Leaf-Value Tables for Pruning Non-Zero-Sum Games

Nathan Sturtevant
University of Alberta Department of Computing Science
Edmonton, AB Canada T6G 2E8
nathanst@cs.ualberta.ca

Abstract

Algorithms for pruning game trees generally rely on a game being zero-sum, in the case of alpha-beta pruning, or constant-sum, in the case of multi-player pruning algorithms such as speculative pruning. While existing algorithms can prune non-zero-sum games, pruning is much less effective than in constant-sum games. We introduce the idea of leaf-value tables, which store an enumeration of the possible leaf values in a game tree. Using these tables we are can make perfect decisions about whether or not it is possible to prune a given node in a tree. Leaf-value tables also make it easier to incorporate monotonic heuristics for increased pruning. In the 3-player perfect-information variant of Spades we are able to reduce node expansions by two orders of magnitude over the previous best zero-sum and non-zero-sum pruning techniques.

1 Introduction

In two-player games, a substantial amount of work has gone into algorithms and techniques for increasing search depths. In fact, all but a small fraction of computer programs written to play two-player games at an expert level do so using the minimax algorithm and alpha-beta pruning. But, the pruning gains provided by alpha-beta rely on the fact that the game is zero-sum. Two-player games most commonly become non-zero-sum when opponent modeling is taken into consideration. Carmel and Markovitch [1996] describe one method for pruning a two-player non-constant-sum game.

A multi-player game is one with three or more players or teams of players. Work on effective pruning techniques in multi-player games began with shallow pruning [Korf, 1991], and continued most recently with speculative pruning, [Sturtevant, 2003]. While a game does not need to be constant-sum for pruning to be applied, the amount of pruning possible is greatly reduced if a game is not constant-sum.

Both two-player and multi-player pruning algorithms consist of at least two stages. They first collect bounds on players' scores, and secondly test to see if, given those bounds, it is provably correct to prune some branch of the game tree. This work focuses on the second part of this process. Alpha-beta and other pruning methods use very simple linear tests as a decision rule to determine whether or not they can prune. For zero-sum games these decisions are optimal. That is, they will make perfect decisions about whether pruning is possible. For non-zero-sum games, however, current techniques will not catch every possibility for pruning.

In order to prune optimally in a non-zero-sum game we must have some knowledge about the space of possible values that can occur within the game tree. Carmel and Markovitch assume a bound on the difference of player scores. We instead assume that we can enumerate the possible outcomes of the game. This is particularly easy in card games, where there are relatively few possible outcomes to each hand. Given that we can enumerate possible game outcomes, we can then make optimal pruning decisions in non-zero-sum games. In addition, these techniques can be enhanced by incorporating information from monotonic heuristics to further increase pruning.

In section 2 we illustrate the mechanics of a few pruning algorithms, before moving to a concrete example from the game of Spades in section 3. In section 4 we examine the computational complexity and the leaf-value table method that is used to implement different decision rules for pruning, followed by experimental results and conclusions.

2 Pruning Algorithms

To prune a node in a game tree, it must be proven that no value at that node can ever become the root value of the tree. We demonstrate how this decision is made in a few algorithms.

2.1 Alpha-beta

We assume that readers are familiar with the alpha-beta prun-

```
maxValue(state, $\alpha$, $\beta$)
    IF cutoffTest(state) return eval(state)
    FOR EACH s in successor(state)
        $\alpha \leftarrow \max(\alpha, \text{minValue}(s, \alpha, \beta))$
        IF $(\beta - \alpha \leq 0)^{\dagger}$ RETURN $\beta$
    RETURN $\alpha$
```

Figure 1: Alpha-beta pseudo-code.

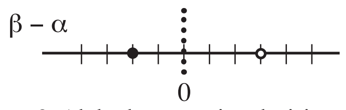

Figure 2: Alpha-beta pruning decision space.

ing algorithm. In Figure 1 we show pseudo-code for the maximizing player, modified slightly from [Russell and Norvig, 1995]. The statement marked † is where alpha-beta determines whether a prune is possible. This is illustrated in Figure 2. The *x*-axis is the value β – α, and we plot points as they are tested. If a point, such as the solid one, falls to the left of 0, we can prune, and if it falls to the right, like the hollow point, we can't. It is useful to understand this as a *linear classifier*, because it makes a single comparison with a linear function to determine if a prune is possible.

2.2 Max^n

The max^n algorithm [Luckhardt and Irani, 1986] is a generalization of minimax for any number of players. In a max^n tree with *n* players, the leaves of the tree are *n*-tuples, where the *i*th element in the tuple is the *i*th player's score or utility for that position. At the interior nodes in the tree, the max^n value of a node where player *i* is to move is the max^n value of the child of that node for which the *i*th component is maximum. At the leaves of a game tree an exact or heuristic evaluation function can be applied to calculate the *n*-tuples that are backed up in the game tree.

We demonstrate this in Figure 3. In this tree there are three players. The player to move is labeled inside each node. At node (a), Player 2 is to move. Player 2 can get a score of 3 by moving to the left, and a score of 1 by moving to the right. So, Player 2 will choose the left branch, and the max^n value of node (a) is (1, 3, 5). Player 2 acts similarly at node (b) selecting the right branch, and at node (c) breaks the tie to the left, selecting the left branch. At node (d), Player 1 chooses the move at node (c), because 6 is greater than the 1 or 3 available at nodes (a) and (b).

2.3 Max^n Pruning Algorithms

Given no information regarding the bounds on players' scores, generalized pruning is not possible. But, if we assume that each player's score has a lower bound of 0, and that there is an upper bound on the sum of all players scores, *maxsum*, we can prune. These bounds do not guarantee that a game is constant-sum, and existing pruning algorithms may miss pruning opportunities if a game is not constant-sum.

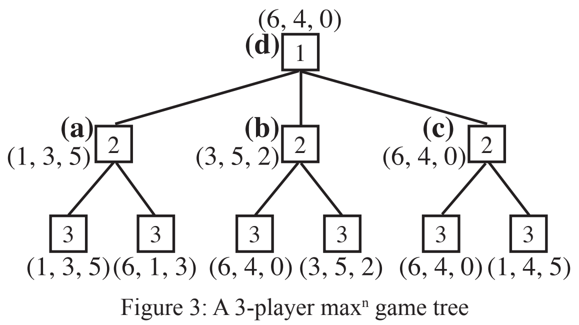

Figure 3: A 3-player max^n game tree

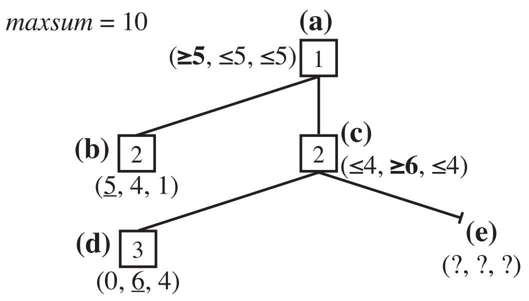

Figure 4: Shallow pruning in a 3-player max^n tree.

2.3.1 Shallow Pruning

Shallow pruning [Korf, 1991] is one of the simplest pruning algorithms for multi-player games. An example of shallow pruning is shown in Figure 4. The sum of players' scores in each max^n value is always 10, so *maxsum* is 10. In this tree fragment Player 1 is guaranteed at least a score of 5 at node (a) by moving towards (b). Similarly, Player 2 is guaranteed 6 points at (c) by moving towards (d). Regardless what the unseen value is at (e), Player 2 will not select that move if it gives him less than 6 points. Thus, Player 1 can never get more than *maxsum* - 6 = 4 points at (c). Player 1 at (a) must decide between a score of 5 at (b) and a score that is less than 4 at node (c). In this case, he will always select node (b), so we do not need to search node (e).

In this example we used consecutive bounds on Player 1 and Player 2's scores to prune. As stated previously, the general idea is to prove that unseen leaf nodes cannot become the max^n value of the game tree. In Figure 4, for instance, we can consider all possible values which might occur at node (b). If any of these can ever become the max^n value at the root of the tree, we cannot prune.

To formalize this pruning process slightly, we construct a *n*-tuple similar to a max^n value called the *bound vector*. This is a vector containing the lower bounds on players' scores in a game tree given the current search. While the max^n value is constrained to sum to *maxsum*, the bound vector can sum to as much as *n·maxsum*. In Figure 4, after exploring every node except node (b) our bound vector is (5, 6, 0). This is because Player 1 is guaranteed 5 points at the root, and Player 2 is guaranteed 6 points at node (c). Existing pruning algorithms prune whenever the sum of the components in the bound vector is at least as large as *maxsum*.

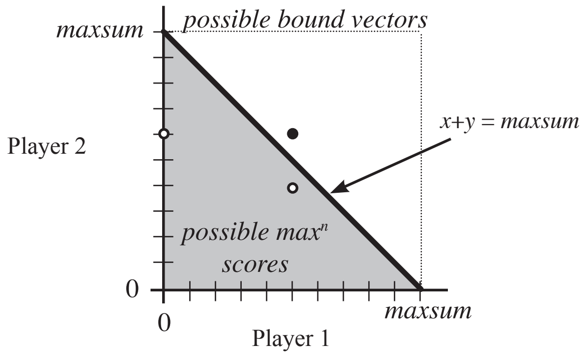

Figure 5: Shallow pruning decision space.

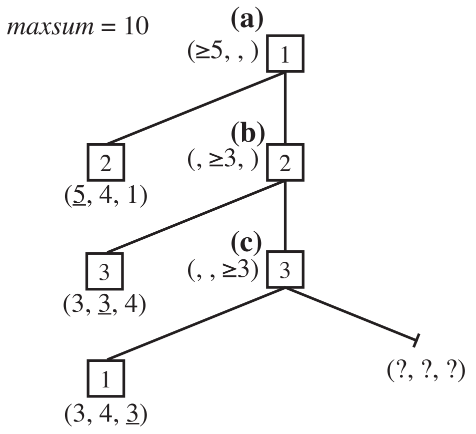

Figure 6: Speculative Pruning

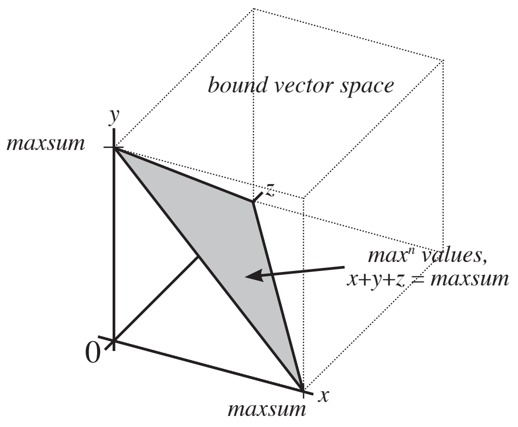

Figure 7: Speculative pruning decision space.

We show a visualization of the shallow pruning space in Figure 5. The x- and y-coordinates are scores or bounds for Player 1 and Player 2 respectively. The shaded area is the space where possible max^n values for Player 1 and 2 will fall. All max^n values in the game must be in this space, because their sum is bounded by *maxsum*. Because shallow pruning ignores Player 3, Player 1 and Player 2's combined scores are not constant-sum, so they can fall anywhere to the left of the diagonal line. Bound vectors, however, can fall anywhere in the larger square. If a bound vector is on or above the diagonal line defined by $x+y = maxsum$, we can prune, because there cannot be a max^n value better than those bound vectors. Like alpha-beta, shallow pruning is using a linear classifier to decide when to prune.

Ignoring Player 3's values, we plot the leaf values (5, 4, -) and (0, 6, -) from Figure 4 as open points, and the bound vector (5, 6, -) used to prune in Figure 4 as a solid point. In this instance, there is no gap between the gray region and the diagonal line, so the line defined by $x+y = maxsum$ is a perfect classifier to determine whether we can prune.

2.3.2 Alpha-Beta Branch-and-Bound Pruning

Although shallow pruning was developed for multi-player games, the basic technique only compares the scores from two players at a time. Alpha-beta branch-and-bound pruning [Sturtevant and Korf, 2000] is similar to shallow pruning, except that it uses a monotonic heuristic to provide bounds for all players in the game. The bound vector in Figure 4 was (5, 6, 0). Player 3's bound was 0, because we had no information about his scores. But, supposing we had a monotonic heuristic that guaranteed Player 3 a score of at least 2 points. Then the bound vector would be (5, 6, 2). This additional bound makes it easier to prune, since we can still prune as soon as the values in the bound vector sum to *maxsum*.

2.3.3 Speculative Pruning

Speculative pruning [Sturtevant, 2003], like alpha-beta branch-and-bound pruning, takes into account all of the players in the game. It does this by considering multiple levels in the game tree at one time. We demonstrate this in Figure 6. In this figure, the important bounds are Player 1's bound at (a), Player 2's bound at (b) and Player 3's bound at (c). These together form the bound vector (5, 3, 3). If these values sum to at least *maxsum* we can guarantee that there will never be a value at the right child of (c) which can become the max^n value of the tree. When pruning over more than 2-ply in multi-player games there is the potential that while a value at (c) cannot become the max^n value of the tree, it can affect the max^n value of the tree. Other details of the speculative pruning algorithm prevent that from happening, but we are only concerned with the initial pruning decision here.

We illustrate the pruning decision rule for speculative pruning in Figure 7. In this case, because we are comparing three player's scores, the decision for whether we can prune depends on a 2-d plane in 3-d space, where each axis corresponds to the score bound for each of the 3 players in the game. Thus each max^n value and bound vector can be represented as a point in 3-d space. For a three-player constant-sum game, all possible max^n values must fall exactly onto the plane defined by $x+y+z = maxsum$, which is also perfect classifier for determining when we can prune. As in shallow pruning, bound vectors can fall anywhere in the 3-d cube.

2.4 Generalized Pruning Decisions

In all the pruning algorithms discussed so far, a linear discriminator is used to prune. This works well when a game is zero-sum or constant-sum, but in non-constant-sum games a linear classifier is inadequate[1].

When a game is non-constant-sum, the boundary of the space of max^n values is not defined by a straight line. We demonstrate this in the next section using examples from the game of Spades. To be able to prune optimally, we need to know the exact boundary of feasible max^n values, so that we can always prune when given a bound vector outside this region. We explain methods to do this in Section 4.

3 Sample Domain: Spades

Spades is a card game for 2 or more players. In the 4-player version players play in teams, while in the 3-player version each player is on their own, which is what we focus on. There are many games similar to Spades which have similar proper-

[1]Further details on the relationship between constant-sum and non-constant-sum games are found in Appendix A, but they are not necessary for understanding the contributions of this paper.

# Tricks Taken	Utility/Evaluation (bid)			Ranked max^n vals
	P1 (1)	P2 (1)	P3 (2)	
(0, 0, 3)	0	0	19+6	(0, 0, 2)
(0, 1, 2)†	0	10+3	20+3	(0, 2, 1)
(0, 2, 1)†	0	9+6	0	(0, 4, 0)
(0, 3, 0)†	0	8+6	0	(0, 3, 0)
(1, 0, 2)	10+3	0	20+3	(2, 0, 1)
(1, 1, 1)†	10+3	10+3	0	(2, 2, 0)
(1, 2, 0)†	10+3	9+3	0	(2, 1, 0)
(2, 0, 1)	9+6	0	0	(4, 0, 0)
(2, 1, 0)†	9+3	10+3	0	(1, 2, 0)
(3, 0, 0)	8+6	0	0	(3, 0, 0)

Table 1: Outcomes for a 3-player, 3-trick game of Spades.

ties. We will only cover a subset of the rules here. A game of Spades is split into many hands. Within each hand the basic unit of play is a trick. At the beginning of a hand, players must bid how many tricks they think they can take. At the end of each hand they receive a score based on how many tricks they actually took. The goal of the game is to be the first player to reach a pre-determined score, usually 300 points.

If a player makes their bid exactly, they receive a score of 10×*bid*. Any tricks taken over your bid are called overtricks. If a player takes any overtricks they count for 1 point each, but each time you accumulate 10 overtricks, you lose 100 points. Finally, if you miss your bid, you lose 10×*bid*. So, if a player bids 3 and takes 3, they get 30 points. If they bid 3 and take 5, they get 32 points. If they bid 3 and take 2, they get -30 points. Thus, the goal of the game is to make your bid without taking too many overtricks.

If all players just try to maximize the number of tricks they take, the game is constant-sum, since every trick is taken by exactly one player, and the number of tricks available is constant. But, maximizing the tricks we take in each hand will not necessarily maximize our chances of winning the game, which is what we are interested in. Instead, we should try to avoid overtricks, or employ other strategies depending on the current situation in the game.

In a game with 3 players and t tricks, there are $(t+1)(t+2)/2$ possible ways that the tricks can be taken. We demonstrate this in Table 1, for 3 players and 3 tricks.

Table 1 is an example of a *leaf-value table*. It contains all possible leaf values and their associated utility in the game. In Spades, we build such a table after each player had made their bid, but before game play begins. The first column enumerates all possible ways that the tricks can be taken by each player. The second through fourth columns evaluate the score for each player, that is their utility for each particular outcome. In this example, Player 1 and Player 2 have bid 1 trick each, and Player 3 bid 2 tricks. If a player does not make their bid they have a score of 0, otherwise we use a heuristic estimate for their score, 10×*bid* - *overtricks* + 3×(*how many opponents miss their bid*). As has been shown for minimax [Russell and Norvig, 1995], we only care about the relative value of each state, not the absolute value. So, in the last column we have replaced each player's utility with a rank, and combined all player's ranks into the final max^n value.

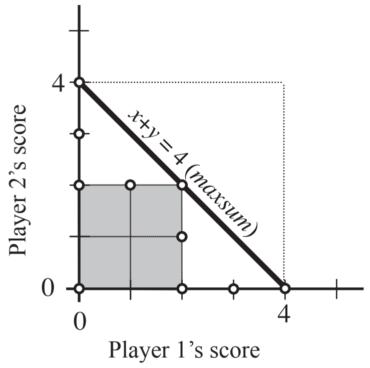

Figure 8: Pruning decision space for Table 1.

As an example, the first possible outcome is that Players 1 and 2 take no tricks, while Player 3 takes 3 tricks. Since Players 1 and 2 both missed their bids, they get 0 points. Player 3 made his bid of 2 tricks, and took 1 overtrick. Since we want to avoid too many overtricks, we evaluate this as 20-1 = 19 points. But, since both Player 1 and 2 miss their bids in this scenario, Player 3 has a bonus of 6 points. This is the best possible outcome for Player 3, so it gets his highest ranking. For Player 1 and 2 it is their worst possible outcome, so they rank it as 0.

We graph the shallow pruning decision space for the first two players of this game in Figure 8. In this game *maxsum* is 4. The 10 possible leaf values for the first two players are all plotted as hollow points in the graph. If we use *maxsum* as a discriminator to decide if we can prune, it will indicate that we can only prune if a bound vector falls on or above the bold diagonal line. But, we can actually prune as long a bound vector is on or above the border of the gray region. So, we can actually prune given any bound vector for player 1 and 2 except (0, 0), (0, 1), (1, 0) and (1, 1).

As a final point, we note that in card games like Spades the number of tricks you take increases monotonically, which can be used as a heuristic to help pruning. This observation is the key behind the alpha-beta branch-and-bound pruning technique. If we are using a utility function like in Table 1 it is very complicated to try to use the monotonic heuristic to help prune. But, because we have a leaf-value table, the task is much easier.

Assume, for instance, that Player 2 has already taken 1 trick. Then, we can ignore all outcomes in which he does not take one trick. This gives us a reduced set of values, which are marked with a † in Table 1. We will discuss to use this in the next section.

4 Leaf-Value Tables

The formal definition of a leaf-value table is a table which holds all possible outcomes that could occur at a leaf node in a game. Looking back to the pruning algorithms in Section 2, we want to replace the linear classifiers with more accurate classifiers, given a leaf-value table. Thus, we need an effi-

```
1   canLeafValueTablePrune(bounds[], heuristicUB[])
2       IF (inHashTable(bounds, heuristicUB))
3           return hashLookUp(bounds, heuristicUB)
4       FOR each outcome in leaf-value table
5           FOR i in players 1..n
6               if outcome[i] > heuristicUB[i]
7                   skip to next outcome;
8           FOR i in players 1..n
9               if outcome[i] ≤ bounds[i]
10                  skip to next outcome;
11          addToHashTable(false, bounds, heuristicUB)
12          RETURN false;
13      addToHashTable(true, bounds, heuristicUB)
14      RETURN true;
```

Figure 9: Leaf-Value Table Pruning pseudo-code.

cient way to both find and compute regions like in Figure 8 to determine when we can prune.

Theorem: The information stored in a leaf-value table is sufficient to make optimal pruning decisions.

Proof: We first assume that we have a bound vector $b = (b_1, b_2, \ldots, b_n)$, where each bound in the vector originates on the path from the root of the tree to the current node. We are guaranteed that a player i with bound b_i will not change his move unless he can get some value v_i where $v_i > b_i$. Thus, if there exists a value $v = (v_1, v_2, \ldots, v_n)$ where $\forall_i\ v_i > b_i$ then we cannot prune, because v is better than every bound in the search so far. Given a leaf-value table, we know every possible value in the game, and so we can explicitly check whether some v exists that meets the above conditions, and thus we can use a leaf-value table to make optimal pruning decisions. □

Because a leaf-value table gives us an exact description of the boundary of the spaces where we can and cannot prune, the only question is how we can use this information efficiently. Given a bound vector, we need to quickly determine whether or not we can prune, because we expect to ask this question once for every node in the game tree.

For small games we can build a lookup table, enumerating all possible bound vectors and whether or not we can prune, which will provide constant time lookup. But, there are many situations where we have a dynamic evaluation function or where the game is too large to build an entire database. Instead, we can simply use a hash table to store portions of the table as we calculate them dynamically.

Simple pseudo-code for leaf-value pruning is in Figure 9. This procedure is called by a pruning algorithm to decide whether or not to prune. We pass in the current bound vector, as well as heuristic upper bounds for any players at the current node in the game. If an entry in the leaf-value table is inconsistent with the heuristic (line 6), we can ignore that entry, because that outcome cannot occur below the current node in the tree. For the remaining outcomes, if any player's score for that outcome that is no better than the his bound (line 9), we can still prune. If every players' score is greater than their respective value of the bound vector, we reach lines 11-12, indicating we cannot prune.

So, returning to our example in Spades, instead of searching in Table 1 with 10 entries, knowing that Player 2 has taken 1 trick reduces the leaf-value table to the 6 entries in Table 1 marked with a †. Reducing the number of entries in this table make it more likely that we can prune.

Every time we attempt to prune, we will have to pay O(*table size*), but this cost is quickly amortized over the look-ups, and doesn't add significant overhead. In a game with more outcomes there are a variety of methods that can be used to reduce the lookup cost. Additionally, we can use the standard *maxsum* check, as it will always be correct if it indicates we can prune.

5 Experimental Results

5.1 Nodes Expanded in Three-Player Spades

We use the game of Spades as a test-bed for the techniques introduced in this paper. Our first experiment will illustrate that leaf-value tables are quite effective for pruning, and will also help illustrate when they will be most effective. To do this, we compare the number of nodes expanded using (1) previous pruning techniques and (2) a variety of evaluation functions in the 3-player version of Spades. One of these evaluation functions is actually from a different bidding game, called "Oh Hell!"

For each method, we counted the total number of nodes expanded to compute first move from 100 hands of Spades, where each player started with 9 cards, and searched the entire game tree (27-ply). Our search used a transposition table, but no other seach enhancements besides the pruning methods explicitly discussed. For these trees, the leaf-value tables contain 55 entries. We present the results in Table 2.

The first column in the table is the average size of the game trees without pruning, 2.7 million nodes. This is determined by the cards players hold, not the evaluation function used, so we will compare the other tree sizes to this value.

The next two columns are the results using speculative pruning, the previous best pruning technique. If we use a non-constant-sum (*NZS*) evaluation function, speculative pruning is able to reduce the average tree size by a factor of 1.36 to 1.9 million nodes. Maximizing tricks, *MT*, is the best-case evaluation function for speculative pruning, because it is constant-sum. In this case the average tree size is reduced to 1.1 million nodes.

	Full Tree	*NZS*	*MT*	*MoMB*	*mOT*	*smOT*	*WL*	*OH*
Depth 27	2.7 M	1.9M	1.1M	400k	37.2k	8.6k	788	40.7k
reduction	-	1.36×	2.37×	6.6×	71×	308×	3362×	65×

Table 2: Overall reduction and average tree sizes in Spades.

	Full Tree	Best NZS	Zero-Sum	LVT
2-Player	481k	225k	14k	22k
reduction	-	2.14×	34.4×	21.9×

Table 3: Overall reduction and average tree sizes in Spades.

	Avg. Points	% Wins
LVT	263	62.3%
prev. methods	226	37.7%

Table 4: Average score over 100 games of Spades.

Now, we consider non-zero-sum evaluation functions. The first function we use is *MoMB*, maximizing the number of opponents that miss their bid. This is quite similar to the strategy of maximizing your tricks, but the average tree size can be reduced further to 400k nodes, a 6.6 fold reduction.

The next evaluation function we use tries to minimize overtricks, *mOT*. This produces much smaller trees, 37.2k nodes on average, a 71 fold reduction over the full tree. A similar evaluation function, *smOT*, gives a slight margin for taking overtricks, because that is how we keep our opponent from making their bid, but tries to avoid too many overtricks. This evaluation function reduces the tree further to 8.6k nodes, over 300 times smaller than the original tree. The *smOT* evaluation is what was used with speculative max^n for the *NZS* experiment. In the end, both algorithms calculate the exact same strategies, but with leaf-value tables we can do it hundreds of times faster.

Finally, we show the simplest evaluation function possible, *WL*. This function gives a score of 1 if we make our bid and 0 if we miss it. In practice we wouldn't want to use this evaluation function because it is much too simple, but it does give a rough estimate on the minimum tree size. Trees searching using this method averaged 788 nodes.

"Oh Hell!" (*OH*) is a similar game to Spades, however the goal of this game is to get as close to your bid as possible. Using this evaluation function, the average tree size was 40.7k nodes, 65 times smaller than the full tree.

Besides showing the effectiveness of leaf-value tables, these experiments help illustrate two reasons for additional pruning from leaf-tables in a non-constant-sum game. The first reason for large reductions is that the non-constant-sum evaluation may significantly reduce the space of possible outcomes in the game. The best example of this is the *WL* evaluation function. But, *smOT* also reduces the possible outcomes over *mOT*, and thus reduces the size of the game trees.

The other factor that is important for pruning is having a monotonic heuristic and an evaluation function that is not monotonic in relationship to the monotonic heuristic. Evaluation functions like *mOT* are non-monotonic, because we intially want to take more tricks, but, after making our bid, we then don't want to take any more tricks. This allows a monotonic heuristic to more tightly constrain the search space, and thus increases pruning.

The *MoMB* evaluation function is monotonic, and only a slight simplification of the *MT* evaluation function, so we see the least gains with this evaluation function.

5.2 Nodes Expanded in Two-Player Spades

We conducted similar experiments in the two-player game of Spades. Two-player non-zero-sum games in the form we have described here are vulnerable to the same deep pruning problems as found in multi-player games. See [Korf, 1991] for a detailed explanation of the multi-player problem. It is not difficult to show the same problem exists in two-player non-zero-sum games. The bottom line is that we cannot prune as efficiently as alpha-beta once we use a non-zero-sum evalution function. In these experiments we did not apply every conceivable game-tree reduction technique, only transposition tables and basic alpha-beta pruning, so in practice we may be able to generate smaller two-player zero-sum game trees.

In the two-player Spades games, we searched 100 hands to depth 26 (13 tricks) using a variety of techniques. The results are in Table 3. The full tree averaged 481k nodes. Using a non-zero-sum evaluation function and the best previous methods produced trees that averaged 225k nodes. Using alpha-beta pruning and a zero-sum evaluation function reduced the trees further to 14k nodes on average. Using leaf-value tables for pruning produced trees were slightly larger, 22k nodes. As referenced above, these trees are larger than those generated by alpha-beta because we cannot apply deep pruning, but they are still much smaller than previously possible given a non-zero-sum evaluation function.

5.3 Quality of Play From Extended Search

Finally, to compare the effect of additional search on quality of play, we then played 100 games of 3-player Spades, where multiple hands were played and the scores were accumulated until one player reached 300 points. Also, each complete game was replayed six times, once for each possible arrangement of player types and orderings, excluding games with all of one type of player. Each player was allowed to expand 2.5 million nodes per turn. One player used leaf-value tables to prune, while the other used previous methods. Hands were played "open" so that players could see each others cards. The results are in Table 4. The player using the LVT was able to win 62.3% of the games and averaged 263 points per game, while the player using previous techniques averaged only 226 points per game.

5.4 Summary

We have presented results showing that leaf-value tables, when combined with previous pruning techniques, can effectively prune non-constant-sum games such as Spades or Oh Hell! In these games, even with four players, the largest leaf-value tables will only have 560 entries, so the cost is relatively small.

Although we do not present experimental results here, we can predict the general nature of results in another game such as Hearts. When played with 4-players, Hearts will have leaf-value tables with at most 2,240 entries. In most situations in Hearts, our evaluation function will be constant-sum. But, there will be some situations where a non-constant-sum evaluation function is needed. Thus, if we use leaf-value tables for such a game, we will have the same gains as previous techniques in portions of the game that are constant-sum. But, when the game is non-constant sum, we will benefit from additional pruning, although the exact amount will depend on the particular situation.

6 Conclusions and Future Work

In this paper we have shown how an enumeration of possible leaf-values in a game tree, called a leaf-value table, can be used to change the linear classifier used in classical pruning algorithms to an arbitrary classifier needed for non-zero-sum games. This technique works particularly well in card games like Spades, where we see up to a 100 fold reduction of nodes expanded over the previous best results using a constant-sum evaluation function, along with gains in quality of play.

This work expands the limits of how efficiently we can search two-player and multi-player games. To a certain extent there is still an open question of how to best use limited resources for play. Recent results in applying opponent modeling to two-player games have not been wildly successful [Donkers, 2003]. In multi-player games, others have shown that deep search isn't always useful, if there isn't a good opponent model available [Sturtevant, 2004]. But, given a reasonable opponent model, this work allows us to use the best evaluation function possible and still search to reasonable depths in the game tree.

In the future we will continue address the broader question of what sort of opponent models are useful, and what assumptions we can make about our opponents without adversely affecting the performance of our play. The ultimate goal is to describe in exactly which situations we should use max^n, and in which situations we should be using other methods. These are broad questions which we cannot fully answer here, but we these additional techniques will provide the tools to better answer these questions.

Acknowledgements

This research benefited from discussions with Jonathan Schaeffer, Martin Müller, Michael Bowling, Akihiro Kishimoto and Markus Enzenberger. The support of Alberta's Informatics Circle of Research Excellence (iCORE) is also appreciated.

A Games Transformations

In this paper we have often made distinctions between games or evaluation functions based on whether they are constant-sum or not. These distinctions can be blurred through minor transformations, however such transformations do not change the underlying nature of the game. In this appendix we explain this in more detail, but the details are not necessary for understanding this paper.

First, we can take a game or evaluation function that is naturally constant-sum and make it non-constant-sum by either adding a player which doesn't actually play in the game, but receives a random score, or by applying an affine transform to one or more players' scores. Neither of these transformations, however, will change the *strategies* calculated by max^n, because max^n makes decisions based on the relative ordering of outcomes, which an affine transform preserves, and because any extra players added will not actually make decisions in the game tree.

If we are unaware of either of these changes, previous pruning algorithms may treat the game as non-constant-sum and miss pruning opportunities. But, leaf-value tables are not affected by either of these changes. Leaf-value tables compute the ranking of all outcomes, so any affine transform applied to an evaluation function will be removed by this process. Because no bounds will ever be collected for any extra players in the game, as they are not actually a part of the game, pruning decisions are unchanged.

Secondly, we can take a game that is non-constant-sum and make it constant-sum by adding an extra player whose score is computed such that it makes the game constant-sum. Again, because this extra player never actually plays in the game, no pruning algorithm will ever collect bounds for this player, and it is equivalent to playing the original game. No extra pruning can ever be derived from such a change.

Adding an additional player may also make a non-linear classifier appear to be linear. But, because the extra player never plays, we will actually need to use the non-linear classifier to make pruning decisions.

Thus, while the difference between constant-sum games and non-constant-sum games can be blurred by simple transforms, the underlying game properties remain unchanged.

References

[Carmel and Markovitch, 1996] Carmel, D. and Markovitch, S., Incorporating Opponent Models into Adversary Search, AAAI-96, Portland, OR (1996)

[Donkers, 2003] Donkers, H.H.L.M., Nosce Hostem: Searching with Opponent Models, PhD Thesis, 2003, University of Maastricht.

[Korf, 1991] Korf, R., Multiplayer Alpha-Beta Pruning. Artificial Intelligence, vol. 48 no. 1, 1991, 99-111.

[Luckhardt and Irani, 1986] Luckhardt, C.A., and Irani, K.B., An algorithmic solution of N-person games, Proceedings AAAI-86, Philadelphia, PA, 158-162.

[Russell and Norvig, 1995] Russell, S., Norvig, P., Artificial Intelligence: A Modern Approach, Prentice-Hall Inc., 1995.

[Sturtevant, 2004] Sturtevant, N.R., Current Challenges in Multi-Player Game Search, Proceedings, Computer and Games 2004, Bar-Ilan, Israel.

[Sturtevant, 2003] Sturtevant, N.R., Last-Branch and Speculative Pruning Algorithms for Max^n, Proceedings IJCAI-03, Acapulco, Mexico.

[Sturtevant and Korf, 2000] Sturtevant, N.R., and Korf, R.E., On Pruning Techniques for Multi-Player Games, Proceedings AAAI-2000, Austin, TX, 201-207.