Generating and Solving Imperfect Information Games

Daphne Koller
University of California
Berkeley, CA 94720
daphne@cs.berkeley.edu

Avi Pfeffer
University of California
Berkeley, CA 94720
ap@cs.berkeley.edu

Abstract

Work on game playing in AI has typically ignored games of *imperfect information* such as poker. In this paper, we present a framework for dealing with such games. We point out several important issues that arise only in the context of imperfect information games, particularly the insufficiency of a simple game tree model to represent the players' information state and the need for randomization in the players' optimal strategies. We describe *Gala*, an implemented system that provides the user with a very natural and expressive language for describing games. From a game description, Gala creates an augmented game tree with *information sets* which can be used by various algorithms in order to find optimal strategies for that game. In particular, Gala implements the first practical algorithm for finding optimal randomized strategies in two-player imperfect information competitive games [Koller *et al.*, 1994]. The running time of this algorithm is *polynomial* in the size of the game tree, whereas previous algorithms were exponential. We present experimental results showing that this algorithm is also efficient in practice and can therefore form the basis for a game playing system.

1 Introduction

The idea of getting a computer to play a game has been around since the earliest days of computing. The fundamental idea is as follows: When it is the computer's turn to move, it creates some part of the game tree starting at the current position, evaluates the 'leaves' of this partial tree using a heuristic evaluation function, and then does a minimax search of this tree to determine the optimal move at the root. This same simple idea is still the core of most game-playing programs. This paradigm has been successfully applied to a large class of games, in particular chess, checkers, othello, backgammon, and go [Russell and Norvig, 1994, Ch. 5]. There have been far fewer successful programs that play games such as poker or bridge. We claim that this is not an accident. These games fall into two fundamentally different classes, and the techniques that apply to one do not usually apply to the other.

The essential difference lies in the information that is available to the players. In games such as chess or even backgammon, the current state of the game is fully accessible to both players. The only uncertainty is about future moves. In games such as poker, the players have *imperfect information*: they have only partial knowledge about the current state of the game. This can result in complex chains of reasoning such as: "Since I have two aces showing, but she raised, then she is either bluffing or she has a good hand; but then if I raise a lot, she may realize that I have at least a third ace, so she might fold; so maybe I should underbid, but ...." It should be fairly obvious that the standard techniques are inadequate for solving such games: no variant of the minimax algorithm duplicates the type of complex reasoning we just described.

In *game theory* [von Neumann and Morgenstern, 1947], on the other hand, virtually all of the work has focused on games with imperfect information. Game theory is mostly intended to deal with games derived from "real life," and particularly from economic applications. In real life one rarely has perfect information. The insights developed by game theorists for such games also apply to the imperfect information games encountered in AI applications.

It is well-known in game theory that the notion of a *strategy* is necessarily different for games with imperfect information. In perfect information games, the optimal move for each player is clearly defined: at every stage there is a "right" move that is at least as good as any other move. But in imperfect information games, the situation is not as straightforward. In the simple game of "scissors-paper-stone," any deterministic strategy is a losing one as soon as it is revealed to the other players. Intuitively, in games where there is an information gap, it is usually to my advantage to keep my opponent in the dark. The only way to do that is by using *randomized strategies*. Once randomized strategies are allowed, the existence of "optimal strategies" in imperfect information games can be proved. 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. Kuhn [1950] has shown for a simplified poker game that the optimal strategy does, indeed, use randomization.

The optimality of a strategy has two consequences: 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 game-playing programs, since they are 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 deduced by intensive testing. Given these important benefits of randomized strategies in imperfect information games, it is somewhat surprising that none of the AI papers that deal with these games (e.g., [Blair *et al.*, 1993; Gordon, 1993; Smith and Nau, 1993]) utilize such strategies.

In this work, we attempt to solve the computational problem associated with imperfect information games: Given a concise description of a game, compute optimal strategies for that game. Two issues in particular must be addressed. First, how do we specify imperfect information games? Describing the dynamics of the players' information states in a concise fashion is a nontrivial knowledge representation task. Second, given a game tree with the appropriate structure, how do we find optimal strategies for it?

We present an implemented system, called *Gala*, that addresses both these computational issues. Gala consists of four components. The first is a knowledge representation language that allows a clear and concise specification of imperfect information games. As our examples show, the description of a game in Gala is very similar to, and not much longer than, a natural language description of the rules of the game. The second component of the system generates game trees from a game description in the language. These game trees are augmented with *information sets*, a standard concept from game theory that captures the information states of the players.

The third component of the system addresses the issue of finding good strategies for such games. Obviously, the standard minimax-type algorithms cannot produce randomized strategies. The game theoretic paradigm for solving games is based on taking the entire game tree, and transforming it into a matrix (called the *normal* or *strategic form* of the game). Various techniques, such as linear programming, can then be applied to this matrix in order to construct optimal strategies. Unfortunately, this matrix is typically exponential in the size of the game tree, making the entire approach impractical for most games.

In recent work, Koller, Megiddo, and von Stengel [1994] present an alternative approach to dealing with imperfect information games. They define a new representation, called the *sequence form*, whose size is *linear* in the size of the game tree. They show that many of the standard algorithms can be adapted to find optimal strategies using this representation. This results in exponentially faster algorithms for solving a large class of games. In particular, they present an effective polynomial time algorithm for solving two-player fully competitive games (such as poker). We have implemented this algorithm as part of the Gala system, and tested it on large examples of several games. The results are encouraging, suggesting that, in practice, the running time of the algorithm is a small polynomial in the size of the game tree.

The final component of Gala presents the optimal strategies in a way that is comprehensible to the user. For any decision point in the game, it tells the user which actions should be played with which probability. The system also provides other information, such as one player's beliefs about the state of another agent, or the expected value of a branch in the tree. This functionality makes Gala a useful tool for game-theory researchers and educators, as well as for users who wish to use Gala as a game-theory based decision support system. Finally, Gala can also play the game according to the computed strategy, making it a basis for a computer game-playing system for imperfect information games.

2 Some basic game theory

Game theory is the strategic analysis of interactive situations. Several aspects of a situation are modeled explicitly: the players involved, the alternative actions that can be taken by each player at various times, the dynamics of the situation, the information available to players, and the outcomes at the end. Given such a model, game theory provides the tools to formally analyze the strategic interaction and recommend 'rational' strategies to the players.

The standard representation of a game in computer science is a tree, in which each node is a possible state of the game, and each edge is an action available to a player that takes the game to a new state. At each node there is a single player whose turn it is to choose an action. The set of edges leading out of a node are the choices available to that player. The player may be chance or 'nature', in which case the edges represent random events. The leaves of the tree specify a payoff for each player. This representation is inadequate for games with imperfect information, because it does not specify the information states of the players. A player cannot distinguish between states of the game in which she has the same information. Thus, any decision taken by the player must be the same at all such nodes. To encode this constraint, the game tree is augmented with *information sets*. An information set contains a set of nodes that are indistinguishable to a player at the time she has to make a decision.

Figure 1 presents part of the game tree for a simplified variant of poker described by Kuhn [1950]. The game has two players and a deck containing the three cards 1, 2, and 3. Each player antes one dollar and is dealt one card. The figure shows the part of the game tree corresponding to the deals $(2,1)$, $(2,3)$, and $(1,3)$. The game has three rounds. In the first round, the first player can either bet an additional dollar or pass. After hearing the first player's bet, the second player decides whether to bet or pass. If player 1 passes and player 2 bets, player 1 gets one more opportunity to decide whether or not to bet. If both bet or both pass, the player with the highest card takes the pot. If one player bets and the other passes, then the betting player wins one dollar. Let (c,d) denote the hands dealt to the two players. Initially, player 1 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 player 2's hand. In her turn, player 2 knows d as well as player 1's action at the first round. Hence, she has two information sets for each d—V_d^p and V_d^b—corresponding to player 1's previous action. Finally, player 1 has an information set U_c' at the third round.

Given a game tree augmented with information sets, one can define the notion of *strategy*. A *deterministic strategy*, like a conditional plan in AI, is a very explicit "how-to-play manual" that tells the player what to do at every possible point in the game. In the poker example, such a manual for player 1 would contain an entry: "If I hold a 3, and I passed on the first round, and my opponent bets, then bet 1." In general, a deterministic strategy for player i specifies a move at each of her information sets. Since the player cannot distinguish between nodes in the same information set, the strategy cannot dictate different actions at those nodes.

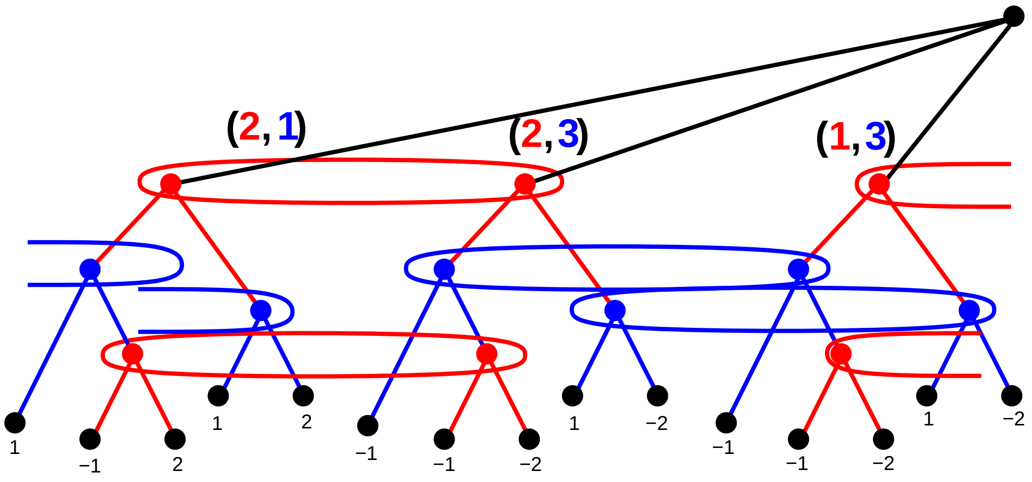


Figure 1: A partial game tree for simplified poker, containing three of the six possible deals. A move to the left corresponds to a pass, a move to the right to a bet. The information sets are drawn as ellipses; some of them extend into other parts of the tree.

Deterministic strategies are adequate for games with perfect information, where the players always know the current state of the game. In those games the information sets of both players are always single nodes, and a deterministic strategy s_i for player i is a function from those nodes at which it is her turn to move to possible moves at that node. The fact that deterministic strategies suffice for such games is the basis for the standard *minimax* algorithm (and its variants) used for games such as chess. In such games, called *zero-sum games*, there are two players whose payoffs always sum to zero, so that one player wins precisely what the other loses. As shown by Zermelo [1913], the strategies produced by the minimax algorithm are optimal in a very strong sense. Player i cannot do better than to play the resulting strategy if the other player is rational. Furthermore, she can publicly announce her intention to do so without adversely affecting her payoffs. A generalized version of the minimax algorithm shows the existence of optimal deterministic strategies for general games of perfect information. The resulting strategy combination $(s_1, \ldots, s_n)$ has the important property of being in *equilibrium*: for any i, player i cannot pick a better strategy than s_i if the other players are all playing their strategy s_j. This is a minimal property that we want of a "solution" to a game: Without it, we are drawn back into the web of second guessing that characterizes imperfect information games. (If she plays the "orthodox" strategy, then I should do X, but she will figure out that this is better for me, so she'll actually do Y, but then ....)

It should be fairly obvious that deterministic strategies will in general not have these properties in games with imperfect information. Deterministic strategies are predictable, and predictable play gives the opponent information. The opponent can then 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. Therefore, players in imperfect information games should use *randomized strategies*.

Randomized strategies are a natural extension of deterministic strategies. Where a deterministic strategy chooses a move at each information set, a randomized strategy (formally called a *behavior strategy*) specifies a probability distribution over the moves at each information set. In our poker example, a randomized strategy μ_1 for player 1 can be described by defining the probability of betting at each information set U_c and U'_c, $c = 1, 2, 3$. A combination of randomized strategies $\mu_1, \ldots, \mu_n$, one for each player, induces a probability distribution on the leaves of the tree, thereby allowing us to define the *expected payoff* $h_i(\mu_1, \ldots, \mu_n)$ for each player i.

In his Nobel-prize winning theorem, Nash showed that the use of randomized strategies allows us to duplicate the successful behavior that we get from deterministic strategies in the perfect information case. In general games, there is always a combination $\mu_1, \ldots, \mu_n$ of randomized strategies that is in equilibrium: for any i, and any strategy μ'_i,

$$h_i(\mu_1, \ldots, \mu_n) \geq h_i(\mu_1, \ldots, \mu'_i, \ldots, \mu_n).$$

That is, no player gains an advantage by diverging from the equilibrium solution, so long as the other players stick to it.

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, the equilibrium pairs μ_1, μ_2 are precisely those where μ_1 is the strategy that maximizes $\max_{\mu'_1} \min_{\mu'_2} h_1(\mu'_1, \mu'_2)$ and μ_2 is the strategy that maximizes $\max_{\mu'_2} \min_{\mu'_1} h_2(\mu'_1, \mu'_2)$ (which, since $h_2 = -h_1$, is precisely $\min_{\mu'_2} \max_{\mu'_1} h_1(\mu'_1, \mu'_2)$). Intuitively, μ_1 is the optimal defensive strategy for player 1: it provides the best worst-case payoff. It is these strategies that we will be most concerned with finding.

3 Gala: a game description language

As we mentioned, the first component of Gala is a knowledge representation language for describing games. This is a Prolog-based language, that uses the power of a declarative representation to allow clear and concise specification of games. The idea of a declarative language to specify games was proposed by Pell [1992], who utilizes it to specify

```
game(blind_tic_tac_toe,
 [players : [a, b],
  objects : [grid_board : array('$size', '$size')],
  params : [size],
  flow : (take_turns(mark,unless(full),until(win))),
  mark : (choose('$player', (X, Y, Mark),
             (empty(X, Y), member(Mark, [x, o]))),
          reveal('$opponent', (X, Y)),
          place((X, Y), Mark)),
  full : (\+(empty(_, _)) ->
           outcome(draw)),
  win : (straight_line(_, _, length = 3, contains(Mark)) ->
           outcome(wins('$player')))]).
```

Figure 2: A Gala description of blind tic-tac-toe

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 are generally of similar length to natural language representations of the rules of the game.

To illustrate some of the features of Gala, Figure 2 presents an example of a complete description for "blind tic-tac-toe," an imperfect information version of standard tic-tac-toe. The players take turns placing marks in squares, but in his turn a player can choose to mark either an `x` or an `o`; he reveals to his opponent the square in which he makes the mark, but not the type of mark used. As usual, the goal is to complete a line of three squares with the same mark.

A game description in Gala is a list of features, each one describing some aspect of the game. For example, `players : [a, b]` indicates that the game is to be played between two players named 'a' and 'b'.

The Gala language has several layers: the lower ones provide basic primitives, while the higher layers use those primitives to provide more complex functionality. The lowest layer provides the fundamental primitives for defining the structure of a game. The `choose(Player, Move, Constraint)` primitive describes the possible moves available to `player` at a given point in the game. It allows `Player` to make any move `Move` satisfying `Constraint`. This last argument can be an arbitrary segment of Prolog code. In our example, `Move` consists of a square, specified by its coordinates `X` and `Y`, and a mark `Mark`; `Constraint` requires that the square be empty and that `Mark` be either `x` or `o`. The first argument to `choose` can also be `nature`, in which case one of a number of events is chosen at random. By default, these random events have uniform probability, but a different probability distribution may be specified. The `outcome` primitive describes the outcome of the game at the end of a particular sequence of moves. This will often be a list of payoffs, one for each player; but, as the example demonstrates, Gala allows other possibilities. The `reveal(Player, Fact)` primitive describes the dynamics of the players' information states. It adds `Fact` to `Player`'s information state. The information added can be simple or an arbitrary Prolog expression. In blind tic-tac-toe, a player chooses both a square and a mark but reveals to his opponent only the mark.

At a somewhat higher level, the `flow` feature describes the course of the game. The game can be divided into phases: some may take place just once, while others can be repeated until a goal is reached. In blind tic-tac-toe, for example, the players take turns executing the sequence of actions specified in the `mark` feature, until the condition specified in the `full` or the `win` feature is satisfied. The `unless` condition is tested before the turn. Gala also allows gameflow to be nested recursively. Each phase can be described by its own series of features, which may include `flow`. The flow of bridge, for example, can be described as follows:

```
flow : (play_phase(bidding), play_phase(take_tricks)), ...
phase(bidding,
 [flow : (take_turns(bid, until(contract_reached))), ...
phase(take_tricks,
 [flow : (play_rounds(trick, 13), ...
```

In order to allow a natural specification of the game, Gala provides a separate representation for the *game state*, where relevant information about the current state of the game is stored. In blind tic-tac-toe, the game state contains the current board position. This information is accessed, for example, by `choose` in order to determine which moves are possible: only those squares that are `empty` are legal moves. The game state is maintained by modifying it appropriately, e.g., by the `place` operation, when the players make their moves. Much of the functionality in the higher levels of the Gala language is devoted to accessing and manipulating the game state.

The intermediate levels of Gala provide a shorthand 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. In blind tic-tac-toe, the statements that deal with the contents of squares are an instance of locations and their contents. Other examples of functionality supported by this level are `move(queen(white), (d,1), (d,8))` and `pay(gambler, pot, Bet)`.

On a more abstract level, we have observed that certain structures and combinations appear in virtually all games. While these are usually sets of one sort 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. A word in Scrabble and a 21 in Blackjack are another type of combination: a collection of objects bearing no particular relationship to each other but forming an interesting group in totality.

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 in which successive elements are related by `Predicate`; `partition(Relation, Set, Classes)` partitions `Set` into equivalence `Classes` based on `Relation`. For a more elaborate example, consider the following code, which concisely tests for all types of poker hand except flushes and straights.

```
detailed_partition(match_rank, Hand, Classes, Ranks, Sizes),
 associate(Sizes, Type,
   [([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 `detailed_partition` takes two inputs, a set—in this case `Hand`—and an equivalence relation—in this case `match_rank`, which relates two cards if they have the same rank. It partitions the set into equivalence classes, and produces three outputs: a list `Classes` of the equivalence classes in decreasing order of size; a corresponding list of the defining property of the equivalence classes, in this case the `Ranks` present in the hand; and a list `Sizes` of the sizes of the different classes. In this example, if `Hand` is `[9♡, 6♣, 9♠, 6♡, 6♢]`, then `Classes` would be `[[6♣, 6♡, 6♢], [9♡, 9♠]]`, `Ranks` would be `[6, 9]`, and `Sizes` would be `[3, 2]`. In poker, `Sizes` contains the relevant structure of the hand, and it is used to classify the hand using an association list. The above hand, for example, is immediately classified as a full house.

The high level modules of Gala build on the intermediate levels to provide more specific functionality that is common to a certain class of games, such as boards that form a grid, playing cards, dice, and so on. In the blind tic-tac-toe example, we declare a grid-board object. This makes a whole range of predicates available that depend on the board being rectilinear. The `straight_line` predicate is an example; it tests for a straight line of three squares containing the same mark. This predicate is defined in terms of `chain`. 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.

A useful feature of Gala is that it allows some parameters of the game to be left unspecified in the game description and provided when the game is played. In blind tic-tac-toe, the board size is such a parameter. This makes it very easy to encode a large class of games in a single program. These parameters can actually be code-containing features. Thus, it is possible to provide the movement patterns of pieces in a game at runtime. This allows a simple interface between Gala and Pell's *Metagame* program [Pell, 1992], which generates symmetric chess-like games randomly.

Given a description of a game in the Gala language, Gala generates the corresponding game tree with information sets as described in Section 2. The tree is defined by the `choose`, `reveal` and `outcome` primitives. The Gala interpreter "plays" the game and constructs the game tree as it encounters these operations. When it encounters a `choose` primitive, a node is added to the tree, and an edge is added for every option available to the player. The interpreter then explores each branch of the tree corresponding to each of the options. If the first argument to `choose` is a player, the system also adds the node to the appropriate information set of that player: the one that contains all the nodes where the player has the same information state. The information state consists of all facts revealed to the player by the `reveal` primitive, the list of choices available to the player, and all decisions previously taken by the player. If the first argument to `choose` is `random`, then the node is marked as a chance node, and the probability of each random choice is recorded. When the interpreter encounters the `outcome` primitive, it adds a leaf to the tree and backtracks to explore other branches.

4 Solving imperfect information games

How do we find equilibrium strategies in imperfect information games? This is, in general, a very difficult problem. Consider the poker example from Section 2. There, we specified a strategy for each of the players using six numbers. 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. The general computational problem is:

$$\begin{array}{lll} \text{Maximize}_x & \min_y h(\boldsymbol{x},\boldsymbol{y}) & \\ \text{subject to} & \boldsymbol{x} \text{ represents a strategy for player 1} & (\star) \\ & \boldsymbol{y} \text{ represents a strategy for player 2} & \end{array}$$

where $h(\boldsymbol{x},\boldsymbol{y})$ denotes the expected payoff to player 1 if the strategies corresponding to $\boldsymbol{x},\boldsymbol{y}$ are played.

It turns out that the heart of the problem is finding an appropriate set of variables for representing the strategy. The first attempt is to use the move probabilities in the behavior strategy. In the poker example, we would then have $\boldsymbol{x} = \{x_c, x'_c \,:\, c = 1,2,3\}$ representing player 1's strategy, and $\boldsymbol{y} = \{y^p_d, y^b_d \,:\, d = 1,2,3\}$ representing player 2's strategy. The problem is that this payoff is a nonlinear function of the x's and y's. In order to avoid this problem, which would force us to use nonlinear optimization techniques, the standard solution algorithms in game theory do not use game trees and behavior strategies as their primary representation. Rather, they operate on an alternative representation called the *normal form*. In the two-player case, the normal form is a matrix $\boldsymbol{A}$ whose rows are all the deterministic strategies of the first player and whose columns are all the deterministic strategies of the second. The entry in the ith row and jth column is the expected payoff to the players when player 1 plays strategy s^i_1 and player 2 plays strategy s^j_2. A randomized strategy can now be viewed as a probability distribution over all the deterministic strategies. Hence, $\boldsymbol{x}$ is simply a probability distribution over rows: it has a variable x_i for each row, such that $x_i \geq 0$ for all i, and $\sum_i x_i = 1$. If player 1 plays $\boldsymbol{x}$ and player 2 plays $\boldsymbol{y}$, then the expected payoff of the game is simply $\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{y}$. Under this representation of strategies, $(\star)$ takes a particularly simple form. It is then fairly easy to show that that appropriate vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ can be found from $\boldsymbol{A}$ using standard linear programming methods.

For non-zero-sum games, the normal form also forms the basis for essentially all solution algorithms. Gala provides access to the normal form algorithms using an interface to the GAMBIT system, developed by McKelvey and Turocy [McKelvey, 1992]. GAMBIT provides a toolkit for solving various classes of games, including games with more than two players and games where the interests of the players are not strictly opposing. Since Gala allows a clear and compact specification of such games, the combined system provides both a representation language and solution algorithms for games describing multi-agent interactions.

Unfortunately, the normal-form algorithms are practical only for very small games. The reason is that the normal form is typically *exponential* in the size of the game tree. This is easy to see: A deterministic strategy must specify an action at each information set. The total number of possible strategies is therefore exponential in the number of information sets, which is usually closely related to the size of the game tree. Consider our 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 at 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 general, the normal-form conversion is typically exponential in terms of both time and space.

This problem makes the standard solution algorithms an unrealistic option for many games. Due to the large branching factor in many games, even the approach of incrementally solving subtrees would not suffice to solve this problem. (This approach also encounters other difficulties in the context of imperfect information games; see Section 6.) Recently, a new approach to solving imperfect information games was developed by Koller, Megiddo, and von Stengel [1994]. This approach uses a conversion to an alternative form called the *sequence form*, which allows it to avoid the exponential blowup associated with the normal form. We will describe the main ideas briefly here; for more details see [Koller *et al.*, 1994].

The sequence form is based on a different representation of the strategic variables. Rather than representing probabilities of individual moves (as in the non-linear representation above), or probabilities of full deterministic 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, for example, player 1 would have $4k+1$ 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]. Player 2 also has $4k+1$ 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], [bet on d after seeing a bet]. Given a randomized strategy, the realization weight of a sequence for a player is the product of the probabilities of the player's moves encoded in the sequence. Essentially, the realization weight of the sequence corresponding to a path down the tree is a conditional probability: the probability that this path is taken given that the other players and nature all cooperate to make this possible. The probability that a path is actually taken in a game is therefore 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.

The *sequence form* of a two-player game consists of a payoff matrix $\boldsymbol{A}$ and a linear system of constraints for each player. In a two player game, the ith row of $\boldsymbol{A}$ corresponds to a sequence σ_1^i for player 1, and the jth column to a sequence σ_2^j for player 2. The entry a_{ij} is the weighted sum of the payoff at the leaves that are reached by this pair of sequences (they are weighted by the probabilities of the chance moves on the path). If a pair of sequences is not consistent with any path to a leaf, the matrix entry is zero. So, for example, the matrix entry for the pair of sequences [bet on 2] and [pass on 1 after seeing a bet] is 1. The matrix entry for the pair [bet on 2] and [pass on 1 after seeing a pass] is 0, since this pair is not consistent with any leaf.

We now solve $(\star)$ using realization weights as our strategic variables. We will have a variable x_{σ_1} for each sequence σ_1 of player 1, and a variable y_{σ_2} for each sequence σ_2 of player 2. Using the analysis above, we can show that the expected payoff of the game $h(\boldsymbol{x},\boldsymbol{y})$ is $\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{y}$. This is precisely analogous to the expression we obtained for the normal form. It remains only to specify constraints on $\boldsymbol{x}$ and $\boldsymbol{y}$ guaranteeing that they represent strategies. For the normal form, these constraints simply asserted that these vectors represent probability distributions. In this case, the constraints are derived from the following fact: If σ is the sequence for player i leading to an information set at which player i has to move,[1] and $m_1,\ldots,m_k$ are the possible moves at that information set, then we must have that $x_\sigma = x_{\sigma m_1} + \cdots + x_{\sigma m_k}$. The only other constraints are that the realization weight of the empty sequence is 1 (because the root of the game is reached in any play of the game), and that $x_\sigma \geq 0$ for all σ.

Note that the sequence form 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. The format of the sequence form resembles that of the normal form in many ways, and it appears that many normal-form solution algorithms can be converted to work for the sequence form. The work of [Koller *et al.*, 1994] focuses on the two-player case. They provide sequence-form variants for the best normal-form algorithms for solving both zero-sum and general two-player games. The result which is of most interest to us is the following:

Theorem 4.1: *The optimal strategies of a two-player zero-sum game are the solutions of a linear program each of whose dimensions is linear in the size of the game tree.*

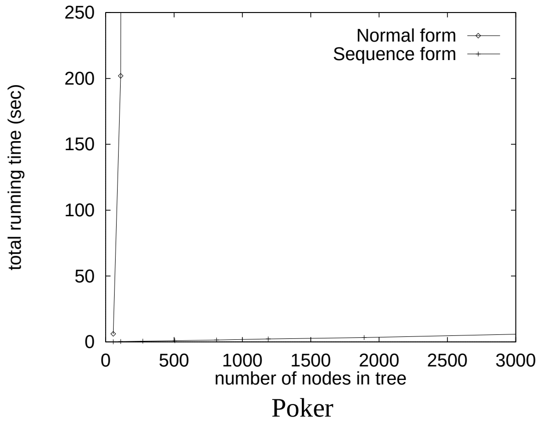
The matrix of the linear program mentioned in the theorem is essentially the sequence form. The resulting matrix can then be solved by any standard linear programming algorithm, such as the *simplex algorithm* which is known to work well in practice. We can also use a different linear programming algorithm whose worst-case running time is guaranteed to be polynomial. Hence, this theorem is the basis for an efficient polynomial time algorithm for finding optimal solutions to two-player zero-sum games.

5 Experimental results

The sequence-form algorithm for two-player zero-sum games has been fully implemented as part of the Gala system. The system generates the sequence form, creates the appropriate linear program, and solves it using the standard optimization library of CPLEX. We compared this algorithm to the traditional normal form algorithm by using GAMBIT to convert the game trees generated by Gala to the normal form, and CPLEX to solve the resulting linear program. We experimented with two games: the simplified poker game described in Section 2, increasing the number of cards in the deck; and an *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 resulting running times are shown in Figure 3. They are as one would expect in a comparison between a polynomial and exponential algorithm.

These results are continued for the sequence form in Figure 4. (It was impossible to obtain normal-form results for the larger games.) There, we also show the division of time between generating the sequence form and solving the resulting

[1]This formulation requires that the players never forget their own moves or information they once had. This implies that there is at most one sequence σ leading to this information set.



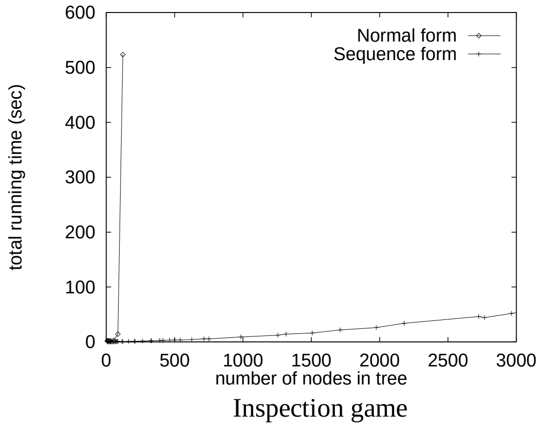
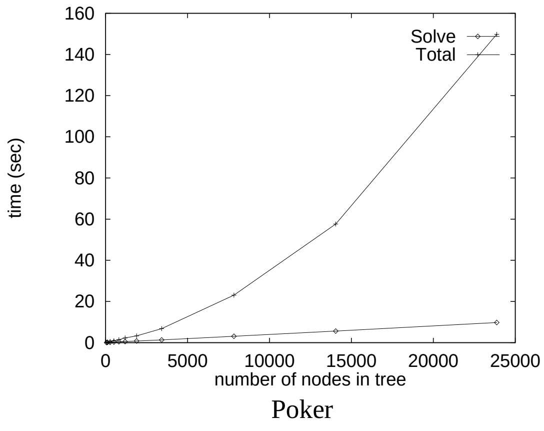


Figure 3: Normal form vs. sequence form running time



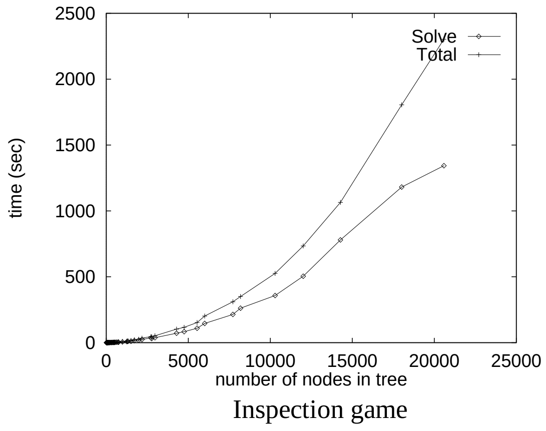


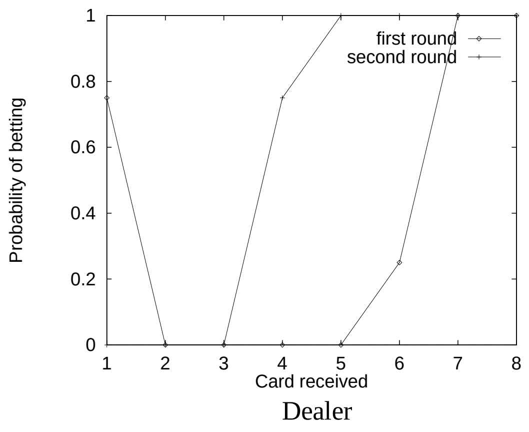
Figure 4: Time for generating and solving the sequence form

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. This leads us to believe that these techniques can be made to run considerably faster by optimizing the sequence-form generator. Finally, note that the algorithm is much faster for poker games than for the inspection games. In the full paper, we explain these results, and define certain characteristics of a game that tend to have a significant effect on the running time of the sequence-form algorithm.

As we remarked above, the final component of the Gala system reads in the strategies computed by this algorithm, and interprets them in a way that is meaningful with respect to the game. In particular, it allows the strategies to be examined by the user, who can then use them as part of the decision-making process. We have discovered that examining these strategies often yields interesting insights about the game. Figure 5 shows the strategies for both players in an eight card simplified poker. Consider the probability that the gambler bets in the first round: it is fairly high on a 1, somewhat lower on a 2, 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 on the final round. In more complex games, we see other examples where "human" behavior (e.g., underbidding) is game-theoretically optimal.

6 Discussion

As in the case of perfect information games, game trees for full-fledged games are often enormous. Although we expect to solve games with hundreds of thousands of nodes in the near future, full-scale poker is much larger than that and it is unlikely we will be able to solve it completely. Of course, chess-playing programs are very successful in spite of the fact that we currently cannot solve full-scale chess. Can we apply the standard game-playing techniques to imperfect information games? We believe that the answer is yes, but the issue is nontrivial. Even the concept of a 'subtree' is not well-defined in such games. For one thing, the program cannot simply create the subtree starting at the current state, since it does not know precisely which node of the game tree is the actual state of the game; it knows only that the node is one of those in a certain information set. In addition, information sets belonging to other players may cross the "subtree boundary," as was the case in Figure 1. It is not obvious how to deal with these problems. We hope to address this issue in future work. Another approach that may well prove fruitful is based on the observation that there is a lot of regularity in the strategies



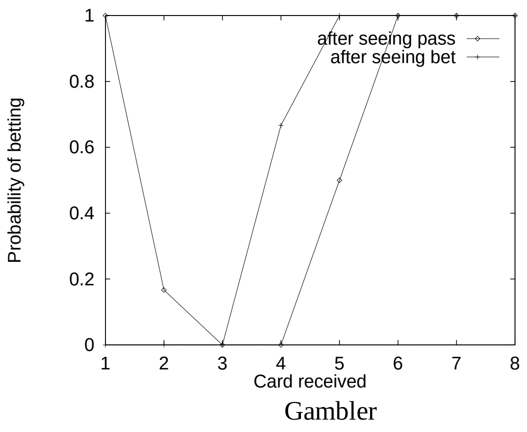


Figure 5: Strategies for 8 card poker

for small poker games: the player often behaves the same for a variety of different hands. This suggests that in order to solve large games, we could abstract away some features of the game, and solve the resulting simplified game completely. For the game of poker, we could abstract by partitioning the set of possible deals into clusters, and then solve the abstracted game. Our experimental results indicate that the resulting strategies would be very close to optimal.

Most of the techniques we discussed in this paper also apply to more general classes of games. Gala provides the functionality for specifying arbitrary multi-player games. Currently, these can only be solved using the traditional (normal-form) algorithms accessed through our GAMBIT interface, and these are practical only for small games. However, the sequence form can be used to represent any perfect recall game, and the results of [Koller *et al.*, 1994] indicate that many of the standard techniques could carry over from the normal form to the sequence form. We hope to use the sequence form approach for more general games, and show that the resulting exponential reduction in complexity indeed occurs in practice. If so, the resulting system may allow an analysis of multi-player games, a class of games that have been largely overlooked. Perhaps more importantly, the system could also be used to solve games that model multi-agent interactions in ‘real life’.

We believe that the Gala system facilitates future research into these and other questions. Its ability to easily specify games of different types and to generate many variants of each game allows any new approach to be extensively tested. We intend to make this system available through a WWW site (`http://www.cs.berkeley.edu/~daphne/gala/`), in the hope that it will provide the foundation for other work on imperfect information games.

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. We also wish to thank Nimrod Megiddo, Barney Pell, Stuart Russell, John Tomlin, and Bernhard von Stengel for useful discussions.

References

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

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

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

[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.

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

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

[Pell, 1992] B. Pell. Metagame in symmetric, chess-like games. In *Heuristic Programming in Artificial Intelligence 3 — The Third Computer Olympiad*. Ellis Horwood, 1992.

[Russell and Norvig, 1994] S.J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1994.

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

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

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