Using Payoff-Similarity to Speed Up Search

Timothy Furtak and **Michael Buro**
Department of Computing Science, University of Alberta
Edmonton, Alberta, Canada T6G 2E8
{furtak|mburo}@cs.ualberta.ca

Abstract

Transposition tables are a powerful tool in search domains for avoiding duplicate effort and for guiding node expansions. Traditionally, however, they have only been applicable when the current state is exactly the same as a previously explored state. We consider a generalized transposition table, whereby a similarity metric that exploits local structure is used to compare the current state with a neighbourhood of previously seen states. We illustrate this concept and forward pruning based on function approximation in the domain of Skat, and show that we can achieve speedups of 16+ over standard methods.

1 Introduction

In the past 20 years there have been great successes for AI in popular abstract games. For example, we now have programs running on personal computers that can defeat the best humans at chess, checkers, Scrabble, Othello, and backgammon; computer Go has been revolutionized by Monte Carlo tree search; and computer poker has advanced to the point where humans no longer dominate. In most perfect information games the α-β algorithm and its various enhancements have been instrumental to these achievements. One important idea is to use transposition tables that store information about previous search results to improve move sorting in subsequent search iterations and to avoid repeating computations in case a state is reached via different move sequences.

In this paper we first introduce a more general concept of sharing data between search trees — payoff-similarity — that can save search effort in game domains in which encountered subtrees may be identical but player payoffs can be different. We then show how payoff-similarity can be used in a popular card game to speed up double dummy solver (DDS) computations by almost an order of magnitude. Finally, we discuss further search reductions based on approximating DDS values and forward pruning, and conclude the paper with ideas for future work.

2 Payoff-Similarity

By definition, a transposition table is used to detect when search reaches a state that is identical to one that has previously been explored, usually via a different path (i.e. a *transposition*). However the functional benefit of a transposition table lies in how it relates the *value* of a previously seen state p to the value of the current state c. Namely, if $c = p$ then $V(c) = V(p)$. More generally, one may consider only the portions of c and p that affect whether their values differ.

This idea leads to a further generalization that if we can compute a bound on the difference between the values of two (arbitrary) states, then we could potentially extract information about the value of the current state from *all* previously seen states. Practically this is infeasible, but in certain domains it is possible to exploit local structure to attain nontrivial bounds relating the values of states within a neighbourhood of each other.

Consider a two-player zero-sum perfect information game in which we want to relate the minimax values $V(p)$ and $V(c)$ of two states p and c, with subtrees T_p and T_c. If $p = c$ and $V(p)$ has already been determined and stored in a transposition table, then when reaching c we can use this information and return $V(c) = V(p)$ immediately, without searching T_c. If we have not encountered c before, T_c needs to be searched.

Now suppose $c \neq p$ but T_c and T_p are similar. Then we may be able to bound $|V(c) - V(p)|$, and knowledge of $V(p)$, or a bound on $V(p)$, could produce α or β cuts and save work.

While finding bounds for $|V(c) - V(p)|$ small enough to create cuts may be hard in general, for particular types of trees this goal is attainable. Suppose, for instance, that T_c and T_p are structurally equivalent, i.e. there is an isomorphism between T_c and T_p that respects which player is to move. The payoffs in corresponding leaves may be different. We call such states c and p **payoff-similar** and can prove the following statement:

Theorem 1. *Let states s and s' be payoff-similar and $|V(l_i) - V(l'_i)| \leq \delta$ for all corresponding leaf pairs (l_i, l'_i) in subtrees T and T' rooted in s and s', respectively. Then $|V(s) - V(s')| \leq \delta$.*

Proof. Because s and s' are payoff-similar, T and T' are structurally equivalent. We proceed by induction on the height of corresponding nodes in T and T'. For corresponding leaves l and l' we know $|V(l_i) - V(l'_i)| \leq \delta$. Now suppose the claim is true for all corresponding node pairs of height $\leq h$. Consider corresponding states s and s' with height $h + 1$. W.l.o.g. let s and s' be states with max to move and successors $s_1, \ldots, s_k$ and $s'_1, \ldots, s'_k$. Then, applying the induction

hypothesis yields:

$$V(s') = \max_i(V(s'_i)) \leq \max_i(V(s_i) + \delta)$$
$$= (\max_i(V(s_i))) + \delta = V(s) + \delta$$

and analogously $V(s') \geq V(s) - \delta$, and therefore $|V(s) - V(s')| \leq \delta$. □

2.1 Seeding the Transposition Table

In a manner similar to building an endgame database, we may pre-populate the transposition table with a selection of states for which we have computed their exact value (or possibly just lower and upper bounds). If our domain is sufficiently amenable then search need not directly encounter any of these "seeded" positions, as long as it gets close enough to make use of their values via the similarity bounds. This can allow for a standard endgame database to be replaced by a transposition table of equivalent efficacy but with significantly fewer entries. Thus we can pay a one-time cost to compute TT entries and then use those results for all future searches.

2.2 Application Domains and Related Work

The payoff-similarity property occurs often in trick-taking card games such as Hearts, Bridge, Sheepshead, and Skat. In these domains certain cards may become "power-equivalent", in terms of their ability to win tricks. For example, if one player is holding ♠78, then (from a perfect-information perspective) playing ♠7 is equivalent to playing ♠8. Less obviously, we may transpose into a state where the relative ranks of each card are the same as in a previously seen state. In games such as Bridge where only the number of tricks made is important, positions may be converted into a "canonical" form, by relabelling card ranks.

The Partition Search algorithm as described in [Ginsberg, 1996] takes such a canonical representation a step further, by storing *sets* of states with the same value. For example, the value of a given position may not depend upon which players hold certain low spades — those states can then be merged into one set, even though they have different canonical representations since the card ownership is different. Without going into detail, partition search backs up a set of equivalently-valued states based on whether a winning set is reachable or whether the current player is constrained to reach a losing set.

Our framework is more analogous to nearest-neighbour classification in that, instead of constructing an explicit representation of the sets, we implicitly define them in terms of previously seen states. Rather than attempting to hand-craft complicated partitioning functions (as with partition search) we require only a similarity metric, which may be significantly easier to construct. We leave the problem of combining the two approaches as future work.

3 Application to Skat

Unlike in Bridge, where the value of a position depends only on how many tricks each player can make (and thus all states with the same canonical representation have the same value), the value of a Skat position also depends on the *value* of those cards. We can say that Skat and related card-games such as Sheepshead and Pinochle have the payoff-similar property in the non-trivial sense, where δ may be greater than 0.

3.1 Skat Rules

The following description is adapted from [Buro *et al.*, 2009]. Skat is a trick-taking card game for 3 players. It uses a short 32-card playing deck, similar to the standard 52-card deck except that cards with rank 2 through 6 have been removed. A hand begins with each of the 3 players being dealt 10 cards, with the remaining 2 cards (the *skat*) dealt face-down.

After an initial bidding phase, one player is designated the *soloist* and the other two players (the *defenders*) form a temporary coalition. The soloist then announces the *game type*, which will determine the trump suit and how many points the soloist will earn if she wins the game. There is one game type for each of the four suits (♢♡♠♣), in which the named suit and the four jacks form the trump suit. These four types are referred to as *suit games*. Another game types is *grand*, in which *only* the 4 jacks are trump.

Once the soloist announces the game type, card-play begins. This phase consists of 10 tricks, which are played in a manner similar to bridge and other trick-taking card games. The soloist's objective is to take 61 or more of the 120 available *card points*. Each card in the game is worth a fixed amount: Aces are worth 11 points and Tens are worth 10. Kings, Queens, and Jacks are worth 4, 3, and 2 points respectively. 7s, 8s, and 9s are not worth any points. We have omitted many of Skat's more detailed rules from this summary; for a more thorough treatment, the interested reader is referred to `www.pagat.com/schafk/skat.html`.

3.2 Computer Skat

At present, the strongest known card-play technique for Skat programs is perfect information Monte Carlo (PIMC) search, which is capable of achieving expert-class playing strength [Buro *et al.*, 2009]. This consists of generating perfect information worlds that are consistent with observed actions (possibly incorporating some form of inference to bias or weight the likelihood of each world) and computing the value of each move. The move that has the best value, averaged over all sampled worlds, is then taken.

The strength of this technique tends to increase monotonically with the number of worlds sampled, albeit with diminishing returns for each additional world. As such, any technique that allows for significantly faster open-handed position solving will tend to result in stronger play. This is especially true in the early game, where positions take significantly longer to solve, and diminishing returns have not yet set in. Very fast solving techniques also open the door to expensive search-driven inference, such as asking: "having seen another player's action, in which worlds would I have made the same action?"

3.3 Payoff-Similarity in Skat

In this section we will refer to the value of Skat positions in terms of the number of card points achievable by the *max* player (the soloist) minus the card points achievable by the *min* players (the defenders), assuming perfect information

and that all players act optimally. Let $V(\cdot)$ be the corresponding state evaluation function.

Theorem 2. *Let s be a Skat position with card values $v_1, \ldots, v_k$. Let s' be the same position, except with card values $v'_i = v_i + \delta$ for some i, and $v'_j = v_j$ for $j \neq i$. Then $|V(s') - V(s)| \leq |\delta|$.*

Proof. Consider the game trees T and T' rooted in s and s'. Then T and T' are structurally equivalent, because cards have only changed in value, not rank. For each corresponding leaf pair (l_i, l'_i) in T and T', $V(l'_i) = V(l_i) \pm \delta$, where the $\pm$ depends on which team won the card with changed value. It follows that s and s' are payoff-similar and thus by Theorem 1, $|V(s) - V(s')| \leq |\delta|$. □

Corollary 3. *Let s be a Skat position with card values $v_1, \ldots, v_k$, and let s' be the same position with arbitrary card values $v'_1, \ldots, v'_k$. Then $|V(s) - V(s')| \leq \sum_{i=1}^{k} |v_i - v'_i|$.*

Proof. This follows immediately from Theorem 2 by applying it to each card value. □

We may use these results to construct a more generalized transposition table, indexed only by the canonical representation. That is, within each suit, we only care about which player owns the most powerful card, the next most powerful, etc., and not the values of those cards.

Like a standard transposition table, the entries store lower and upper bounds for the value of a given position. Unlike a standard transposition table we may have multiple entries for each index, with each entry annotated with the card values that were used to produce those bounds. A transposition table lookup then consists of converting the current state into a canonical form, computing the corresponding index, and then looping over all entries with that index.

This gives us a range $\langle \alpha^*, \beta^* \rangle$ within which the true value of the state provably lies, with $\alpha^* = \max_i(\alpha_i - \delta_i)$, and $\beta^* = \min_i(\beta_i + \delta_i)$, for each entry i. If this range is outside the current search window ($\beta^* \leq \alpha$ or $\beta \leq \alpha^*$) then we may immediately prune the state. Otherwise we can use it to tighten our α-β search window. If we ever encounter the exact same state (including card values) then $\delta_i = 0$ and the transposition table operates in the standard manner.

As an aside, if we can determine when the team that acquires a particular card i does not change, then we can add δ_i to $V(s)$ directly, rather than just increasing the δ bound. In Skat, this can be seen to occur for the highest remaining trump card, and all subsequent (in rank) uninterrupted trump owned by the same team.

3.4 Seeding the Transposition Table

Because there are relatively few canonical hand indices it becomes practical to precompute transposition table entries ahead of time, so that they may be used for all future searches. The effectiveness of this precomputation is ultimately determined by whether the reduction in search nodes is enough to offset the additional time required to query the extra transposition table entries. The reduction in search nodes is affected by two factors: the distribution of δ errors, which affects the tightness of the returned bounds; and the height within the search tree at which the entries occur.

Table 1: The number of canonical indices for suit and grand games is equal to "# of suit splits" × "ownership".

# of cards per hand	# of suit splits suit game	grand	ownership $\binom{3}{i,i,i}$
1	7	7	6
2	23	25	90
3	50	56	1,680
4	79	92	34,650
5	97	113	756,756
6	93	109	17,153,136
7	70	80	399,072,960
8	40	45	9,465,511,770
9	16	17	227,873,431,500
10	4	4	5,550,996,791,340

Thus, we would like to find an optimal balance between storing entries sufficiently high in the search tree, and having a sufficiently diverse collection of associated card values so that the returned bounds are non-trivial. In the interest of compactness we will store the precomputed entries using an implicit indexing scheme, such that the canonical index and the card value assignment do not need to be included in the table. This gives us less flexibility as to which entries and card values we can use to populate the table, but it also allows us to use only one byte per table entry (the actual DDS value).

Table 1 presents one possible implicit indexing scheme. The first component listed is the number of "suit splits" (after reducing by suit isomorphisms). Each split can be thought of as the number of cards remaining in each suit, without regard to which player holds those cards. The second component is the actual ownership assignment of those cards.

By inspecting Table 1, we can see that storing 5-card hands is the most we can practically achieve with current commodity systems. This corresponds to 0.72MB for each card valuation — we will want hundreds or thousands of entries (to cover the space of values), and we will need one table for each player to move (3) and game type (grand and suit game). We could construct larger tables on disk, but the increased access times would likely overshadow any decreases in node counts.

Optimizing TT Entries

Note that the maximum error, δ, of any canonical TT lookup is independent of the "ownership" index component (which is of the motivations for choosing this decomposition). Thus we need only select (and store) one set of card-value representatives for each of the possible suit splits. We may also note that δ is decomposable into the errors from each suit (i.e. $\delta = \delta_\diamondsuit + \delta_\heartsuit + \delta_\spadesuit + \delta_\clubsuit$). Thus, for any particular suit split, we can attempt to maximize the TT value coverage (minimizing the expected error) within each suit independently. The set of representatives for a given suit split is then the cross product of the representatives for each suit. Consider the possible card valuations for a non-trump suit of size 2:

$$\{\langle 0,0\rangle, \langle 0,3\rangle, \langle 0,4\rangle, \langle 0,10\rangle, \langle 0,11\rangle, \langle 3,4\rangle, \langle 3,10\rangle, \langle 3,11\rangle, \langle 4,10\rangle, \langle 4,11\rangle, \langle 10,11\rangle\}$$

If we had a budget of, say, 3 representatives from that suit then we might choose $\{\langle 0,3\rangle, \langle 4,10\rangle, \langle 10,11\rangle\}$. Note that there is no technical reason forcing us to select our representatives from the set of possible card values — we could even choose card values from the real numbers.

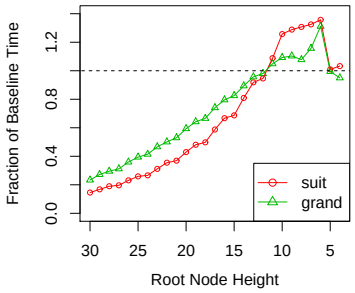

Figure 1: Skat DDS search times using payoff-similarity.

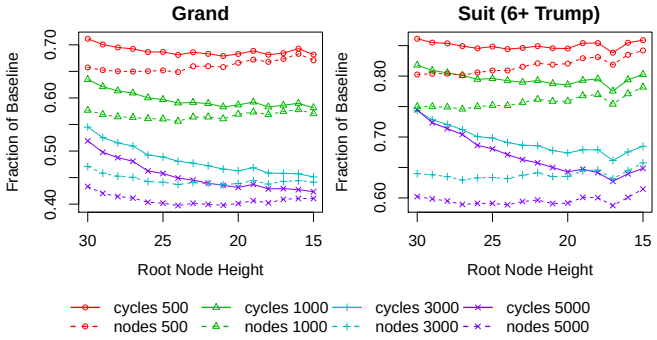

Figure 2: Reduction in nodes and CPU-cycles using precomputed transposition tables of various sizes.

Once we have found an optimal (or near optimal) set of representatives for each per-suit budget (k cards, n representatives), our task is to allocate our total budget amongst each suit split, so as to minimize the (possibly weighted) sum of expected δ values over all splits. E.g., for each a-b-c-d suit split, how many representatives should suit a have? Suit b? Etc. (Grand games have an additional suit e that represents Jacks, but these cards are all the same value.) This is accomplished via a relatively straightforward dynamic program, given a total memory budget. As a practical matter, the goal of this optimization is not entirely clear. One may attempt to optimize the worst-case δ error, optimize for the expected distribution of card values, or something else. We optimized against an unweighted distribution of all possible card values, as this produced the best results seen.

3.5 Experimental Results

To illustrate the effectiveness of payoff-similarity, we incorporated the described canonical indexing into a C++ MTD(f) solver that computes the exact card-point score of an open-handed position. This baseline solver is a high-performance program that uses hand-tuned move-ordering heuristics similar to those described in [Buro *et al.*, 2009], but generally cannot collapse card ranks into a canonical form due to the differing values of cards within a suit.

Figure 1 shows the results of adding payoff-similarity to the baseline solver, with each data point averaged over 10000 positions. Suit and grand games see a wallclock time reduction of 85% and 77% respectively, for positions at height 30, the hardest positions to solve. The payoff-similarity overhead does not outweigh the corresponding gains until the positions become essentially trivial (hands with 4 cards or fewer).

The payoff-similarity solver then became our new baseline player for examining the effectiveness of using precomputed TT entries. That is, the next results are over and above any gains from canonical indexing.

The results of using precomputed TT entries are shown in Figure 2. Budgets of 500, 1000, 3000, and 5000 valuations per table corresponds to memory requirements of 1.1, 2.2, 6.5, and 10.8 gigabytes per game type. Node and CPU-cycle reductions are relative to our exact solver with no precomputed TT entries, with all solvers exploiting payoff-similarity. The same 10000 positions were used within each game type.

Overhead from having to examine multiple table entries for each lookup means that the reduction in time does not match the reduction in nodes, and we can see a crossover for suit games, where the largest table is (slightly) slower than the second largest at height 30. Additionally there are diminishing returns in terms of reducing δ with larger tables, and the subsequent reduction in nodes expanded.

With the largest table sizes we achieve time reductions of 25% for suit games and 48% for grand games at a root height of 30. The greater improvement for Grand games is due to the monotonic ordering of card values within each suit — the trump suit for suit games includes Jacks, which are stronger than Aces but worth fewer points. This greater number of possible valuations means our representatives are further apart in the suit game tables.

4 Estimating DDS Results

Although computing a single DDS result is a relatively fast operation, there are a number of inference techniques which would benefit from being able to compute more results than are currently practical with a modest number of commodity processors. To this end, there exists a time-accuracy tradeoff, where a fast estimator may be used to replace either an entire DDS search, or, more generally, some subtree(s) within a search. If the estimated values are not too far from the exact values, then the value at the root node of the search must also necessarily be close to the true value. Since search is usually only needed to distinguish the relative quality of the possible moves, exact values are only necessary insofar as they can do so. Recall that for PIMC we need to compute the value of each successor state from the current position.

In our framework, value estimation can be viewed as a probabilistic analogue of bounded similarity. Instead of comparing our current state against a previously searched TT position, we have some function that tries to generalize across all positions (usually involving local features from a training set), and an associated empirical error distribution.

For highly tactical positions we expect that selecting moves using the values returned from DDS searches is in some sense the "right" thing to do. However this also assumes a level of coordination between players which is usually not the case in most situations. If the estimated search values more accurately reflect the real imperfect information values, then the move selection process could be both faster and stronger when using estimated or heuristic values.

4.1 Static Analysis

Before going into detail on our methods for estimating the DDS value of a position, we wish to first make note of methods for quickly bounding the *exact* DDS value. Given that teams cannot lose the points that they have already won, we are only trying to determine how the remaining card points are partitioned between the two teams.

We shall refer to any method which bounds the true DDS value and involves at most a small amount of search (preferably zero) as *static analysis*. Our best static analysis method involves the use of card- and trick-counting arguments at the start of a trick to produce upper and lower bounds on the number of points that the current player can make. For instance, if the declarer is to act and he were to "play from the top" in all safe suits, while the defenders threw off their least-valuable cards, how many points would the declarer make? Using static analysis within search decreases the amount of time required by about 13% (or 3% if a defender is to move at the root) for suit games, and 24% for grand games.

4.2 Linear Approximators

To estimate the result of a DDS search we trained several feature-based linear approximators, one for each combination of: game type (suit, grand), player to move (we assume that the soloist is in a fixed seat), and number of cards in hand. Because positions with small numbers of cards are already fast to solve, we only constructed estimators for start-of-trick positions where all players have 6, 7, 8, 9, or 10 cards in their hand. Mid-trick positions have more variables to consider and would require additional memory, disk space, and training.

Each approximator works in the standard manner: given a collection of feature functions $\Phi = \{\phi_1, \ldots, \phi_k\}$, the estimated value of a state s is $\sum_{i=1}^{k} \beta_i[\phi_i(s)]$, where each ϕ_i returns an index (effectively partitioning the states), and $\beta_i[]$ is an array of learned values.

The β values are trained using least-squares linear regression, with the training set being hundreds of millions of randomly generated states, labeled with exactly computed DDS values. To avoid overfitting, we generated at least 20 positions corresponding to each β_i entry. That is, we sampled 20 positions from each $\phi_i^{-1}(j)$. Due to overlap and different partition sizes, β entries may have more than 20 training points.

Features

The majority of the features used are so-called "constellation" features — given a list of cards (rank and suit), a constellation feature tracks the location of each card. Each card may be in one of 4 locations: the hand of player 1, 2, or 3, or *out*. Since we are only trying to predict the number of points yet to be made by the declarer, we do not need to keep track of which team won each of the *out* cards. For example, we use the following (plus some other) constellation features for suit games: Jacks + Aces; Aces + Tens; Top 2 Jacks + non-trump Aces and Tens; each non-trump suit; low cards (789s); trump between all pairs of players. Plus counting features such as: # of low cards each player has; # of trump each player has.

These features were chosen by hand, to fit within our memory budget and hopefully be reasonably predictive — we certainly do not claim that they are optimal.

Table 2: Linear regression estimator statistics — one estimator per: game type, number of tricks played, and player to move. Two measures of accuracy are listed: the average absolute error and the standard deviation (in parentheses).

		suit game			grand		
NT	P	LR	LR+SA	LR_{bdd}+SA	LR	LR+SA	LR_{bdd}+SA
0	S	2.2 (3.1)	2.0 (3.0)	1.8 (2.9)	2.8 (3.9)	2.3 (3.6)	1.9 (3.3)
0	D1	3.5 (4.7)	3.2 (4.5)	2.9 (4.3)	4.1 (5.4)	3.7 (5.1)	3.2 (4.8)
0	D2	3.6 (4.7)	3.2 (4.5)	2.9 (4.3)	4.1 (5.7)	3.7 (6.0)	3.2 (5.8)
1	S	2.4 (3.4)	2.1 (3.3)	1.9 (3.1)	2.7 (3.8)	2.2 (3.5)	1.7 (3.1)
1	D1	3.7 (4.9)	3.3 (4.7)	3.0 (4.4)	4.0 (5.3)	3.6 (5.0)	3.0 (4.6)
1	D2	3.7 (4.8)	3.3 (4.6)	2.9 (4.3)	4.0 (5.3)	3.5 (5.0)	3.0 (4.5)
2	S	2.5 (3.5)	2.1 (3.3)	1.9 (3.1)	2.8 (3.9)	2.3 (3.6)	1.8 (3.3)
2	D1	3.7 (4.9)	3.3 (4.7)	2.9 (4.3)	4.1 (5.5)	3.7 (5.2)	3.1 (4.7)
2	D2	3.6 (4.8)	3.2 (4.6)	2.8 (4.2)	4.1 (5.4)	3.6 (5.2)	3.0 (4.7)
3	S	2.6 (3.6)	2.2 (3.3)	1.8 (3.0)	2.9 (4.1)	2.3 (3.6)	1.8 (3.3)
3	D1	3.4 (4.5)	3.0 (4.3)	2.5 (3.9)	4.1 (5.5)	3.6 (5.3)	3.0 (4.7)
3	D2	3.3 (4.4)	2.9 (4.2)	2.5 (3.9)	4.1 (5.5)	3.6 (5.3)	3.0 (4.7)
4	S	2.5 (3.4)	2.0 (3.1)	1.6 (2.9)	3.0 (4.2)	2.3 (3.7)	1.7 (3.3)
4	D1	3.0 (4.0)	2.6 (3.8)	2.3 (3.6)	3.9 (5.3)	3.3 (5.0)	2.8 (4.6)
4	D2	2.9 (4.0)	2.6 (3.8)	2.2 (3.6)	3.9 (5.3)	3.4 (5.0)	2.8 (4.5)

Accuracy

Due to our aforementioned static analysis function we can (and do) check if the value returned by our linear estimator is wildly inaccurate. We can also cap the approximation error of each data point *during* the linear regression, instead of just using $\hat{y} - y$. Thus, we can help prevent the regression from "chasing" a data point where the estimate is very inaccurate.

Because our linear regression is performed via gradient descent, capping the error of a training point does not produce a plateau where we don't care about better approximating that point — rather, it caps the contribution of that data point to the slope of the regression. The regression will still attempt to correct for these errors, but it won't try as hard.

Table 2 shows the training error for several data sets, with and without using error capping during regression (LR_{bdd} and LR, respectively). We can see that using this method results in a significantly lower average absolute error and standard deviation. Although not shown, the errors for LR_{bdd} (without static analysis) are significantly greater than for LR.

4.3 Pruning

The ProbCut algorithm as described in [Buro, 1995] works by correlating the result of a shallow search with the value returned by a deeper search. That is, we can experimentally determine the expected value and variance of $v_{\text{shallow}} - v_{\text{deep}}$. Then, by performing a shallow search, we can form a window of arbitrary confidence within which we expect the value of a deep search to lie. If this window is outside the current α-β bounds then we can immediately prune, saving the effort of a deep search for the price of a small probability of error. If time permits, one may re-search a position with a wider window to increase confidence in the result.

This technique was applied successfully to Othello, where solving positions exactly is intractable. However in our regime the cost of exactly solving a position is much lower. So much so that sophisticated incremental search techniques have not received much attention, largely due to effective pruning and move ordering heuristics that reduce the effective average branching factor to less than 1.65. A result of

this low branching factor is that the overhead of incremental search techniques such as iterative deepening tends to outweigh any gains. Moreover, since we are using the PIMC algorithm there are many open-handed positions that we wish to solve, rather than solving one position with increasing confidence. That said, we may still select a ProbCut-style confidence threshold a priori, to use for searching many worlds.

Indeed, by taking the empirical standard deviation computed during the linear regression and combining it with our confidence threshold, we can form a window around the values returned by our estimators, within which we expect the true position value to lie. We do not use this window to tighten our the current α-β bounds (although we could), we only check if it is outside our cutoff bounds — if it is outside then we cut, otherwise we continue searching as normal.

4.4 Experimental Results

Our baseline DDS solver for examining our linear approximators is our payoff-similarity player with no precomputed TT entries. Table 3 lists the amount of time needed by the estimation player as a fraction of the baseline player, for various p_{cut} confidence thresholds. There is a clear trade-off in terms of search time vs. evaluation error at the root node.

Corresponding to the trend in Table 2, results for Grand games aren't as good as for suit games, presumably due to inadequate features. To explore the effect of approximation error on real card-play, we ran a tournament with two "Kermit" PIMC players from [Buro *et al.*, 2009], where we replaced the Java DDS solver with our C++ solver, with and without approximation.

We played 2000 close human-bid suit games with our approximation player using various p_{cut} values, with the results shown in Table 4. The number of worlds each player examined was a function of game stage, so that each player solved the same number of DDS positions. The same 2000 deals were used in all matches, but the DDS positions drawn by PIMC varied across matches.

Although this was an exploratory experiment and the tournament scores are close, the results seen match our intuition, with very aggressive pruning doing worse in general. Surprisingly it seems that an appropriate p_{cut} value may result in a PIMC player that is both stronger and faster. We speculate that this is due to the estimators undervaluing positions where tactical card-play requiring perfect knowledge is needed to achieve the optimal score.

Table 3: DDS search with linear estimation at height 30. (P: player to move; p_{cut}: probability of sound cut; r: correlation coefficient; σ: standard deviation of search error)

		suit game			grand		
P	p_{cut}	Avg. time fraction	r	σ	Avg. time fraction	r	σ
S	0.1	0.115	1.00	3.0	0.001	0.98	3.9
S	0.3	0.183	0.99	2.7	0.001	0.98	3.7
S	0.5	0.265	0.99	2.4	0.002	0.98	3.6
S	0.7	0.421	0.99	2.0	0.027	0.98	3.4
S	0.9	0.590	1.00	1.6	0.134	0.99	3.1
D1	0.1	0.124	0.98	3.4	0.001	0.96	5.3
D1	0.3	0.185	0.99	3.1	0.001	0.96	5.2
D1	0.5	0.257	0.99	2.8	0.002	0.97	5.1
D1	0.7	0.387	0.99	2.3	0.057	0.97	4.7
D1	0.9	0.547	0.99	1.9	0.194	0.98	4.0
D2	0.1	0.125	0.98	3.6	0.001	0.96	5.4
D2	0.3	0.191	0.99	3.2	0.001	0.96	5.4
D2	0.5	0.268	0.99	2.9	0.002	0.96	5.2
D2	0.7	0.403	0.99	2.5	0.066	0.97	4.8
D2	0.9	0.569	0.99	2.0	0.216	0.98	4.0

Table 4: Tournament results for the baseline player against an estimation player with various p_{cut} thresholds. The same 2000 initial deals (close human-bid suit games) were played twice with roles reversed. The number of worlds considered by both PIMC players was fixed, and increased with the trick number. The baseline player spent 15s on average per game.

p_{cut}	Average score diff. per game	Win %	Speedup
0.1	-0.67	49.6	2.21
0.2	-0.88	49.5	2.07
0.3	-0.79	49.5	1.94
0.4	+0.47	50.2	1.91
0.5	+0.80	50.4	1.71
0.6	+1.29	50.6	1.68
0.7	+1.01	50.6	1.50
0.8	+0.09	50.1	1.52
0.9	+0.84	50.4	1.39

5 Conclusions and Future Work

In this paper we have introduced the concept of payoff-similarity of game states which can be used to find exact bounds for minimax values of related states. Incorporating this technique into an already fast exact double dummy solver for Skat in conjunction with pre-computing transposition table values has shown speed-up factors of 8 and more in the beginning of the game. In a second step, we equipped our solver with a linear state value predictor to prune likely irrelevant parts of the search tree using a method similar to ProbCut. This addition increased the speed by another factor of 2 without losing playing strength, which is remarkable and worth future investigations into how forward-pruning for perfect information games performs in imperfect information games when using PIMC search. Moreover, with a total speed-up of 16+, state inference based on PIMC search is now becoming feasible in Skat. With this we expect the playing strength of Skat programs to increase substantially.

The concept of payoff-similarity may also be applicable to single-agent domains with changing operator costs, in which we could try to increase the value of search heuristics by querying values of related states.

Acknowledgments

Financial support was provided by NSERC.

References

[Buro *et al.*, 2009] M. Buro, J.R. Long, T. Furtak, and N. Sturtevant. Improving state evaluation, inference, and search in trick-based card games. In *Proceedings of IJCAI*, 2009.

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

[Ginsberg, 1996] Matthew L. Ginsberg. Partition search. In *Proceedings of AAAI-96*, pages 228–233. AAAI Press, 1996.