A Skat Player Based on Monte Carlo Simulation

Sebastian Kupferschmid and Malte Helmert

Albert-Ludwigs-Universität Freiburg, Germany.
{kupfersc, helmert}@informatik.uni-freiburg.de

Abstract. We apply Monte Carlo simulation and alpha-beta search to the card game of *Skat*, which is similar to Bridge, but different enough to require some new algorithmic ideas besides the techniques developed for Bridge. Our Skat-playing program integrates well-known techniques such as *move ordering* with two new search enhancements. *Quasi-symmetry reduction* generalizes symmetry reductions, popularized by Ginsberg's Partition Search algorithm, to search states which are "almost equivalent". *Adversarial heuristics* generalize ideas from single-agent search algorithms like A* to two-player games, leading to guaranteed lower and upper bounds for the score of a game position. Combining these techniques with state-of-the-art tree search algorithms, our program determines the game-theoretical value of a typical Skat hand (with perfect information) in 10 milliseconds.

1 Introduction

Although mostly unknown in the English-speaking world, the game of Skat is the most popular card game in continental Europe, surpassed in world-wide popularity only by Bridge and Poker. With about 30 million casual players and about 40,000 people playing at a competitive level, Skat is mostly a German phenomenon, although national associations exist in twenty countries on all six inhabited continents. It is widely considered the most interesting card game for three players.

Despite its popularity, Skat has not been studied extensively by the AI community. This is not due to lack of challenge, as Skat is definitely a game of skill — significant experience is required to reach tournament playing strength — and all existing computer implementations play rather poorly. In this paper, we explore how an existing approach for playing Bridge, *Monte Carlo simulation* using a fast solver for perfect information games, can be applied to the game of Skat.

The paper is structured as follows. Section 2 briefly introduces the rules of Skat. In Section 3, we review the idea of Monte Carlo simulation for card games. Section 4 describes the general architecture of our Skat player, followed by the central Section 5, which describes a fast algorithm for computing the outcome of Skat games with perfect information. Section 6 presents empirical results and Section 7 concludes.

2 Skat

Skat is a three-player game played with 32 cards, a subset of the usual Bridge deck. At the beginning of a game, each player is dealt ten cards, which must not be shown or

communicated to the other players. The remaining two cards, called the *skat*, are placed face down on the table. Like in Bridge, each hand is played in two stages, *bidding* and *card play*.

The bidding stage determines the alliances for this hand: The successful bidder, henceforth called the *declarer*, plays against the other two. Often, several players compete to become the declarer. In this event, the winner of the bidding process mostly depends on the number of jacks a player holds, and on their suits. Players may also improve their bids by declaring some special contracts (such as *hand*, *schneider* and *schwarz* games), but these are nuances that we will not discuss further. We point out that, different to Bridge, bidding does *not* have a significant influence on the number of tricks needed to win the deal, with some minor exceptions.

We will not explain the bidding process further and refer to the official rules [6] for details. The declarer decides on the kind of game, for which there are six possibilities: *grand* games, *null* games, and *suit* games for each of the four suits (♣, ♠, ♡, ♢).

Card play proceeds as in Bridge, except that the trumps and card ranks are different. In grand games, the four jacks are the only trumps. In suit games, the four jacks and the seven other cards of the selected suit are trumps. There are no trumps in null games. Non-trump cards are grouped into suits as in Bridge. Each card has an associated *point value* between 0 and 11, and the declarer must score more points than the opponents (i.e. at least 61 points) to win. Null games are an exception and follow *misère* rules: the declarer wins iff he scores no trick. Trumps, ranks and point values of the cards are illustrated in Fig. 1.

Before declaring the game, the declarer may pick up the skat and then discard any two cards from his hand, face down. These cards count towards the declarer's score.

3 Monte Carlo Simulation

The main algorithmical problem when dealing with card games like Bridge or Skat is *uncertainty*. For perfect-information games like Chess, efficient algorithms exist that could be readily applied if it were not for the fact that the opponents' cards are hidden. In fact, the state space of these card games is comparatively small, and it is not too difficult to compute an optimal strategy with knowledge of the deal. However, taking randomness into account is much more challenging (cf. the work by Koller and Pfeffer [8]).

Ranks

Grand games	♣J, ♠J, ♡J, ♢J (trumps)
	A, 10, K, Q, 9, 8, 7 (non-trumps)
Suit games	♣J, ♠J, ♡J, ♢J, A, 10, K, Q, 9, 8, 7 (trumps)
	A, 10, K, Q, 9, 8, 7 (non-trumps)
Null games	A, K, Q, J, 10, 9, 8, 7

Point values

A: 11, 10: 10, K: 4, Q: 3, J: 2, 9: 0, 8: 0, 7: 0

Fig. 1. Ranks and point values of Skat cards. Higher ranking cards are listed further to the left.

Monte Carlo approaches, first proposed in this context by Levy [10] and later implemented by Ginsberg [5] in his Bridge-playing program GIB, reduce the problem to the perfect information case using the following strategy: Whenever the computer player is asked to play a card, it generates a set of deals which are consistent with previous play. Each of these deals is then completely analyzed by a fast solver for perfect information games. In theory, this can be done with a traditional alpha-beta search engine. The results of these analyses are then used to vote on the card to play in the actual (uncertain) game.

The Monte Carlo approach has two fundamental problems. The first problem is that the samples might not be representative of the real card distribution. This is not so much caused by the fact that only a limited number of deals are analyzed, because the law of large numbers guarantees that this statistical error can be made arbitrarily small. The real issue is that not all deals should be generated with equal probability, because different distributions are not equally plausible given the previous course of play.

To reflect this, we would need to take into account the *conditional probability* that an opponent will play a given card given a certain deal and previous play. For example, if the declarer starts the game with ♣A and the other players follow suit with ♣7 and ♣10, it is highly unlikely that the third player still holds a clubs card (except for ♣J, which is part of the trump suit, not the clubs suit). However, it is difficult to quantify information of that kind, both in theory and in practice.[1]

The second fundamental problem of the Monte Carlo approach is that even if *all* possible deals are analyzed and the conditional probabilities are correct,[2] the algorithm does not play perfectly. Intuitively, the reason for this is the fact that the correct card to play may depend on information that the player cannot possibly know. Formally, this problem is discussed extensively by Frank and Basin [2].

Despite these fundamental limitations, Monte-Carlo-based approaches have been successful in the Computer Bridge world. Indeed, most current systems rely on sampling methods to some extent. We believe that they should be at least as effective for Computer Skat, and possibly more so, because the bidding phase of a Skat game allows for much less information gathering than in Bridge.

4 General Architecture

Our Skat player consists of two parts, a bidding engine and a card play engine. The bidding engine is responsible for determining the highest bid that the player is willing to make and for deciding which two cards to discard in case it wins the bid. The card play engine is responsible for the actual card play after the game has been declared. At its core is a fast algorithm for solving Skat games with perfect information. We call this component the *double dummy solver*, borrowing from Bridge terminology, even though the term *dummy* is Bridge-specific. The double dummy solver is explained in

[1] *Mixed Strategy Nash Equilibria* are the most commonly applied theoretical solution concept for such games [3].

[2] This condition requires an exact mental model of the opponents, and is thus not practically possible for human opposition.

the following sections, while the rest of this section is dedicated to the bidding and card play engines.

4.1 Bidding Engine

In theory, it is possible to implement the bidding engine by Monte Carlo sampling using the following strategy: First, select N random deals. Then, for each of the six kinds of games and each possible way of discarding two cards, query the double dummy solver to decide whether or not the game can be won. However, this requires $6 \cdot \binom{12}{2} \cdot N = 396N$ queries, which is prohibitively high even for a modest number of samples.

Typically, the choice of cards to discard is straight-forward, as most candidates can be eliminated by simple rules of thumb. A mixed approach that computes Monte Carlo samples for each kind of game and implements rules for discarding requires only $6N$ queries.

We have instead adopted a completely rule-based approach for both bidding and discard procedure. The rules were generated by the following learning algorithm. First, we used the double dummy solver to analyze 126,000 deals, where both the discarded cards and the kind of game were randomized. For each of the resulting hands of the computer player, the algorithm evaluated a number of hand-crafted features, e.g. number of jacks and length of each suit, and paired these with the outcome of the game (1 for win or 0 for loss). Then, a Least Mean Squares algorithm fitted a linear function from the feature space to the real numbers. The resulting *success estimator* was supposed to estimate the winning probability given the features of a hand. It was then used in place of the double dummy solver in the bidding stage of the game, so that instead of $396N$ calls to the double dummy solver, a corresponding number of computations of the success estimate was needed, which could be computed sufficiently fast.

Unfortunately, the resulting bidding behavior leaves something to be desired. Although reasonable choices were made most of the time, some decisions were truly puzzling. While we do not discuss the bidding engine further, as this part contains no technical innovations and requires some domain knowledge for understanding the choice of features, we note that it is currently the Achilles heel of the Skat player.

4.2 Card Play Engine

As noted before, the card play engine is based on Monte Carlo simulations using the double dummy solver. In Ginsberg's GIB, a score is calculated for each card and each sample deal. The algorithm plays the card with the highest average score. A similar scheme can be used for Skat. However, it is not immediately clear how the score should be measured. Counting the point total achieved by the computer player is not reasonable, since playing a card that reliably achieves a point total of 65 (and thus a win) is preferable to playing a card that leads to a total of 80 most of the time but rarely drops to 55 (and thus a loss).

On the other hand, simply counting whether or not the deal is won is also problematic, as there is no incentive for the algorithm to win the game by a score of 100:20, rather than, say, 63:57. Thus it will willingly give away points to the opponent as long

as it does not see its victory endangered.[3] Because of statistical error and the inaccuracies of Monte Carlo methods mentioned earlier, this can lead to situations where an easy victory is cast away lightly.

To avoid both problems, we follow a combined approach. Winning a hand is most important, so the set of winning cards of each sample deal is computed first. Those cards which win a maximal number of sample deals are further analyzed by computing the average point total across all sample deals. The card play engine finally selects a card which maximizes this value. This leads to card play that prefers safety to point accumulation, but accumulates points where safely possible.

To support this approach, the double dummy solver can run in a fast *qualitative mode*, in which it only determines whether or not a given card is a winner, and in a slower *quantitative mode*, in which it computes exact scores for cards. We describe these in the following section. Null games are always solved in qualitative mode as they end as soon as the declarer wins a trick. They are not computationally challenging and we do not describe them further, assuming grand or suit games in the following.

5 Double Dummy Solver

When running in qualitative mode, the double dummy solver uses a zero-window alpha-beta search to determine whether or not the score for the declarer when playing a given card is at least 61. When running in quantitative mode, the solver uses the MTD framework proposed by Plaat et al. [12]. Specifically, we employ the MTD(0) algorithm: First, we determine whether or not the declarer can achieve a single point. If the answer is positive, a new search determines whether or not he can achieve two points[4], three points, and so on, until the answer is negative. Although it is somewhat counter-intuitive that this procedure should be an improvement over standard alpha-beta search, the higher number of cut-offs, combined with the use of a transposition table for storing intermediate results, usually makes it much faster. Most recomputations of subtree values only require a transposition table lookup, reducing the cost of re-search. In our experiments, zero window search outperformed standard alpha-beta by an order of magnitude when using a transposition table.

Therefore, regardless of mode, we can in the following assume that we are conducting a zero-window search. For clarity of presentation, we will only consider the most common search window $[60, 61]$, although other search bounds do occur in quantitative mode. The basic search algorithm, without any search enhancements, is shown in Fig. 2. In the rest of this section, we discuss four enhancements to the basic search algorithm which lead to significant speed-ups, concluding our presentation with a refined search algorithm that includes all enhancements.

[3] This is a general problem for game playing algorithms searching to the end of the game. Schaeffer reports similar "unreasonable" behavior in his checkers-playing program *Chinook*, which uses perfect endgame databases. The Chinook team went to some lengths to change this behavior [13].

[4] This is actually redundant, as scores of 1 or 119 are impossible.

```
def search(p):
  if p isa leaf position:
    return p.declarer_score ≥ 61
  else if p isa declarer node:
    for q in succ(p):
      if search(q) = true:
        return true
    return false
  else:
    for q in succ(p):
      if search(q) = false:
        return false
    return true
```

Fig. 2. Basic search algorithm.

5.1 Transposition Table

The first and obvious enhancement to Fig. 2 is the use of a transposition table. However, using a transposition table efficiently in this setting requires some care. At every stage of the game, the current position can be adequately represented by the player to move, the remaining cards, the cards in the current trick (if any), and the *running score*, i.e. the number of points won by the declarer in previous tricks. The running score is an important part of the position because it influences the evaluation of the position (win/loss and exact point value). However, it has *no* effect on the optimal strategy for the rest of the game, the *subgame* rooted at this position in game theory terminology.

Therefore, to keep the transposition table small and allow as many lookups as possible, it is desirable not to consider the running score a part of the position information. This means that it is not sufficient to store win/loss values in the transposition table, not even for the qualitative solver: A subgame in which the declarer can achieve 40 points can be either a win or a loss, depending on the running score as this subgame is reached. Thus, the transposition table must always store exact point values or bounds on exact values, not boolean results.

Using transposition tables in this way, we get a simple cut-off criterion for search nodes: A subgame is not searched further if the transposition table shows that the total of the running score and the lower bound on the future score is at least 61, or the total of the running score and the upper bound on the future score is at most 60.

5.2 Move Ordering

It is commonly known that alpha-beta search performance is dramatically influenced by the order in which the different move alternatives are considered [11].

Typical implementations of alpha-beta search use three heuristics for finding a good ordering, i.e. one where an optimal move is considered early: *transposition table moves*, *history heuristic* and *killer heuristic*. All these techniques are roughly based on the idea that a move which is good in a certain context is often good in other contexts. We

have implemented the first of these techniques: If playing a certain card leads to a cut-off in some subgame, this card is always considered first when this subgame is later reexamined with different search bounds.

The remaining cards are ordered with the aim of reducing branching. If there is currently no card on the table, we prefer playing a suit of which the other players hold at least one card (so that they must follow suit), but only few cards (so that their choice is limited). More to the point, for each card we multiply the number of allowed answers for the other two players, preferring cards which minimize this value. Within a suit, cards of higher rank are preferred.

In our experiments, this ordering heuristic reduces the average number of investigated search nodes by a factor of 3.45, while reducing the average search time by a factor of 3.02, compared to the original, arbitrary move ordering. We analyze the impact of move ordering in more detail in Section 6, together with the effect of the other search enhancements to be introduced now.

5.3 Quasi-Symmetry Reduction

Ginsberg reports that his Bridge playing program GIB [5] is accelerated by an order of magnitude by replacing alpha-beta search with his *Partition Search* algorithm [4]. Partition Search aims at increasing the number of subgames that can be solved by transposition table lookups. It does so by not storing single game positions but *equivalence classes* of game positions in the transposition table. This is very effective in Bridge as the number of equivalent positions can be expected to be high.

Many of the equivalences of Bridge positions are due to the fact that only the *relative* rank of cards is important for determining optimal play; the *absolute* rank is irrelevant. We say that two cards held by the same player are *rank-equivalent* iff they are in the same suit and no card on the table or in another player's hand is ranked between them.

Unfortunately, unlike Bridge, it is usually the case in Skat games that *all* rank-equivalent cards must be considered because of different point values. In a Bridge game, a player that holds both ♠K and ♠Q need never play the king before the queen (or vice versa). The same is not true of a Skat position. For example, if the player can win the trick playing the king, eventually winning the deal by a margin of 61:59 points, playing the queen instead might lose the deal. Especially if the difference in value between the two cards is greater than one, for example in the case of ♠10 and ♠K, the lines of play that begin with these cards often look completely different.

However, it can be proven that if two cards are rank-equivalent, then the difference between the values of the subgames started by playing either of these cards is bounded by their difference in point value [9]. Thus, if we can prove that playing ♠10 results in a declarer score of 72 and ♠K is no longer in play, then playing ♠Q results in a declarer score in the interval $[65, 79]$, since in this situation ♠10 and ♠Q are rank-equivalent, and the difference in point value is $10 - 3 = 7$.

We use rank-equivalence for a technique we call *quasi-symmetry reduction*, which decreases the branching factor of interior nodes of the search tree: Whenever the search algorithm considers playing a card c which is rank-equivalent to a previously considered card c', we fetch the transposition table entry for the position reached by playing

c' and check if there is any hope in playing c instead of c'. For example, if the transposition table shows that playing ♠Q at some declarer node yields at most 57 points, then playing ♠K can yield at most 58 points, so that the move need not be considered.

In our experiments, exploiting quasi-symmetries significantly reduces the number of search nodes. However, much of this gain is countered by an increased cost per node for rank-equivalence checking and transposition table lookups. The most efficient version of the algorithm, which is the one we report on, only exploits quasi-symmetries for cards whose point values differ by at most one.

In our experiments, quasi-symmetry reduction reduces the average number of search nodes by a factor of 2.38 and average running time by a factor of 2.03.

5.4 Adversarial Heuristics

As a final search enhancement, the double dummy solver uses a forward pruning technique which we will now describe. In Section 5.1, we explained that whenever a position is re-explored during search, the search algorithm fetches a lower bound L and upper bound U on the declarer score in this subgame from the transposition table. If M is the running declarer score, then the subgame is not searched further if $M + L \geq 61$ or $M + U \leq 60$. Our forward pruning technique extends this early termination check to positions which are *not* present in the transposition table. To this end, we must compute (preferably narrow) bounds L and U for arbitrary subgames.

How can such bounds be calculated? In single-agent search problems, lower bounds on the actual search cost are typically computed by *relaxing* the problem at hand, i.e. by increasing the set of allowed moves. For example, the minimal weighted matching heuristic for Sokoban [7] can be interpreted as the length of an optimal solution to a relaxed problem where boxes may be moved to adjacent empty squares regardless of the position of the man. The Manhattan heuristic for the $n^2 - 1$ puzzle can be similarly understood as the length of an optimal solution to a relaxed problem where tiles may always be moved to adjacent positions, even if these are occupied.

When extending these ideas to an adversarial search context, care must be taken to correctly reflect the role of the MAX and MIN players. For any given subgame, we can compute an *upper bound* to the score of the MAX player by *extending* the set of possible moves for MAX and/or *reducing* the set of possible moves for MIN. Conversely, a *lower bound* can be computed by extending the set of possible moves for MIN and/or reducing the set of possible moves for MAX. Any such modification leads to correct bounds that can be exploited during search without compromising the validity of the search in any way, unlike common forward pruning techniques such as *null-move pruning* in Chess or Buro's *ProbCut* [1] in Othello. We call bounds derived in such a way *adversarial heuristics* because of their similarity to heuristics used in single-agent (non-adversarial) search.

The key to good adversarial heuristics is modifying the sets of allowed moves in such a way that the resulting bounds are reasonably narrow, but cheap to compute. For lower bounds on the declarer score in Skat, the following two modifications of the game rules satisfy this criterion:

1. The declarer may only play cards that are guaranteed to win the trick. If this means that he has no legal moves, the opponents may claim the remaining points.

```
def search(p):
  if p isa leaf position:
    return p.declarer_score ≥ 61
  else if p isa declarer node:
    if p in transposition_table:
      (L, U) := transposition_table(p)
    else:
      (L, U) := adversarial_heuristics(p)
    if M + L ≥ 61:
      return true
    if M + U ≤ 60:
      return false
    for q in order_moves(succ(p)):
      if q ∼ q' for q' considered earlier:
        (L',U') := transposition_table(q')
        if M + U' + δ(q,q') ≤ 60:
          continue
      if search(q) = true:
        return true
    return false
  else:
    ... {analogous to declarer node case}
```

Fig. 3. Search algorithm with all search enhancements; $\delta(q, q')$ denotes the point difference between the two cards being played. We omit some details like updating the transposition table, which are handled in the standard way.

The rationale between this modification is that the optimal strategy for the opponents becomes difficult to compute once they are able to control the game. We eliminate this expensive computation by requiring the declarer to force the game.

2. In addition to normal moves, an opponent may swap the point values of two cards in his hand before playing a card, provided that the two cards are in the same suit.

This modification eliminates a strategical dilemma for the opponents: In some situations, it is difficult to decide whether they should play a card of minimal *point value* or minimal *rank*. For example, consider a diamonds game where the declarer plays $\clubsuit J$ and is thus guaranteed to win the trick. The first opponent holds two trumps, $\heartsuit J$ and $\diamondsuit A$. In some situations, it is preferable to play $\heartsuit J$, only losing two points to the declarer. In other situations, it is better to play $\diamondsuit A$, losing eleven points to the declarer but keeping the higher-ranked card in order to win a trick later. In the modified game, the best reply is obvious: Swap the point values of the ace and jack, so that the ace is worth two points and the jack is worth eleven points, then play the ace.

We point out that swapping the point values of cards is rarely needed, because rank ordering and point value ordering are consistent for all non-trump suits, and in the case of grand games even for the trump suit. Thus, the second modification does not usually have a large impact on the quality of the bounds.

An experienced Skat player will notice that computing an optimal strategy in the modified game is almost trivial, except for situations where an opponent need not follow suit. Indeed, the value of a game position in the modified game can be computed in time $O(N)$, where N is the number of remaining cards. For details on how this can be done, we refer to the first author's master's thesis [9].

Similar ideas can be applied to obtain *upper bounds* on the declarer score. However, this case is slightly more complicated, so we refer to the first author's master's thesis again for details.

In our experiments, using adversarial heuristics reduces the average number of search nodes by a factor of 1.80 and the average run time by a factor of 1.58.

The complete search algorithm, including all enhancements, is shown in Fig. 3.

6 Experiments

In the previous section, we described the implementation of our Skat double dummy solver, focusing on three central search enhancements: move ordering, quasi-symmetry reduction, and adversarial heuristics. In this section, we provide a more detailed empirical analysis of the performance gains offered by these features.

In practice, it is not sufficient to consider the effectiveness of a search enhancement in isolation. It is quite possible that a given enhancement leads to a dramatic improvement in run time by itself, but offers no gain when implemented together with other features. For this reason, we evaluated all possible combinations of move ordering (MO), quasi-symmetry reduction (QR) and adversarial heuristics (AH), including the empty set. All configurations used a transposition table.

Features			Nodes in 1,000			Run Time		
MO	**QR**	**AH**	**Mean**	**Dev.**	**Med.**	**Mean**	**Dev.**	**Med.**
			2772	8853	181	0.84s	3.11s	0.04s
×			804	3669	29	0.28s	1.42s	0.01s
	×		1163	3457	102	0.41s	1.36s	0.03s
		×	1538	5223	57	0.53s	2.02s	0.02s
×	×		317	1499	17	0.12s	0.65s	0.01s
×		×	626	3182	12	0.20s	1.10s	0.01s
	×	×	658	2152	34	0.25s	0.87s	0.01s
×	×	×	244	1300	7	0.11s	0.60s	0.01s

Fig. 4. Mean value, standard deviation and median of node count and running time for 100,000 randomly generated Skat games.

Fig. 4 shows that the three enhancements work well in combination. Although the speedups are not completely orthogonal, each of the three features is a useful addition to all configurations without it. The results are based on 100,000 randomly generated suit games. Grand games are easier, null games much easier to solve. The high standard deviations and low medians show that the results are far from being normally distributed.

Most deals are solved very quickly, but occasional outliers heavily influence the average case performance.

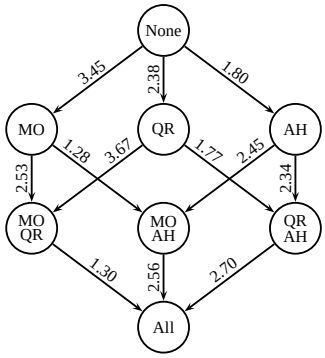

Fig. 5. Node count reductions for the various search enhancements.

Fig. 5 depicts a lattice illustrating the usefulness of adding each feature to any configuration. The arrow labels show the reduction of average node count achieved when going from one configuration to another. The figure shows that there are some diminishing returns, but the general picture is quite positive.

7 Conclusion

We have shown that by enhancing a state-of-the-art game-playing algorithm with a number of suitable search enhancements, it is possible to build a fast double dummy solver for the game of Skat. Of course, in practice we are not just interested in the performance of the double dummy solver, but also in the quality of play of the overall system.

This is somewhat harder to quantify because such experiments are difficult to automate. We played 18 games against human and machine opposition. Against two human players of moderate strength, the system ended on a close second place. Post-mortem analysis revealed flawless card play but improvable bidding behavior. Against two computer players[5] the system played very convincingly, winning every single game when it played as a declarer and all games but one when it played in the opposing party [9].

The logical next step for future work is to improve the bidding engine. In theory, the general approach of learning rules from a set of features and self-play data seems reasonable. However, our choice of features and learning algorithm might not be the best possible. Alternatively, hand-crafted rules could be used, but this is tedious and requires expert domain knowledge.

[5] Played by **XSkat 3.4**; cf. `http://www.xskat.de/`, a program with rule-based card play.

Another possible direction for further study is the investigation of alternative search algorithms such as proof number search or B^*. The classical drawback of these approaches is their high memory consumption, but in Skat games, all visited positions can easily be kept in memory.

Finally, it would be interesting to apply the search enhancements we introduced to other games. Quasi-symmetry reduction is a potentially useful technique for all games where points are accumulated in the course of play, which includes, but is not limited to, all trick-based card games. Adversarial heuristics can be usefully applied in a similar way to classical heuristics whenever complete solution of a game is feasible. We see a good potential for other games amenable to Monte-Carlo approaches, for games with a strong threat structure like Go-Moku or Hex, and for Amazon subgames.

References

[1] Michael Buro. ProbCut: An effective selective extension of the alpha-beta algorithm. *ICCA Journal*, 18(2):71–76, 1995.

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

[3] Drew Fudenberg and Jean Tirole. *Game Theory*. MIT Press, 1991.

[4] Matthew L. Ginsberg. Partition search. In *Proc. AAAI-96*, pages 228–233, 1996.

[5] Matthew L. Ginsberg. GIB: Steps toward an expert-level bridge-playing program. In *Proc. IJCAI-99*, pages 584–593, 1999.

[6] International Skat Players Association. Skat order 1999. Web site: http://www.ispaworld.org/, 2003.

[7] Andreas Junghanns and Jonathan Schaeffer. Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artificial Intelligence*, 129(1–2):210–251, 2001.

[8] Daphne Koller and Avi Pfeffer. Representations and solutions for game-theoretic problems. *Artificial Intelligence*, 94(1–2):167–215, 1997.

[9] Sebastian Kupferschmid. Entwicklung eines double dummy skat solvers – mit einer anwendung für verdeckte skatspiele. Master's thesis, University of Freiburg, 2003.

[10] David N. L. Levy. The million pound bridge program. In David N. L. Levy and Donald F. Beal, editors, *Heuristic Programming in Artificial Intelligence — The First Computer Olympiad*, pages 95–103. Ellis Horwood, 1989.

[11] Judea Pearl. *Heuristics — Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.

[12] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin. Best-first fixed-depth minimax algorithms. *Artificial Intelligence*, 87(1–2):255–293, 1996.

[13] Jonathan Schaeffer. Search ideas in Chinook. In Jaap van den Herik and Hiroyuki Iida, editors, *Games in AI Research*, pages 19–30. Universiteit Maastricht, The Netherlands, 2000.