Representing Kriegspiel States with Metapositions

Paolo Ciancarini and Gian Piero Favini
Dipartimento di Scienze dell'Informazione, University of Bologna, Italy
Email: {cianca,favini}@cs.unibo.it

Abstract

We describe a novel approach to incomplete information board games, which is based on the concept of *metaposition* as the merging of a very large set of possible game states into a single entity which contains at least every state in the current information set. This merging operation allows an artificial player to apply traditional perfect information game theory tools such as the Minimax theorem.

We apply this technique to the game of Kriegspiel, a variant of chess characterized by strongly incomplete information as players cannot see their opponent's pieces but can only try to guess their positions by listening to the messages of a referee. We provide a general representation of Kriegspiel states through metaposition trees and describe a weighed maximax algorithm for evaluating metapositions. We have tested our approach competing against both human and computer players.

1 Introduction

Incomplete information board games are an excellent testbed for the study of decision making under uncertainty. In these games, only a subset of the current state of the game is made known to the players, who have to resort to various methods in order to find an effective strategy.

Kriegspiel is a chess variant in which the players cannot see their opponent's pieces and moves: all rules of Chess stay, but the game is transformed into an incomplete information game. We have built a program called Darkboard, which plays a whole game of Kriegspiel searching over a game tree of metapositions. The preliminary results we have are quite encouraging.

This paper is organized as follows: in the next section we describe the concept of metaposition. Then we apply this concept to Kriegspiel. In Sect. 3 we show how we apply weighed maximax to metapositions; in Sect. 4 we introduce an evaluation function for a game tree of metapositions. Finally, in Sect. 5 we describe some experimental results and draw our conclusions.

2 Metapositions

The original concept of *metaposition* was introduced in [Sakuta, 2001], where it was used to solve endgame positions for a Kriegspiel-like game based on Shogi. The primary goal of representing an extensive form game through metapositions is to transform an imperfect information game into one of perfect information, which offers several important advantages and simplifications, including the applicability of traditional techniques associated with these games. A metaposition, as described in the quoted work, merges different, but equally likely moves, into one state (but it can be extended to treat moves with different priorities).

In its first formulation, a metaposition is a set of game states grouping those states sharing the same strategy spaces (legal moves) available to the player. A given move for the first player is guaranteed to be legal across every state in the set, or in none at all. If we consider the game tree from the point of view of metapositions instead of single game states, it is readily seen that the game becomes, nominally, one of perfect information. When the first player moves, the current metaposition (which coincides with the current information set) is updated by playing that move on all the boards, as it is certainly legal. The opponent's moves generate a number of metapositions, depending on the resulting strategy space for the first player. If the first player knew his new strategy space beforehand, he would be able to uniquely determine the new metaposition because different metapositions have, by definition, different strategy spaces associated to them.

Unfortunately, a player's strategy space is not known beforehand in such games as Kriegspiel. There is, obviously, no way to find out whether a move is legal other than by trying it. Therefore, an extension of the definition of metaposition is needed, refining the concept of strategy space. Often, a move which is likely to be illegal on the referee's board is a good strategy for the player. Illegal moves are the main mechanism for acquiring information on the opponent's piece setup. It makes little sense to discard their analysis only because the referee knows they are illegal.

A second formulation of metapositions is as follows.

Definition. If $\mathbf{S}$ is the set of all possible game states and $\mathbf{I} \subseteq \mathbf{S}$ is the information set comprising all game states compatible with a given sequence of observations (referee's messages), a metaposition $\mathbf{M}$ is any *opportunely coded* subset of

$\mathbf{S}$ such that $\mathbf{I} \subseteq \mathbf{M} \subseteq \mathbf{S}$.

The strategy space for $\mathbf{M}$ is the set of moves that are legal in at least one of the game states contained in the metaposition. These are *pseudolegal* moves, assumed to be legal from the player's standpoint but not necessarily from the referee's. A metaposition is endowed with the following functions:

- a *pseudomove* function `pseudo` that updates a metaposition given a move try and an observation of the referee's response to it;
- a *metamove* function `meta` that updates a metaposition after the unknown move of the opponent, given the associated referee's response;
- an *evaluation* function `eval` that outputs the desirability of a given metaposition.

From this definition it follows that a metaposition is any superset of the game's information set (though clearly the performance of any algorithm will improve as $\mathbf{M}$ tends to $\mathbf{I}$). Every plausible game state is contained in it, but a metaposition can contain other states which are not compatible with the history. The reason for this is two-fold: on one hand, being able to insert (opportune) impossible states enables the agent to represent a metaposition in a very compact form, as opposed to the immense amount of memory and computation time required if each state were to be listed explicitly; on the other hand, a compact notation for a metaposition makes it easy to develop an evaluation function that will evaluate whole metapositions instead of single game states. This is the very crux of the approach: metapositions give the player an illusion of perfect information, but they mainly do so in order to enable the player to use a Minimax-like method where metapositions are evaluated instead of single states. For this reason, it is important that metapositions be described in a concise way so that a suitable evaluation function can be applied.

It is interesting to note that metapositions move in the opposite direction from such approaches as Monte Carlo sampling, which aim to evaluate a situation based on a significant subset of plausible game states. This is perhaps one of the more interesting aspects of the present research, which moves from the theoretical limits of Monte Carlo approaches as stated, for example, in [Frank and Basin, 1998], and tries to overcome them. In fact, a metaposition-based approach does not assume that the opponent will react with a best defense model, nor is it subject to strategy fusion because uncertainty is artificially removed.

The nature of the "opportune coding" required to represent a metaposition, a superset of the usually computationally intractable information set $\mathbf{I}$, will depend on the specific game. As far as Kriegspiel is concerned, we move from the results in [Ciancarini *et al.*, 1997] on information set analysis in order to win a Kriegspiel endgame. Here the authors use the information set in order to recognize position patterns in the King and Pawn versus King (KPK) endgame, however performing no search in the problem space.

The first representation of Kriegspiel situations using metapositions together with an evaluation function was given in [Bolognesi and Ciancarini, 2003] and [Bolognesi and Ciancarini, 2004]. Their analysis is, however, limited to a few chosen endgame examples, such as King and Rook versus King (KRK). Because of the small size of the game's information set in these particular scenarios (which is limited to the possible squares where the opponent's King may be), metapositions coincide exactly with the information set in the quoted papers ($\mathbf{M} = \mathbf{I}$). The present work deals with a generic full game of Kriegspiel, with the opponent controlling an arbitrary number of pieces, and such an assumption is unreasonable.

Our approach to coding a Kriegspiel metaposition is, essentially, the abstract representation of a chessboard containing both real pieces, belonging to the players and *pseudopieces* (ghost pieces that may or may not exist). Trivially, a metaposition coded in this fashion represents a number of states equal to the product of the number of pseudopieces on each square. Each square, therefore, has the following information attached to it.

- Piece presence: whether the square contains an allied piece.
- Pseudopiece presence: a bitfield representing the possible presence of opposing pieces at the given location. There are seven possible pseudopieces, and any number of them may appear simultaneously at the same square: King, Queen, Rook, Bishop, Knight, Pawn and Empty. The last is a special pseudopiece indicating whether the square may be empty or is necessarily occupied.
- Age information: an integer representing the number of moves since the agent last obtained information on the state of this square. This field provides the integration of some of the game's history into a metaposition in a form that is easily computable.

Moreover, a metaposition will store the usual information concerning such things as castling rights and the fifty moves counter, in addition to counters for enemy pawns and pieces left on the chessboard. It is easy to notice that such a notation is extremely compact; in fact, each square can be represented by two bytes of data.

A pseudopiece is, essentially, a ghost piece with the same properties as its real counterpart. It moves just like a real piece, but can move through or over fellow pseudopieces, except in specific cases. For example, it is possible to enforce rules to prevent vertical movement across a file where an opponent's pawn is known to be.

A metaposition follows and mantains the following invariant: *if a pseudopiece is absent at a given location, then no piece of that type can appear there in any state of the current information set*. The opposite is not true, and because of their relaxed movement rules, pseudopieces may appear in places where a real enemy piece could not be according to the information set. This is equivalent to saying that $\mathbf{I} \subseteq \mathbf{M}$. A metaposition then represents a much larger superset of the information set, and in certain phases of the game, some pseudopieces can be found at almost every location.

2.1 Updating knowledge

Metapositions not deal with moves, but with *pseudomoves* and *metamoves*. A pseudomove represents the agent's move, which can be legal or illegal, and has an associated observation (a set of umpire messages sent in response to the move attempt). A metamove represents the collective grouping of all the possible opponent's moves, and it is associated to an observation, too. Darkboard implements `pseudo` and `meta` by accepting a metaposition and an observation, with an updated metaposition being returned as the output. Clearly, `pseudo` reduces the uncertainty by eliminating some pseudopieces, whereas `meta` increases it by spawning new pseudopieces.

Intuitively, `meta` does such things as clearing all pseudopieces on the moved piece's path and infer the position of the opponent's King from check messages; `pseudo` has every pseudopiece spawn copies of itself on every square it can reach in one move. It is readily seen that such operations maintain the $\mathbf{I} \subseteq \mathbf{M}$ constraint that defines a metaposition. A number of optimizations are possible to improve their accuracy and therefore quality of play, but because of the loose nature of a Kriegspiel metaposition, they are not required.

As a metaposition represents a grouping of a very large number of positions which cannot be told apart from one another, it is clear that updating such a data structure is no trivial task; in truth, this process does account for the better part of the agent's computation time. Updating an explicitly listed information set with a pseudomove would involve finding all the positions compatible with the outcome of that move (legal, not legal, check, etc.), discarding anything else, and applying the move to the compatible chessboards. Updating a metaposition after a metamove would prove an even more daunting task, as we would have to consider each possible move for each possible chessboard in the set. Again, this is a problem that can only be overcome through a suitable approximation (or by limiting the number of chessboards down to a manageable pool, as in [Parker *et al.*, 2005]).

It may appear strange that the heart of the program's reasoning does not lie in the evaluation function `eval` but in `pseudo` and `meta`: after all, their equivalent in a chess-playing software would trivially update a position by clearing a bit and setting another. However, the evaluation function's task is to evaluate the current knowledge. The updating algorithms compute the knowledge itself: thus it is important to infer as much information as possible in the process. In fact, one interesting point about this approach is that the updating functions and the evaluation function can be improved upon separately, increasing the program's performance without the need for the two components to have any knowledge of each other.

3 Game tree structure

Since a metaposition's evolution depends exclusively on the umpire's messages, clearly it becomes necessary to simulate the umpire's next messages if a game tree is to be constructed. Ideally, the game tree would have to include every possible umpire message for every available pseudomove so that it can be evaluated with a weighed algorithm keeping into account the likelyhood of each observation. Unfortunately, a quick estimate of the number of nodes involved rules out such an option. It is readily seen that:

- All pseudomoves may be legal (or they would not have been included by the generation algorithm), but most can be illegal for some game state.
- All pseudomoves that move to non-empty squares can capture (except for pawn moves), and we would need to distinguish between pawn and piece captures.
- Most pseudomoves may lead to checks.
- Some pieces may lead to multiple check types, as well as discovery checks.
- The enemy may or may not have pawn tries following this move.

A simple multiplication of these factors may yield several dozens potential umpire messages for any single move. But worst of all, such an estimate does not even take into account the possibility of *illegal moves*. An illegal move forces the player to try another move, which can, in turn, yield more umpire messages and illegal moves, so that the number of cases rises exponentially. Furthermore, the opponent's metamoves pose the same problem as they can lead to a large number of different messages.

- On the opponent's turn, most pieces can be captured, unless they are heavily covered or in the endgame.
- The king may typically end up threatened from all directions through all of the 5 possible check types.
- Again, pawn tries may or may not occur, and can be one or more.

For these reasons, any metaposition will be only evolved in exactly one way, and according to one among many umpire messages. This applies to both the player's pseudomoves and the opponent's hidden metamoves. There will be heuristics in place to pick a 'reasonable' message, and the more accurate this is, the more effective the whole system will get.

As a consequence, the tree's branching factor for the player's turns is equal to the number of potential moves, but it is equal to 1 for the opponent's own moves. This is equivalent to saying that the player does not really see an opponent, but acts like *an agent in a hostile environment*.

It should be noted that this is not the same assumption that Minimax algorithms make when they suppose that player MIN will choose the move that minimizes the evaluation function. Here we are not expecting the opponent to play the best possible move, but instead we assume an *average* move will be played, one that does not alter the state of the game substantially.

As a side effect, because only one possible umpire message for the opponent's metamove is explored, the metamove can be merged with the move that generated it, so that each level in the game tree no longer represents a ply, but a full move.

Interestingly, the branching factor for this Kriegspiel model is significantly smaller than the average branching factor for the typical chess game, seeing as in chess either player has a set of about 30 potential moves at any given time, and Kriegspiel is estimated to stand at approximately twice that

value (in theory; practice yields smaller values due to tighter defence patterns). Therefore, a two-ply game tree of chess will feature about $30^2 = 900$ leaves, whereas the Kriegspiel tree will only have 60. However, the computational overhead associated with calculating 60 metaposition nodes is far greater than that for simply generating 900 position nodes, and as such some kind of pruning algorithm may be needed.

3.1 Umpire prediction heuristics

[Bolognesi and Ciancarini, 2003], in tackling Kriegspiel endgames, where the artificial player's moves have only three possible outcomes (silent, check, illegal) and having to choose one to expand upon, rely upon the evaluation function to pick the most unfavorable option. However, even such a modest luxury seems beyond reach in the present work due to both the number of options and their different probabilities. The only remaining way is for us to propose a set of hard-coded heuristics that work well most of the time, and make sure that they will work reasonably even when they are proved wrong. Our player generates the umpire messages that follow its own pseudomoves in the following way.

- Every move is always assumed to be legal. Most of the time, an illegal move just provides information for free, so a legal move is usually the less desirable alternative.
- The player's moves do not generally capture anything, with the following exceptions:
  - Pawn tries. These are always capturing moves by their own nature.
  - Non-pawn moves where the destination square's Empty bit is not set, since the place is necessarily non-empty. This encourages the program to retaliate on captures.
  - After an illegal move, the agent may consider an identical move, but shorter by one square, as a capturing move.
- If any of the above apply, the captured entity is always assumed to be a pawn, unless pawns should be impossible on that square, in which case it is a piece.
- Pawn tries for the opponent are generated if the piece that just moved is the potential target of a pawn capture.

On the other hand, the following rules determine the umpire messages that follow a metamove.

- The opponent never captures any pieces, either. The constant risk that allied pieces run is considered by the evaluation function instead.
- The opponent never threatens the allied King. Again, King protection is matter for the evaluation function.
- Pawn tries for the artificial player are never generated.

The above assumptions are overall reasonable, in that they try to avoid sudden or unjustified peaks in the evaluation function. The umpire is silent most of the time, captures are only considered when they are certain, and no move receives unfair advantages over the others. There is no concept of a 'lucky' move that reveals the opponent's king by pure coincidence, though if that happens, our program will update its knowledge accordingly.

Even so, the accuracy of the prediction drops rather quickly. In the average middle game, the umpire answers with a non-silent message about 20-30% of the time. Clearly, the reliability of this method degrades quickly as the tree gets deeper, and the exploration itself becomes pointless past a certain limit. At the very least, this shows that any selection algorithm based on this method will have to weigh evaluations differently depending on where they are in the tree; with shallow nodes weighing more than deeper ones, and even so, exploration becomes fruitless past a certain threshold.

3.2 The selection algorithm

Now that the primitives have been discussed in detail, it is possible to describe the selection algorithm for the metaposition-based player. Several variants on this approach have been developed, optimizing the algorithm for fast play over the Internet Chess Club using such methods as pruning and killer-like techniques; this is its first and basic formulation.

The whole stratagem of metapositions was aimed at making traditional minimax techniques work with Kriegspiel. Actually, since MIN's moves do not really exist (MIN always has only one choice) if we use the compact form for the tree, with each node representing two plies, the algorithm resembles a *weighed maximax*. Maximax is a well-known criterion for decision-making under uncertainty. This variant is weighed, meaning that it accepts an additional parameter $\alpha \in]0, 1[$, called the *prediction coefficient*. The algorithm also specifies a maximum depth level k for the search. Furthermore, we define two special values, $\pm\infty$, as possible output to the evaluation function `eval`. They represent situations so desirable or undesirable that they often coincide with victory or defeat, and should not be expanded further.

Defining **Mt** as the set of all metapositions and **Mv** as the set of all possible chess moves, the selection algorithm makes use of the following functions:

- `pseudo`: $(\mathbf{Mt} \times \mathbf{Mv}) \rightarrow \mathbf{Mt}$, which generates a new metaposition from an existing one and a pseudomove, simulating the umpire's responses as described in the last section.
- `meta`: $\mathbf{Mt} \rightarrow \mathbf{Mt}$, which generates a new metaposition simulating the opponent's move and, again, virtual umpire messages.
- `generate`: $\mathbf{Mt} \rightarrow \mathbf{Vector_{Mv}}$, the move generation function.
- `eval`: $(\mathbf{Mt} \times \mathbf{Mv} \times \mathbf{Mt}) \rightarrow \mathbb{R}$, the evaluation function, accepting a source metaposition, an evolved metaposition (obtained by means of *pseudo*), and the move in between.

The algorithm defines a *value* function for a metaposition and a move, whose pseudocode is listed in Figure 1. The actual implementation is somewhat more complex due to optimizations that minimize the calls to *pseudo*.

It is easily seen that such a function satisfies the property that a node's weight decrease exponentially with its depth.

```
function value (metaposition met, move mov, int depth): real
begin
  metaposition met2 := pseudo(met, mov);
  real staticvalue := eval(met, mov, met2);
  if (depth ≤ 0) or (staticvalue = ± ∞)
    return staticvalue
  else
  begin
    //simulate opponent, recursively find
MAX.
    metaposition met3 := meta(met2);
    vector movevec := generate(met3);
    real best := max_{x∈movevec} value(met3, x, depth-1);
          //weighed average with parent's
value.
    return (staticvalue*α)+best*(1 − α)
  end
end.
```

Figure 1: Pseudocode listing for value function.

Given the best maximax sequence of depth d from root to leaf $m_1, \ldots, m_d$, where each node is provided with static value $s_1, \ldots, s_d$, the actual value of m_1 will depend on the static values of each node m_k with relative weight α^k. Thus, as the accuracy of the program's foresight decreases, so do the weights associated with it, and the engine will tend to favor good positions in the short run.

Parameter α is meant to be variable, as it can be used to adjust the algorithm's willingness to take risks, as well as our level of confidence in the heuristics that generate the simulated umpire messages. Higher values of α lead to more conservative play for higher reward in the short run, whereas lower values will tend to accept more risk in exchange for possibly higher returns later on. Generally, the player who is having the upper hand will favor open play whereas the losing player tends to play conservatively to reduce the chance of further increasing the material gap. Material balance and other factors can therefore be used to dynamically adjust the value of α during the game.

4 An evaluation function for metapositions

The evaluation functions for chess programs have usually three main components: material count, mobility, and position evaluation. A metaposition evaluation function, however, does not work on a single chessboard, but on an entity representing billions of chessboards, and may need to introduce equivalent, but different concepts. For example, our evaluation function currently has three main components that it will try to maximize throughout the game: *material safety*, *position*, and *information*.

4.1 Material safety

Material safety is a function of type $(\mathbf{Mt} \times \mathbf{Sq} \times \mathbf{Bool}) \rightarrow [0, 1]$. It accepts a metaposition, a square and a boolean and returns a safety coefficient for the friendly piece on the given square. The boolean parameter tells whether the piece has just been moved (as it is clear that a value of *true* decreases the piece's safety; statistically speaking, the risk of losing the piece being moved is much higher). A value of 1 means it is impossible for the piece to be captured on the next move, whereas a value of 0 indicates a very high-risk situation with an unprotected piece.

It should be noted, however, that material safety does not represent a probability of the piece being captured, or even an estimate of it; its result simply provides a reasonable measure of the exposure of a piece and the urgency with which it should be protected or moved away from danger.

4.2 Position

Our player includes the following factors into its evaluation function:

- A pawn advancement bonus. In addition, there is a further bonus for the presence of multiple queens on the chessboard.
- A bonus for files without pawns, and friendly pawns on such files.
- A bonus for the number of controlled squares, as computed with a special protection matrix. This factor is akin to mobility in traditional chess-playing software.

In addition, the current position also affects material rating, as certain situations may change the values of the player's pieces. For example, the value of pawns is increased if the player lacks sufficient mating material.

An additional component is evaluated when Darkboard is considering checkmating the opponent. A special function represents perceived progress towards winning the game, partly borrowed from [Bolognesi and Ciancarini, 2003], thus encouraging the program to push the opponent's pseudokings towards the edges of the chessboard.

4.3 Information

One of the crucial advantages of using metapositions lies in the ability to estimate the quality and quantity of information available to the player. In fact, because we are operating with a large superset of the information set which necessarily incorporates the current true state of the game, to acquire information simply means to aim towards reducing the size of the metaposition's position set; therefore, an indicator based on size (for example, the sum of all the pseudopieces on the chessboard) can enter the evaluation function and the player will strive towards states with reduced uncertainty. An approch such as Monte Carlo cannot do this, as its belief state works on a small subset of the information set wherein each single state is dogma when evaluated.

Our player will attempt to gather information about the state of the chessboard, as the evaluation function is designed to make information desirable (precisely, it is designed to make the lack of information undesirable) by reducing a function, which we call *chessboard entropy* (E), satisfying the following.

- The function's value increases after every metamove from the opponent, that is $(m_2 = \mathrm{meta}(m_1)) \Rightarrow E(m_2) \geq E(m_1)$.

- The function's value decreases after each pseudomove from the player, that is $(m_2 = \mathrm{pseudo}(m_1, x \in \mathbf{Mv})) \Rightarrow E(m_2) \leq E(m_1)$.

Therefore, the chessboard entropy is constantly affected by two opposing forces, acting on alternate plies. We can define $\Delta E(m, x), m \in \mathbf{Mt}, x \in \mathbf{Mv}$ as $E(\mathrm{pseudo}(\mathrm{meta}(m, x))) - E(m)$, the net result from two plies. Our program will attempt to minimize ΔE in the evaluation function. In the beginning, entropy increases steeply no matter what is done; however, in the endgame, the winner is usually the player whose chessboard has less entropy.

Darkboard's algorithm for computing entropy revolves around the age matrix, encouraging the program to explore squares with a higher age value (meaning that they have not been visited in a long time). Clearly, there are constants involved: making sure there are no enemy pawns on the player's second rank is more important than checking for their presence on the fifth rank.

5 Experimental results and conclusions

We remark that the ruleset used for our program is the one enforced on the Internet Chess Club, which currently hosts the largest Kriegspiel community of human players. Our metaposition-based Kriegspiel player, Darkboard, is currently, to the best of our knowledge, the only existing artificial player capable of facing human players over the Internet on reasonable time control settings (three-minute games) and achieve well above average rankings, with a best Elo rating of 1814 which placed it at the time among the top 20 players on the Internet Chess Club. We note that Darkboard plays an average of only 1.415 tries per move, and therefore it does not use the advantage of physical speed to try large amounts of moves.

Darkboard defeats a random-moving opponent approximately 94.8% of the time. It defeats a random player with basic heuristics (a player which will always capture when possible but otherwise move randomly) approximately 79.3% of the time; the rest are draws by either stalemate or repetition.

Darkboard won Gold medal at the Eleventh Computer Olympiad which took place from May 24 to June 1, 2006 in Turin. The player defeated an improved version of the Monte Carlo player described in [Parker *et al.*, 2005] with a score of 6-2.

In view of these results, we argue that using metapositions to evaluate a superset of the current game state rather than a subset of it yields very encouraging results for those games with strongly incomplete information and an extremely large belief state.

References

[Bolognesi and Ciancarini, 2003] A. Bolognesi and P. Ciancarini. Computer Programming of Kriegspiel Endings: the case of KR vs K. In J. van den Herik, H. Iida, and E. Heinz, editors, *Advances in Computer Games 10*, pages 325–342. Kluwer, 2003.

[Bolognesi and Ciancarini, 2004] A. Bolognesi and P. Ciancarini. Searching over Metapositions in Kriegspiel. In J. van den Herik and H. Iida, editors, *Computer and Games 04*, volume (to appear) of *Lecture Notes in Artificial Intelligence*. Springer, 2004.

[Ciancarini *et al.*, 1997] P. Ciancarini, F. DallaLibera, and F. Maran. Decision Making under Uncertainty: A Rational Approach to Kriegspiel. In J. van den Herik and J. Uiterwijk, editors, *Advances in Computer Chess 8*, pages 277–298. Univ. of Rulimburg, 1997.

[Frank and Basin, 1998] I. Frank and D. Basin. Search in games with incomplete information: A case study using bridge card play. *Artificial Intelligence*, 100(1-2):87–123, 1998.

[Parker *et al.*, 2005] A. Parker, D. Nau, and VS. Subrahmanian. Game-Tree Search with Combinatorially Large Belief States. In *Int. Joint Conf. on Artificial Intelligence (IJCAI05)*, volume (to appear), Edinburgh, Scotland, 2005.

[Sakuta, 2001] M. Sakuta. *Deterministic Solving of Problems with Uncertainty*. PhD thesis, Shizuoka University, Japan, 2001.