsymbol{y}$, and similarly $H_2(\mu_1, \mu_2) = \boldsymbol{x}^T \boldsymbol{B}\boldsymbol{y}$.

The vectors $\boldsymbol{x}, \boldsymbol{y}$ provide us with an alternative set of variables for which we can solve. They allow us to reduce the problem of finding equilibrium strategies in two-player games to a simple problem of optimization over a linear space of vectors. For the zero-sum case, the problem ($\star$) described above can be written as:

Find $\boldsymbol{x}$ which achieves:

$$\begin{array}{ll} \max_{\boldsymbol{x}} \min_{\boldsymbol{y}} & \boldsymbol{x}^T \boldsymbol{A}\boldsymbol{y} \\ \text{subject to} & \sum_{i=1}^m x_i = 1 \\ & \sum_{j=1}^n y_j = 1 \\ & \boldsymbol{x}, \boldsymbol{y} \geq 0. \end{array}$$

This problem can be reformulated, by an appropriate use of *linear programming duality* [Dantzig, 1963], as a simple *linear programming problem*, resulting in the following theorem [von Neumann and Morgenstern, 1947]:

Theorem 5.1: [von Neumann, 1947] *The normal form of a zero-sum game defines a linear program (LP) whose solutions are the equilibria (maximin strategies) of the game.*

The standard Simplex algorithm for linear programming can be used to solve this LP very effectively. Other standard LP solution algorithms, while slower in practice, can be used to guarantee a worst-case polynomial time for the solution.

A different, but related, transformation allows the reformulation of the equilibrium problem for general two-player games as a *linear complementarity problem* (see [Cottle *et al.*, 1992] for a definition):

Theorem 5.2: *The normal form of a general two-player game defines a* linear complementarity problem (LCP) *whose solutions are the equilibria of the game.*

One of the solutions to this LCP (each of which corresponds to an equilibrium) can be found by the Lemke-Howson algorithm [Lemke and Howson, 1964]. This algorithm resembles the simplex algorithm both in its general operation and in the fact that, while requiring exponential time in the worst case, it is fast in practice. If a comprehensive list of equilibria is required, a general exhaustive enumeration scheme can be used (as in [Cottle *et al.*, 1992, p.17]). This, of course, requires exponential time.

At this point, one might think that the problem of solving games is essentially solved, at least in the two-player case. In the zero-sum case, we have a fairly efficient polynomial-time algorithm.[13] In the general case, the problem of finding a single equilibrium is not known to be solvable in worst-case polynomial time. We do, however, have an algorithm which is effective in practice.

However, a closer examination shows that Theorems 5.1 and 5.2 are highly misleading. Although useful, the normal form is not a representation which naturally captures a person's intuititive model of a game. In practice, a game-theoretic analyst would model the situation using some alternative

[13] In fact, we really cannot expect to do better for a normal form game. It is easy to show that the converse to Theorem 5.1 also holds, i.e., the problem of solving a linear program can be reduced to that of solving a normal form game.

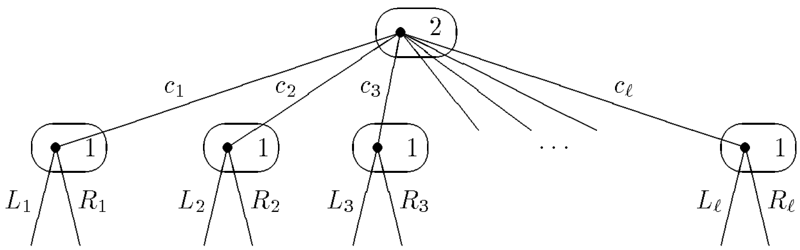


Figure 10: A simple game where player 1 has ℓ information sets and 2^ℓ pure strategies.

formalism, probably a game tree, and then convert to the normal form for the purposes of finding the solution. In both theorems, the size of the problem to be solved—the linear program or linear complementarity problem—is the size of the resulting normal form. And, unfortunately, the normal form of an extensive form game can be very large.

To understand this, consider the simple game tree in Figure 10. In this game, player 1 has ℓ information sets, one for each possible move of player 2 at the initial state. At each of these information sets, the player has to pick one of the two possible moves. Thus, a pure strategy can be described as an ℓ-length vector of R's and L's. The total number of possible pure strategies is therefore 2^ℓ. The normal form of this game contains a row for each and every one of these pure strategies. Therefore, the size of the normal form corresponding to this game is exponential in the size of the game tree!

More generally, our strategy space is the set of functions from information sets to moves, and is therefore also exponential in the number of information sets.[14] Thus, in the worst-case, the process of solving the game via the algorithms in Theorems 5.1 and 5.2 incurs an exponential blowup, not only in time, but also in space (which, in practice, is much worse). Clearly, this problem renders these algorithms impractical in the worst case.

One might think that this blowup does not arise in practice, but only in artificial games like the one above. So, while the worst case is bad, perhaps the typical case is quite reasonable. Unfortunately, this is not the case. Consider our simple poker example, generalized to a deck with k cards. For each card c, player 1 must decide whether to pass or bet, and if he has the option, whether to pass or bet in the third round. There are three courses of action for each c, so the total number of possible strategies is 3^k. Player 2, on the other hand, must decide on her action for each card d and each of the two actions possible for the first player in the first round. The number of different decisions is therefore $2k$, so the total number of deterministic strategies is $2^{2k} = 4^k$. Since the normal form has a row for each strategy of one player and a column for each strategy of the other, it is also exponential in k, while the size of the game tree is only $9k + 1$.

In fact, this blowup occurs in many real-life games, and appears to have been a significant obstacle to the widespread use of game theory as an analysis technique:

> This astronomical increase in the number of variables to be determined actually occurs in some important real-world problems and often forces the analyst to abandon the game theoretic approach.
>
> An Overview of the Mathematical Theory of Games
> William F. Lucas, Cornell University

[14] This analysis uses the standard conversion of an extensive form game into the normal form. Alternative conversions into a *reduced normal form* also exist. While the reduced normal form is smaller than the normal form, it is still exponential in the size of the game tree. See [von Stengel, 1996] for details.

5.2 Sequence form

The exponential blowup associated with the normal form makes the standard solution algorithms an unrealistic option for many games. Recently, however, a new approach to solving imperfect information games was developed by Koller, Megiddo, and von Stengel [1994]. This approach uses an alternative representation called the *sequence form*, which avoids the exponential blowup associated with the normal form.[15] We will describe the main ideas briefly here; for more details see [Koller *et al.*, 1994; von Stengel, 1996; Koller *et al.*, 1996].

The sequence form is based on a different representation of the strategic variables. Rather than representing probabilities of individual moves (as in the extensive form), or probabilities of full pure strategies (as in the normal form), the variables represent the *realization weight* of different *sequences* of moves.

Essentially, a sequence for a player corresponds to a path down the tree, but it isolates the moves under that player's direct control, ignoring chance moves and the decisions of the other players. In our poker game of Figure 5, for example, the gambler has 13 sequences: in addition to the empty sequence (which corresponds to the root of the game) he has four sequences for each card c: [bet on c] (in which case there is no third round), [pass on c], [pass on c, bet in the last round], and [pass on c, pass in the last round]. The dealer also has 13 sequences: the empty sequence, and for each card d, the four sequences [bet on d after seeing a pass], [pass on d after seeing a pass], [bet on d after seeing a bet], [pass on d after seeing a bet].

More precisely, let k be a player, and let p be a node of the game tree. There is a unique path from the root to p. On this path, certain edges correspond to moves of player k. The string of labels of these edges is denoted by $\sigma^k(p)$ and is called the *sequence* of choices of player k leading to p. It may be the empty sequence $\emptyset$, for example if p is the root. Essentially, the sequence $\sigma^k(p)$ describes the choices that player k has to make so that p can be reached in the game. I.e., a pure strategy π^k can only reach p if it chooses to make every move in $\sigma^k(p)$ at the information set when the move is relevant.

Our goal is to describe a randomized strategy via some set of weights associated with sequences. (Just as, for mixed strategies, we described a randomized strategy via a set of weights associated with pure strategies.) We therefore consider the probability that, for a given randomized strategy μ_k, a certain sequence is realized in play. Clearly, a player cannot unilaterally determine whether a sequence is realized. For example, the sequence [pass on c, bet in the last round] can only be realized if the dealer decides to bet in her turn. However, the player's strategy does determine whether a sequence is realized *given that the appropriate decision points in the game are reached.*

Thus, for a given randomized strategy μ_k, we define the realization weight of a sequence σ^k (some sequence for player k) as a conditional probability:

Definition 5.3: The *realization weight* of a sequence σ_k under μ_k, denoted $\mu_k(\sigma_k)$, is defined to be the probability that player k, playing according to μ_k, will take the moves in σ_k, given that the corresponding information sets are reached in the game. The set of realization weights $\mu_k(\sigma_k^1), \ldots, \mu_k(\sigma_k^{m_k})$, where $\sigma_k^1, \ldots, \sigma_k^{m_k}$ is the set of sequences of player k, is called a *realization plan* for player k. ∎

[15] In 1996, it was discovered that the sequence form was first suggested by Romanovsky [1962]. But Romanovsky's paper was published only in Russian and so his result was completely unknown to the scientific community in the West. This led to the later but independent development of the sequence form by Koller, Megiddo, and von Stengel.

$$\text{(a)} \quad \text{Gambler:} \quad \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 1 & 0 \end{pmatrix} \boldsymbol{x} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}.$$

$$\text{(b)} \quad \text{Dealer:} \quad \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix} \boldsymbol{y} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}.$$

Figure 11: Constraints on realization plans for 3-card poker

It is fairly easy to see that the realization weight $\mu_k(\sigma_k)$ is simply the product of the probabilities, according to μ_k, of the moves listed in σ_k.

We can find equilibrium strategies for player k by searching over the space of possible realization plans. More precisely, as described at the beginning of this section, we treat the realization weights of player k as variables $x_k^1, \ldots, x_k^{m_k}$, and search over the space of possible assignments to these variables for one that satisfies our optimality conditions (e.g., minimax). Note that there is at most one sequence, and therefore one variable, for each node in the game tree. Compare this to the exponential number of variables required to represent a strategy in the normal form.

Several crucial properties are required in order to make this process feasible. First, in order to substitute a search over realization plans for a search over randomized strategies, we must provide a correspondence between the two spaces. Clearly, not every vector of numbers (of the right length) actually represents some randomized strategy.

Lemma 5.4: [Koller and Megiddo, 1992] *If player k has perfect recall, then there exists a matrix E and a vector e such that a non-negative vector $\boldsymbol{x}$ of dimension m_k represents a randomized strategy for player k if and only if $E\boldsymbol{x} = e$. Furthermore, the matrix E and vector e can be derived from the game tree in linear time.*

That is, for a vector to represent a randomized strategy, it must satisfy certain constraints. These constraints (as encoded in the matrix E and the vector e) are quite simple: They force the weights to "add up" the right way. For example, the weight of the sequences [pass on c, bet in the last round] and [pass on c, pass in the last round], denoted $[\text{P}_c, \text{P}'_c]$ and $[\text{P}_c, \text{B}'_c]$, must add up to the weight of the sequence [pass on c], denoted $[\text{P}_c]$.

More generally, let σ_k be the sequence for player k leading to an information set at which player k is to move,[16] and let $c_1, \ldots, c_\ell$ be the possible moves at that information set. The sequence σ_k is realized if and only if one of its continuations is realized, and these are mutually exclusive events. Therefore, we must have that $x_{\sigma_k} = x_{\sigma_k \circ c_1} + \cdots + x_{\sigma_k \circ c_\ell}$, where $\sigma_k \circ c$ is the sequence obtained from concatenating the move c to the sequence σ_k. The only other constraints are that the realization weight of the empty sequence is 1 (because the root of the game is realized in any play of the game), and that $x_\sigma \geq 0$ for all σ.

Thus, a realization plan for the gambler is a 13-element vector $\boldsymbol{x}$ satisfying the constraints in Figure 11(a). The constraints are written under the assumption that the sequences are ordered

[16] The perfect recall assumption implies that there is at most one sequence σ_k leading to this information set.

as in Figure 12. Thus, the constraints asserting that the weights of $[\mathrm{P}_c, \mathrm{P}'_c]$ and $[\mathrm{P}_c, \mathrm{B}'_c]$ sum up to the weight of $[\mathrm{P}_c]$ (for each of the three different values of c) are encoded in lines 3, 5, and 7 of the matrix. Similarly, a realization plan for the dealer is a 13-element vector $\boldsymbol{y}$ satisfying the constraints in Figure 11(b).

The sequence form transformation allows us to accomplish the task of searching over the space of randomized strategies, by searching over the space of vectors satisfying the constraints. Furthermore, once an appropriate vector is found, we can easily recover the corresponding randomized strategy. That is, if a realization plan satisfies the condition of Lemma 5.4, then we can recover the corresponding randomized strategy μ_k from the realization weights. Let c be a possible move at some node p in the game tree where player k is to move, and let σ_k be the sequence of player-k moves leading to p. Then:

$$\Pr_{\mu_k}(c) = \frac{x_{\sigma_k \circ c}}{x_{\sigma_k}},$$

if $x_{\sigma_k} > 0$. Otherwise, we can define $\Pr_{\mu_k}(c)$ arbitrarily. It is straightforward to verify that the resulting strategy μ_k induces the realization plan $\boldsymbol{x}$.

The analysis above shows that we can represent a randomized strategy as a realization plan, and easily convert between the two representations. However, it is still not clear what advantages we have gained from doing so. The key point is that we describe the payoff function H as a linear function of the realization plan variables; this linearity was the key to the normal-form solution algorithms.

To derive this linearity property, we must divide each path in the game tree into its components. The probability that a path is actually taken in a game is the product of the probability of all of the moves on the path. This product can be re-expressed as the product of the realization weights of all the players' sequences on that path, times the probability of all the chance moves on the path. That is, for a given tuple $(\mu_1, \ldots, \mu_N)$ of randomized strategies,

$$\Pr_{\boldsymbol{\mu}}(p) = \beta(p) \cdot \prod_{k=1}^{N} \mu_k(\sigma_k(p)) ,$$

where $\beta(p)$ denotes the product of the chance probabilities on the path to p. Incorporating this expression into Definition 4.1, we obtain that:

$$H(\boldsymbol{\mu}) = \sum_{\text{leaves } p} h(p) \cdot \beta(p) \cdot \prod_{k=1}^{N} \mu_k(\sigma_k(p)) . \tag{1}$$

If $\boldsymbol{x}$ is a realization plan for player k corresponding to the strategy μ_k, then $\mu_k(\sigma_k)$ is precisely x_{σ_k}, so that $H(\boldsymbol{\mu})$ is, indeed, linear in the realization weight variables for each of the players.

In the two player case, the linearity of Equation (1) allows us to achieve a matrix formulation analogous to the one we used for the normal form. In this representation, called the *sequence form*, we again define a matrix $\boldsymbol{A}$ (or a pair of matrices $\boldsymbol{A}, \boldsymbol{B}$). But here, the rows correspond to the sequences (rather than pure strategies) of player 1, and the columns correspond to the sequences of player 2. The entry a_{ij} is the weighted sum of the payoff at the leaves that are reached by the pair of sequences σ_1^i and σ_2^j, i.e.,

$$a_{ij} = \sum_{p \,:\, \sigma_1(p)=\sigma_1^i, \sigma_2(p)=\sigma_2^j} \beta(p) \cdot h(p).$$

For example, the matrix entry for the pair of sequences [bet on a Queen] (denoted B_Q) and [pass on a Jack after seeing a bet] (denoted P_J^b) is 1 (obtained from the one leaf consistent with this pair

	ϵ	P_J^p	B_J^p	P_J^b	B_J^b	P_Q^p	B_Q^p	P_Q^b	B_Q^b	P_K^p	B_K^p	P_K^b	B_K^b
ϵ	0	0	0	0	0	0	0	0	0	0	0	0	0
P_J	0	0	0	0	0	-1	0	0	0	-1	0	0	0
P_J, P'_J	0	0	0	0	0	0	-1	0	0	0	-1	0	0
P_J, B'_J	0	0	0	0	0	0	-2	0	0	0	-2	0	0
B_J	0	0	0	0	0	0	0	1	-2	0	0	1	-2
P_Q	0	1	0	0	0	0	0	0	0	-1	0	0	0
P_Q, P'_Q	0	0	-1	0	0	0	0	0	0	0	-1	0	0
P_Q, B'_Q	0	0	2	0	0	0	0	0	0	0	-2	0	0
B_Q	0	0	0	1	2	0	0	0	0	0	0	1	-2
P_K	0	1	0	0	0	0	1	0	0	0	0	0	0
P_K, P'_K	0	0	-1	0	0	0	-1	0	0	0	0	0	0
P_K, B'_K	0	0	2	0	0	0	2	0	0	0	0	0	0
B_K	0	0	0	1	2	0	0	1	2	0	0	0	0

Figure 12: Payoff matrix for the sequence form for simplified poker.

of sequences). Note that if a pair of sequences is not consistent with any path to a leaf, the matrix entry is zero. Thus, the matrix entry for the pair [bet on a Queen] and [pass on a Jack after seeing a pass] is 0. Figure 12 shows the matrix for the sequence form for the simplified poker game.

Let $\boldsymbol{x}$ denote a realization plan for player 1, and $\boldsymbol{y}$ denote a realization plan for player 2. The entry x_i of $\boldsymbol{x}$ is the weight of the sequence σ_1^i at the ith row of $\boldsymbol{A}$, i.e., $x_i = \mu_1(\sigma_1^i)$, where μ_1 is the randomized strategy corresponding to $\boldsymbol{x}$ (assuming one exists). Similarly, $y_j = \mu_2(\sigma_2^j)$. It now easily follows from the definitions of a_{ij}, x_i, and y_j, and from Equation (1) that

$$H_1(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{x}^T \boldsymbol{A} \boldsymbol{y},$$

precisely as for the normal form.

We can now solve ($\star$) using realization weights as our strategic variables. The similarity between the normal form and sequence form formulations of ($\star$) allow the use of very similar techniques to those used in the proofs of Theorems 5.1 and 5.2, resulting in the following theorems:

Theorem 5.5: [Koller *et al.*, 1994] *The sequence form of a zero-sum game defines a linear program (LP) whose solutions are the equilibria (maximin strategies) of the game.*

As before, standard LP solution algorithms provide us with polynomial time algorithms for solving such games.

In the more general case:

Theorem 5.6: [Koller *et al.*, 1994] *The sequence form of a general two-player game defines a* linear complementarity problem (LCP) *whose solutions are the equilibria of the game. One of these solutions (each of which corresponds to an equilibrium) can be found by Lemke's algorithm [Lemke, 1965].*

Lemke's algorithm is a simple variant of the Lemke-Howson algorithm, with the same general characteristics.

At first glance, these results seem very similar to the normal-form results, so it might not be clear what we have gained. The key is that the sequence form (and the representation of a realization plan) is at most linear in the size of the game tree, since there is at most one sequence for each node in the game tree, and one constraint for each information set. Furthermore, it can be generated very easily by a single pass over the game tree. Thus, for example, the LP algorithm for zero-sum games is polynomial *in the size of the game tree* as opposed to polynomial *in the size of the normal form.* (The algorithms for the general case are analogous, except that the worst-case behavior is exponential in the size of the corresponding representation.) Since the normal form is typically *exponential* in the size of the game tree, these results provide an exponential time reduction over the standard normal-form algorithms!

6 The Gala system

The Gala system builds on the techniques described in the previous sections to provide a complete system for specifying and solving games. The system, described in Figure 13, uses three main stages for processing a game: generating a game tree from a Gala specification, solving the game tree, and examining the strategies.

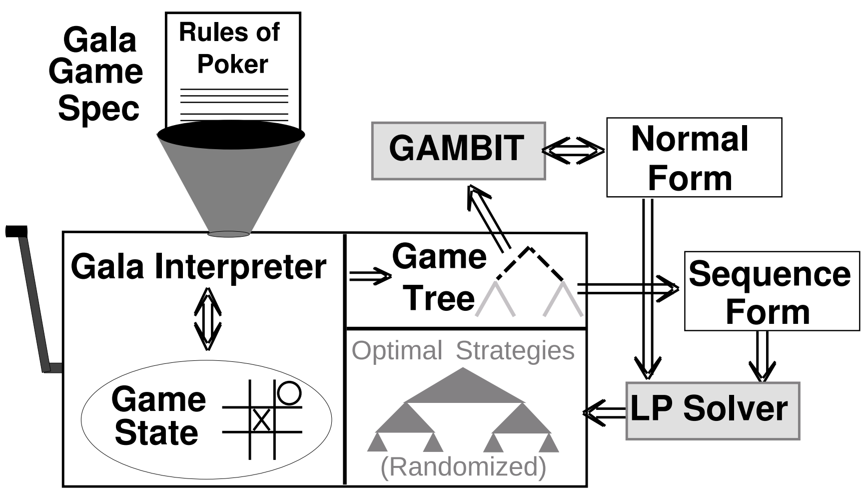


Figure 13: Architecture of the Gala system

The system as a whole provides end-to-end functionality, from the game specification to the examination of the optimal strategies. The result is a very useful package for experimental game-theoretic analysis. To illustrate this process, consider the game of poker. As demonstrated in Section 3.5, the specification of the general game is compact and natural. Appropriate instantiations of the parameters of the program allow the generation of a variety of games. These games can be effectively solved, and the strategy interpreted. We illustrate this below.

6.1 Game tree generation

In the first stage, the system receives as input a game specified in the Gala language (as described in Section 3). The system interprets the Gala code, running through the various possible executions of the game. The interpreter works by maintaining the game state, including the local state of the agents, and updating it according to the Gala semantics (Section 3). When the interpreter reaches a choice point, it explores all of the possible outcomes, in a depth-first fashion.

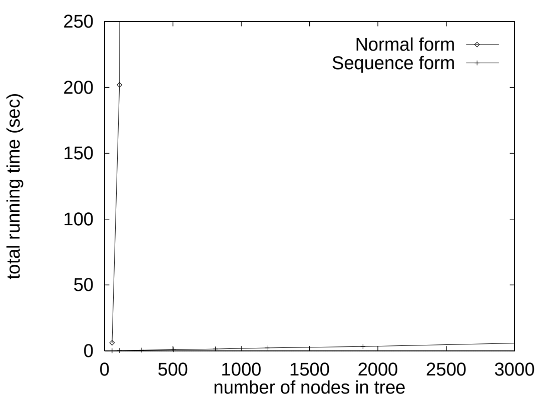
The interpreter is implemented in Prolog, relying on the underlying Prolog interpreter for interpreting the embedded Prolog predicates in the Gala code. It also utilizes the ability of Prolog to deal with nondeterminism to facilitate the generation of the different possible plays of the game.

As a result, the interpreter searches the entire execution tree of the program. During the generation of the execution tree, the corresponding game tree, as defined in Section 3.6, is generated. Whenever a `choose` statement is encountered, a node in the game tree is created. The node is added to the appropriate information set: the information set consisting of all nodes where the player's local state is the same (according to Prolog structural equality).

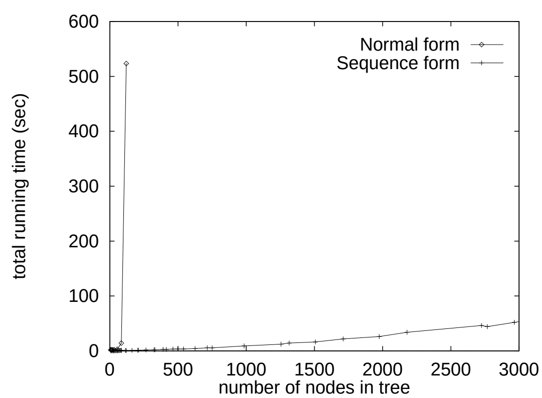
6.2 Solving the game

In the second stage of processing, the system solves the resulting extensive-form game. Currently, this occurs via one of two methods. The first is using the sequence form. The Gala system converts the game tree to its sequence form, and outputs the resulting matrices. These can be read by the appropriate solution algorithm. The current implementation of the system supports only the solution of zero-sum games, by using commercial linear programming software packages[17] over the resulting sequence form. We are currently working on the implementation of the case of general two-player games. The Gala system is the first to provide an implementation of any of the sequence form algorithms.

The other solution method is via an interface to the GAMBIT system [McKelvey, 1992], a state-of-the-art game theory software package which provides a number of game-theoretic solution algorithms (primarily for the normal form). Gala outputs the extensive form of a game in a format readable by GAMBIT, and calls the appropriate GAMBIT routines over the result. In particular, GAMBIT provides algorithms for multi-player games, so that such games written in the Gala language can be solved using the two systems in combination.



Poker



Inspection game

Figure 14: Normal form vs. sequence form running time

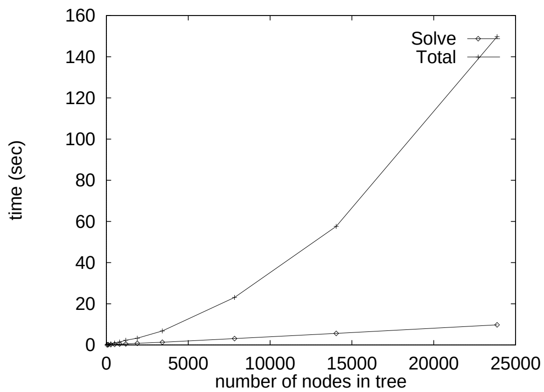
The Gala system, via its interface to GAMBIT, allows us to compare the performance of the sequence form and the normal form algorithms on practical problems. We experimented with two games: simplified poker, instantiated from the Gala program in Figures 7 and 8, and inspection game, instantiated from the program of Figure 6.

The experiments were performed as follows: The Gala system was used to generate different variants of these games, induced by different instantiations to the parameters. We generated poker

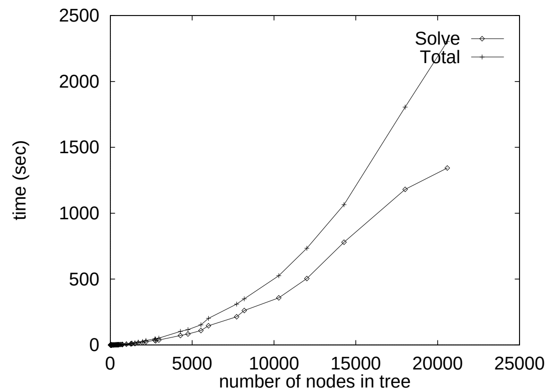
[17] CPLEX, Matlab and OSL are all supported.

games with different numbers of cards in the deck, and inspection games with a varying number of stages and inspections. For the sequence form experiments, the extensive form was transformed to the sequence form by the Gala solution engine and the resulting linear program was solved using CPLEX. The times for the conversion to sequence form and the solution of the linear program were added together.

For the normal form experiments, the game trees were printed to a file and converted to normal form by the GAMBIT system. The normal form linear program was again solved using CPLEX. The times for the conversion to normal form and the solution of the linear program were added together.



Poker



Inspection game

Figure 15: Time for generating and solving the sequence form

The resulting running times are shown in Figure 14. They are as one would expect in a comparison between a polynomial and exponential algorithm. The results are continued for the sequence form in Figure 15. (It was impossible to obtain normal-form results for the larger games.) In more recent experiments, we have solved poker games with over 140,000 nodes (see below). The largest games we have solved take about an hour on a Sun UltraSPARC II.

We believe that the Gala system can easily be modified to deal with much larger games, simply by streamlining the implementation. Figure 15 also shows the division of time between generating the sequence form and solving the resulting linear program. For the poker games, we can see that generating the sequence form takes the bulk of the time. Solving even the largest of these games takes less than 10 seconds. Thus, we believe that the system can be made to run considerably faster by optimizing the sequence-form generator. Furthermore, the current implementation of this generator stores the game tree in main memory as it is generated. As a consequence, the amount of main memory available posed the most severe constraint on the size of our experiments. A more careful implementation would write the game tree to disk as it is created, and then generate the sequence form from it (a process which can be done in a single pass over the tree). We believe that these minor modifications will allow the system to efficiently deal with much larger games.

In general, the main influences on the complexity of solving a sequence form game are the number of nodes in the tree and the structure of the information sets. In poker, for example, the total size of the tree is determined by the initial number of deals and the number of nodes in the subtree for each deal, which in turn depends on `cash-ante`. Figure 16 shows the number of cards in the deck, the number of cards dealt to each player, and the number of rounds for the largest games solved so far. The most extreme points are 127 card deck, 1 card each, 1 round; 3 card deck, 1 card each, 11 rounds; and 11 card deck, 5 cards each, 3 rounds.

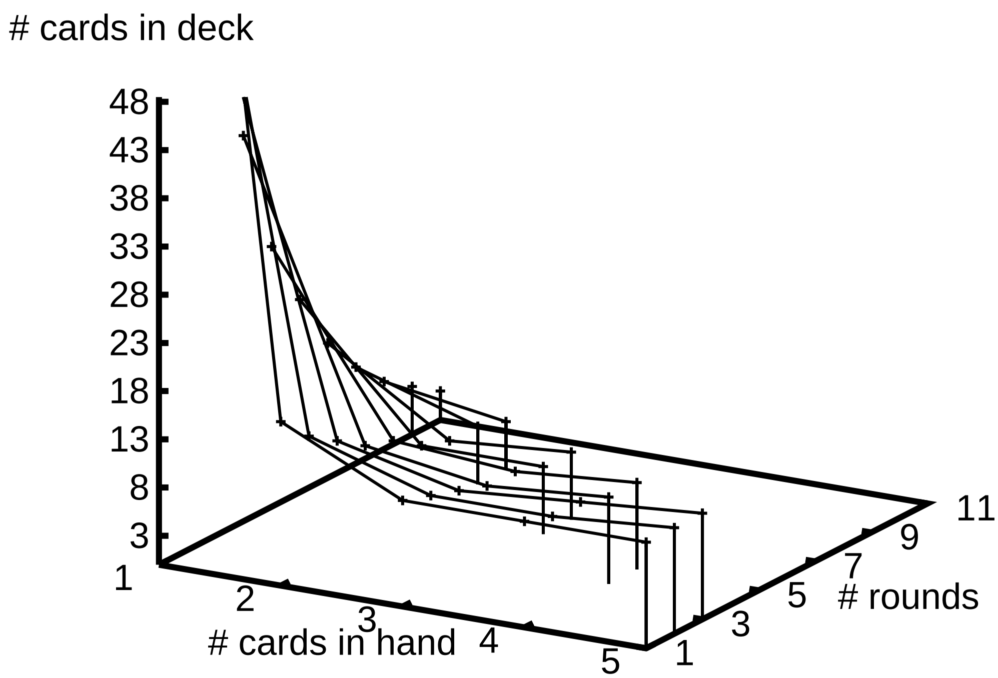


Figure 16: Parameters of largest poker games solved.

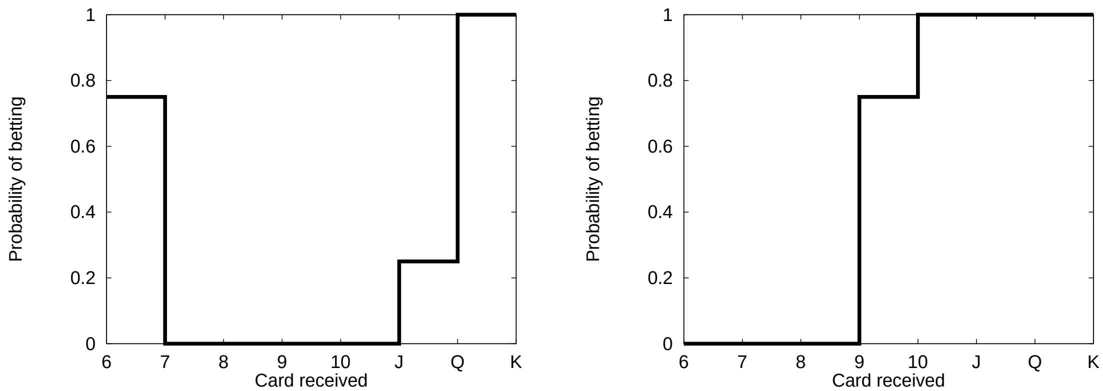
The influence of the information-set structure is apparent when we compare the performance of the sequence-form algorithm for poker games and for inspection games. Games such as poker, which are wide and shallow, tend to have many *parallel* information sets (information sets which cannot be reached in the same play of a game); they typically gain the most from the use of the sequence form. In the inspection game, on the other hand, the tree tends to be deep and narrow, reducing the computational savings obtained from the sequence-form representation.

6.3 Examining the strategies

The final stage of processing allows the user to examine the optimal strategies discovered by the solution engine. The user can navigate the game tree and view the optimal behavior strategy at any information set. The GAMBIT interface is also useful, since GAMBIT offers a nice graphical user interface for displaying game trees and the resulting strategies.

The Gala system also provides other useful information about the equilibrium solution. Once a pair of strategies is fixed, every branch in the game tree has a *value*, which is the expected outcome of the game given that the branch is taken and from that point the players continue to play according to the specified strategies. Also, the strategies induce a probability distribution over the plays of the game. Therefore, at any of her information sets, the player can derive a *belief state*—the resulting probability distribution over the nodes in the information set. Gala allows the user to examine the value of any action and the belief state of a player.

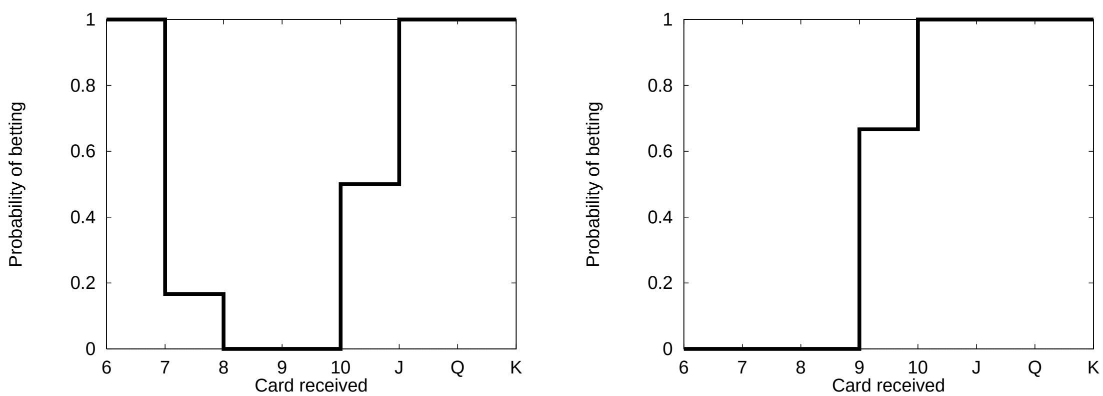
Figures 17 and 18 present the strategies resulting from this analysis for an eight-card version of the simplified poker discussed throughout the game. (We have an eight-card deck, each player is



First round

Second round

Figure 17: Gambler strategies for 8-card poker.



After seeing pass

After seeing bet

Figure 18: Dealer strategies for 8-card poker.

dealt one card, has an ante of \$1, and another dollar with which to bet.)[18] They provide an interesting example of the insights that can be achieved by such an analysis. Consider the probability that the gambler bets in the first round: it is very high on a 1, 0 on the middle cards, and then goes up for the high cards. The behavior for the low cards corresponds to bluffing, a characteristic that one tends to associate with the psychological makeup of human players. Similarly, after seeing a pass in the first round, the dealer bets on low cards with very high probability. Psychologically, we interpret this as an attempt to discourage the gambler from 'changing his mind' and betting in the final round. In more complex games, we see other examples where "human" behavior (e.g., underbidding) is game-theoretically optimal.

The fact that such 'psychological' properties are game-theoretically optimal in poker was first pointed out by Kuhn [1950]. It was a result of a laborious manual analysis for the 3-card poker game described above. The Gala system makes this type of analysis an effortless process, and allows it to be carried out for much more complex games.

As this example illustrates, the system also provides the exact probabilities for the various actions (e.g., bluffing on a low hand). Thus, the output of the system can be implemented directly. Before, only very simple games could be solved exactly. Thus, the outcome of a game-theoretic

[18] The game was generated and solved in under 2 seconds using the Gala system. The corresponding normal form algorithm did not terminate after two days of execution.

solution process was a strategy for a greatly simplified game, preventing the results from being directly applicable.

7 Discussion

7.1 Equilibrium and optimality

Throughout the paper, we have not questioned the use of Nash equilibrium as the definition of the desired solution. In practice, however, the Nash equilibrium is not always appropriate, particularly in the case of extensive-form games. The problem arises even in the (relatively very simple) case of zero-sum games. The definition of a maximin strategy (as set out in Theorem 4.5) only requires that the strategy μ_1 be optimal in the worst case, i.e., when player 2 is playing optimally. This situation is unlikely to materialize when human players are involved: human players invariably make mistakes.

The maximin criterion results in strategies that are guaranteed to do no worse against a sub-optimal adversary than against an optimal one. But the strategies are not guaranteed to take advantages of mistakes when they become apparent. This can lead to very counterintuitive behavior. For example, assume that player 1 is guaranteed to win \$1 against an optimal player 2. But now, the latter makes a mistake which allows player 1 to immediately win \$10000. It is perfectly consistent with the definition for the 'optimal' (maximin) strategy to continue playing so as to win the \$1 that was the original goal.

It is interesting to compare this behavior and the one resulting from an optimal strategy for a perfect information game. In the perfect information case, the maximin definition also allows for strategies with the same bizarre behavior. However, the specific backward induction algorithm used to solve the game avoids such strategies by picking, at each node, the move which is optimal *at that node*. Unfortunately, in the case of imperfect information games, it is much harder to define a notion of "optimal from this point on" (see Section 7.2 for some discussion). Essentially, the player might not know which node the game is actually at, and designing strategies that are optimal *from an information set* is much more complicated.

These difficulties are well-known, and have led to the development of an extensive suite of alternative (more refined) solution concepts. The discussion of the different options is beyond the scope of this paper; see [van Damme, 1983] for a survey. Currently, there seems to be no consensus as to the "right solution" to this problem. Nevertheless, it would be useful to incorporate one or more of the alternative solutions into the Gala system. Work is in progress on the computational aspects of these more refined solution concepts. See [McKelvey and McLennan, 1996; van den Elzen *et al.*, 1996] for some results.[19]

There is an alternative approach to the problem of dealing with less-than-perfect players. Rather than simply reacting to suboptimality when it is detected, we can try to learn the type of mistake that a certain player is prone to make. This approach can be used when there is a long-term interaction with the same player (or with players sharing similar flaws in play), either within a single (long) game or over a series of games. The ability of the Gala language to capture regularities in the game may be particularly useful in this context, since the high-level description of a game state can provide features for the learning algorithm.

[19][McKelvey and McLennan, 1996] also contains an extensive survey of computational methods for finding equilibria of non-zero-sum games.

7.2 Scaling up

While we can now solve games with tens of thousands of nodes, we are nowhere close to being able to solve huge games such as full-scale poker, and it is unlikely that we will ever be able to do so. A game tree for five-card draw poker, for example, where players are allowed to exchange cards, has over 10^{25} different nodes. The situation (for zero-sum games) is now quite similar to that of perfect-information games: We have algorithms that are fairly efficient *in the size of the game tree*; unfortunately, the game tree is often extremely large.

Nevertheless, chess-playing programs are very successful in spite of the fact that we currently cannot solve full-scale chess. Can the standard game-playing techniques be applied to imperfect information games? We believe that, in principle, the answer is yes, but the issue is nontrivial.

The standard algorithms for playing perfect-information games have the following general form:

1. Expand an initial subtree of the game, beginning at the root node, and using a heuristic evaluation function to assign a payoff to the leaves of the subtree.
2. Compute optimal pure strategies in the resulting subtree.[20]
3. Choose the action at the root that is optimal in the subtree, and wait for the opponent to choose an action.
4. Return to step 1, using the node reached as a consequence of the actions as the new root.

Can we adapt this approach to solving imperfect-information games? Initially, everything seems to work out fine. Just as before, we expand an initial subtree of the game, beginning at the root node, and use a heuristic evaluation function to assign a payoff to the leaves of the subtree. We solve the resulting tree to obtain randomized strategies for each player. We now have a strategy for the initial stages of the game, so play can proceed.

The problem arises when we attempt to apply this idea to subsequent positions in the game. At that point, the player might not know what node she is actually in. That is, the actual node might be p, while the player knows only that she is in the information set u which contains p. The player must therefore consider and somehow combine the subtrees rooted at each of the nodes in u, to obtain a single decision for the entire information set.

Of course, the player also has beliefs over the nodes in the information set, as determined by the strategies computed before and the probabilities of the chance moves. Therefore, one might think that the problem can be solved by a simple process of reasoning by cases. We consider each of the nodes in the information set separately, generating a game tree representing the assumption that this is the true state of the world. We find an optimal move in each tree. Finally, we choose a randomized strategy, in which each move is weighted by the probability that the true state is one for which it is optimal.

This approach is clearly flawed: it assumes that the player has perfect information when making her decision. Consider, for example, a variant of the inspection game where, in each round, the inspector also has the possibility of gathering intelligence before inspecting. This action has a cost, but does not waste an entire inspection. It might well be the optimal course of action. However, it will never be chosen by the algorithm described above. The algorithm considers two options: Either a violation has occurred, in which case one should simply inspect; or, a violation has not occurred, in which case the intelligence gather is a waste of resources.

[20]The solution process is often interleaved with the process of generating the game tree, so that partial results can be used to guide the expansion process to more relevant parts of the game tree.

A somewhat more plausible approach incorporates the player's uncertainty about the current position: The player creates a new game tree, in which the root node is a chance node, and its children are all the nodes in u. The chance move represents the player's uncertainty, so that the probability of the chance move leading to each node agrees with the player's belief state. The analysis now proceeds with the new tree.

The flaw in this algorithm is more subtle. While it does incorporate the player's uncertainty, it ignores the opponent's uncertainty. After all, even if the player knows that the true state is one of the nodes in u, the opponent does not necessarily have the same information. Consider the second round in three-card poker when the dealer is trying to decide on her action following a pass by the gambler. The dealer knows her own hand, a Jack. The two nodes in her information set correspond to those where the gambler has a Queen and a King. In any game that begins by choosing one of these two nodes at random, the gambler will always have a better card than the dealer, and know it, so the dealer has no reason to bluff. However, in the actual game, if the gambler has a Queen he will not know that he has the better card. And, in fact, the optimal strategy requires that the dealer bluff with non-zero probability, to take advantage of this situation.

The point is that the opponent's information state can have a large effect on one's strategy. Thus, one cannot eliminate nodes that the opponent considers possible, even if one knows for a fact that they are not. Similarly, nodes that neither player considers possible, but that one player thinks the other player considers possible, can also have an effect on a player's strategy. The only nodes that can be completely eliminated from consideration are those for which it is *common knowledge* [Fagin *et al.*, 1995] that they cannot be reached.[21]

It might be possible to develop a solution approach, analogous to the one described above, where we only eliminate such nodes. The resulting algorithm may be useful in some games. For example, in a poker game which consists of a deal followed by many rounds of betting, the number of reachable nodes stays constant, since all bets are common knowledge.[22] Unfortunately, many games have a more complex information structure, in which knowledge is dynamically created and revealed. This is the case with the inspection game and with variants of poker in which cards are dealt during the course of the game. In such games, the elimination of unreachable nodes may not significantly reduce the size of the game tree. Some other method of pruning the tree is needed, perhaps one based on the *value of information* metric (see, e.g., [Howard *et al.*, 1977]). We believe that developing an algorithm that prunes the tree in a principled manner presents an interesting research problem.

Besides the difficulties involved in pruning the tree, care must also be taken in designing the heuristic evaluation function used to evaluate nodes at the leaves of the subtree. This function must take into account the knowledge of the players in a state. In other words, the evaluation of a node must consider not only the node itself, but other nodes in the same information set. (In fact, the information state of all the players may conceivably affect the node's value.) For example, a naive evaluation function for poker might estimate that the player with the better hand will win the current stake, but this fails to differentiate between the states where a player knows she has the better hand and one where she does not, and this knowledge affects the value of the state. In three-card poker, if your opponent possesses a Jack, the state in which you possess a King is better than the one in which you possess a Queen, because you cannot be fooled by a bluff.

Even if we were to circumvent all of these obstacles, the bounded-depth-search approach is not a general solution. While it may work when the game tree is deep but not too wide, some game

[21] A fact φ is common knowledge if both players know φ, both players know that they both know φ, they know that they know that they know φ, and so on.

[22] Unfortunately, the main challenge in poker is not the depth of the tree but its width.

trees (full scale poker, for example) cannot even be expanded to depth 1. One approach that may be useful for such games (as well as for others) is based on abstraction. Many similar game states are mapped to the same abstract state, resulting in an abstract game tree much smaller than the original tree. The abstract tree can then be solved completely, and the strategies for the abstract game can be used to play the real game.

In poker, for example, this approach is likely to work very well. It is implausible that each of the $\binom{52}{5}$ hands that a player can get (in 5-card draw) generates a completely different strategy. Indeed, our experimental results in Section 6 reveal a lot of regularity in the strategies, with the gambler exhibiting identical behavior for the cards 6, 7 and 8. We believe that the solution for an appropriately abstracted game will result in strategies that are very close to optimal. We are currently in the process of experimenting with various possible abstractions.

Gala's ability to concisely and naturally represent a game via its rules may even allow us to construct appropriate abstractions automatically. Even more interestingly, it raises the intriguing possibility that we might, one day, develop solution algorithms that solve games directly from their rules.

7.3 Conclusion

The Gala system can provide the infrastructure for experimental game-theoretic research. Different abstractions and variants of a game can be generated easily and solved efficiently. Gala therefore provides a convenient tool for testing and evaluating different approaches for solving games.

Perhaps more importantly, Gala can play a crucial role in making game-theoretic reasoning more accessible to people and computer systems. The system allows complex games to be described simply, in a language even a layperson (one who is not a game-theory expert) can use. This allows the game to be refined until it adequately models the given situation. The effective solution algorithms implemented in the system allow solutions to be obtained in a reasonable amount of time. Finally, the solutions can be examined and interpreted in a way that is legible for a non-expert human. We hope that the public availability of Gala (available from `http://robotics.stanford.edu/~koller/gala.html`) will encourage the use of game-theoretic reasoning in day-to-day life.

Acknowledgements

We are deeply grateful to Richard McKelvey and Ted Turocy for going out of their way to ensure that the GAMBIT functionality we needed for our experiments was ready on time. We also thank the International Computer Science Institute at Berkeley for providing us access to the CPLEX system in the early stages of the project. We also wish to thank Nimrod Megiddo, Barney Pell, Stuart Russell, Yoav Shoham, John Tomlin, Bernhard von Stengel, Michael Wellman, and Salim Yusufali for useful discussions and comments. Some of this work was done while both authors were at U.C. Berkeley, with the support of a University of California President's Postdoctoral Fellowship and an NSF Postdoctoral Associateship in Experimental Science. More recent work was supported through the generosity of the Powell foundation, and by ONR grant N00014-96-1-0718.

References

[Aumann and Hart, 1992] R. J. Aumann and S. Hart, editors. *Handbook of Game Theory, Vol. 1.* North-Holland, Amsterdam, 1992.

[Avenhaus *et al.*, 1995] R. Avenhaus, B. von Stengel, and S. Zamir. Inspection games. In R. J. Aumann and S. Hart, editors, *Handbook of Game Theory, Vol. 3, to appear*. North-Holland, Amsterdam, 1995.

[Blair *et al.*, 1993] J. R. S. Blair, D. Mutchler, and C. Liu. Games with imperfect information. In *Working Notes of the AAAI Fall Symposium on Games: Planning and Learning*, 1993.

[Cottle *et al.*, 1992] R. W. Cottle, J.-S. Pang, and R. E. Stone. *The Linear Complementarity Problem*. Academic Press, 1992.

[Dantzig, 1963] G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.

[Fagin *et al.*, 1995] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about Knowledge*. MIT Press, Cambridge, Mass., 1995.

[Franklin *et al.*, 1993] M. Franklin, Z. Galil, and M. Yung. Eavesdropping games: A graph-theoretic approach to privacy in distributed systems. In *Proceedings of the 34th Annual IEEE Symposium on Foundations of Computer Science*, pages 670–679, 1993.

[Gordon, 1993] S. Gordon. A comparison between probabilistic search and weighted heuristics in a game with incomplete information. In *Working Notes of the AAAI Fall Symposium on Games: Planning and Learning*, 1993.

[Howard *et al.*, 1977] R. A. Howard, J. E. Matheson, and K. L. Miller, editors. *Readings on the Principles and Applications of Decision Analysis*. Stanford Research Institute, Strategic Decisions Group, Menlo Park, CA, 1977.

[Kaelbling *et al.*, 1996] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Planning and acting in partially observable stochastic domains. Submitted for publication. Can be obtained from `http://www.cs.duke.edu/~mlittman/topics/pomdp-page.html`, 1996.

[Koller and Megiddo, 1992] D. Koller and N. Megiddo. The complexity of two-person zero-sum games in extensive form. *Games and Economic Behavior*, 4:528–552, 1992.

[Koller *et al.*, 1994] D. Koller, N. Megiddo, and B. von Stengel. Fast algorithms for finding randomized strategies in game trees. In *Proceedings of the 26th Annual ACM Symposium on the Theory of Computing*, pages 750–759, 1994.

[Koller *et al.*, 1996] D. Koller, N. Megiddo, and B. von Stengel. Efficient solutions of extensive two-person games. *Games and Economic Behavior*, 14:247–259, 1996.

[Kuhn, 1950] H. W. Kuhn. A simplified two-person poker. In H. W. Kuhn and A. W. Tucker, editors, *Contributions to the Theory of Games I*, pages 97–103. Princeton University Press, 1950.

[Kuhn, 1953] H. W. Kuhn. Extensive games and the problem of information. In H. W. Kuhn and A. W. Tucker, editors, *Contributions to the Theory of Games II*, pages 193–216. Princeton University Press, 1953.

[Lemke and Howson, 1964] C. E. Lemke and J. T. Howson, Jr. Equilibrium points in bimatrix games. *Journal of the Society for Industrial and Applied Mathematics*, 12:413–423, 1964.

[Lemke, 1965] C. E. Lemke. Bimatrix equilibrium points and mathematical programming. *Management Science*, 11:681–689, 1965.

[Lucas, 1972] W. F. Lucas. An overview of the mathematical theory of games. *Management Science*, 15, Appendix P:3–19, 1972.

[McKelvey and McLennan, 1996] R. D. McKelvey and A. McLennan. Computation of equilibria in finite games. In *Handbook of Computational Economics*. 1996. To appear.

[McKelvey, 1992] R. D. McKelvey. GAMBIT*: Interactive Extensive Form Game Program*. California Institute of Technology, 1992.

[Nash, 1951] J. F. Nash. Non-cooperative games. *Annals of Mathematics*, 54:286–295, 1951.

[Norton, 1995] R. Norton. Winning the game of business. *Fortune Magazine*, 131(2):36, 1995.

[Parikh, 1992] P. Parikh. A game-theoretic account of implicature. In *Proceedings of the Fourth Conference on Theoretical Aspects of Reasoning about Knowledge (TARK)*. Morgan Kaufmann, 1992.

[Pell, 1992] B. Pell. Metagame in symmetric, chess-like games. In H. van der Herik and L. Allis, editors, *Heuristic Programming in Artificial Intelligence 3 — The Third Computer Olympiad*. Ellis Horwood, 1992.

[Rasmusen, 1989] E. Rasmusen. *Games and Information: An Introduction to Game Theory*. Basil Blackwell, Oxford, U.K. and Cambridge, Mass., 1989.

[Romanovsky, 1962] J. V. Romanovsky. Reduction of a game with perfect recall to a constrained matrix game (in russian). *Doklady Akademii Nauk SSSR*, 144:62–64, 1962.

[Rosenschein and Zlotkin, 1994] J. S. Rosenschein and G. Zlotkin. Consenting agents: Designing conventions for automated negotiation. *AI Magazine*, 15(3):29–46, 1994.

[Shenker, 1995] S. J. Shenker. Making greed work in networks: A game-theoretic analysis of switch service disciplines. *IEEE/ACM Transactions on Networking*, 3(6):819–831, 1995.

[Smith and Nau, 1993] S. J. J. Smith and D. S. Nau. Strategic planning for imperfect-information games. In *Working Notes of the AAAI Fall Symposium on Games: Planning and Learning*, 1993.

[van Damme, 1983] E. van Damme. *Refinements of the Nash Equilibrium Concept*. Lecture Notes in Economics and Mathematical Systems. Springer-Verlag, Berlin and New York, 1983.

[van den Elzen *et al.*, 1996] A. van den Elzen, B. von Stengel, and D. Talman. Tracing equilibria in extensive games by complementary pivoting. Manuscript, 1996.

[von Neumann and Morgenstern, 1947] J. von Neumann and O. Morgenstern. *The Theory of Games and Economic Behavior*. Princeton University Press, 2nd edition, 1947.

[von Stengel, 1996] B. von Stengel. Efficient computation of behavior strategies. *Games and Economic Behavior*, 14:220–246, 1996.

[Zermelo, 1913] E. Zermelo. Über eine Anwendung der Mengenlehre auf die Theorie des Schachspiels. In E. W. Hobson and A. E. H. Love, editors, *Proceedings of the Fifth International Congress of Mathematicians II*, pages 501–504. Cambridge University Press, 1913.