*-Minimax Performance in Backgammon

Thomas Hauk, Michael Buro, and Jonathan Schaeffer

Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2E8
{hauk,mburo,jonathan}cs.ualberta.ca

Abstract. This paper presents the first performance results for Ballard's *-Minimax algorithms applied to a real–world domain: backgammon. It is shown that with effective move ordering and probing the Star2 algorithm considerably outperforms Expectimax. Star2 allows strong backgammon programs to conduct depth 5 full-width searches (up from 3) under tournament conditions on regular hardware without using risky forward pruning techniques. We also present empirical evidence that with today's sophisticated evaluation functions good checker play in backgammon does not require deep searches.

1 Introduction

*-Minimax is a generalization of Alpha-Beta search for minimax trees with chance nodes [4][6][7]. Like Alpha–Beta search, *-Minimax can safely prune subtrees which provably do not influence the move decision at the root node. Although introduced by Ballard as early as 1983, *-Minimax has not received much attention in the AI research community. In fact, it never found its way into strong backgammon programs. This is surprising in view of the importance of searching deeper in deterministic two–player perfect information games like chess, checkers, and Othello. Curious about the reasons for this we set out to investigate. Here we report the results we obtained.

The paper is organized as follows: we first briefly discuss search algorithms for trees with chance nodes. Then we survey the development of strong backgammon programs and describe the GNU backgammon program in some detail because we used its evaluation function in our *-Minimax experiments. Thereafter, we discuss implementation issues and our experimental setup, and present empirical results on the search performance and playing strength obtained by *-Minimax in backgammon. We conclude the paper by suggesting future research directions.

2 Heuristic Search in Trees with Chance Nodes

The baseline algorithm for trees with chance nodes analogous to Minimax search is the Expectimax algorithm [9]. Just like Minimax, Expectimax is a full-width search algorithm. It behaves exactly like Minimax except it adds an extra component for dealing with chance nodes (in addition to Min or Max nodes): at chance nodes, the heuristic value (or Expectimax value) is equal to the sum of the heuristic values of its successors weighted by their individual probabilities.

Just as cutoffs in Minimax search can be obtained by computing value bounds which are passed down to successor nodes (the Alpha-Beta algorithm), so too can we derive a strategy for pruning subtrees in Expectimax based on value windows. Cutoffs at Min and Max nodes are found in the same way as in Alpha-Beta search, i.e. whenever a value of a Max node successor is not smaller than the upper bound β passed down from the parent node, no further successors have to be considered. Similarly, cutoffs can be obtained at Min nodes by comparing successor values with the lower bound α. Ballard discovered that also at chance nodes cutoffs are possible [4]: if we know lower and upper bounds on the values leaf nodes can take (called L and U respectively), we can determine bounds on the value of a chance node based on the passed down search window (α, β), the values of successors we already determined $(V_1, \ldots, V_{i-1})$, the current successor value V_i, and pessimistic/optimistic assumptions about the successor values yet to be determined $(V_{i+1}, \ldots, V_n)$. Assuming probability P_j for event j we obtain the following sufficient conditions for the node value V to fall outside the current search window (α, β):

$$V_i \leq (\alpha - \sum_{j=1}^{i-1} P_j \cdot V_j - U \cdot \sum_{j=i+1}^{n} P_j)/P_i \quad \Rightarrow \quad V \leq \alpha$$

$$V_i \geq (\beta - \sum_{j=1}^{i-1} P_j \cdot V_j - L \cdot \sum_{j=i+1}^{n} P_j)/P_i \quad \Rightarrow \quad V \geq \beta$$

These value bound computations are at the core of Ballard's Star1 algorithm which prunes move i when the resulting window for V_i is empty.

Ballard's family of Star2 algorithms improves upon Star1 by exploiting the regularity of trees in which successors of chance nodes have the same type (either Min or Max). This condition, however, is not a severe constraint because any tree with chance nodes can be transformed into the required form above by merging nodes and introducing artificial single action nodes. To illustrate the idea behind Star2, we consider the case where a chance node is followed by Max nodes. In order to establish non-trivial lower bounds on the chance node value it is sufficient to *probe* just a subset of moves at the following Max nodes. I.e.

$$\sum_{j=1}^{n} P_j V_j \geq \sum_{j=1}^{m} P_j V_j^*,$$

where $m \leq n$ and V_j^* is obtained by maximizing values over a subset of moves available at successor j. A straight–forward and simple probing strategy — which we use in our backgammon experiments — is to consider only single good moves at each successor node and continuing the search like Star1 if the probing phase does not produce a cutoff. Note that probing itself calls the Star2 routine recursively.

For a more detailed description and analysis the reader is referred to Ballard's original work [4] and our companion paper [7] which also provides pseudo-code.

3 Backgammon Programs: Past and Present

With a large state space (estimated to be bigger than 10^{20} states [12]) and an imposing branching factor (there are 21 unique dice rolls, and about 20 moves per roll, on average), it is not surprising that most of the early computer backgammon programs were *knowledge-based*. Knowledge-based systems do not rely much on search, but rather attempt to choose moves based on *knowledge* about the domain, usually programmed into the system by a human expert.

The first real success in computer backgammon was BKG, developed by Hans Berliner. In 1979, BKG played the world champion at the time, Luigi Villa, and managed to defeat him 7-1 in a five point match [5]. While many people were shocked, even Berliner himself would concede weeks after the match that BKG had been lucky with rolls and made several technical blunders. However, Villa had not been able to capitalize on those mistakes – such is the life with dice.

The second milestone in computer backgammon was Neurogammon [11], the work of IBM researcher Gerald Tesauro. Neurogammon used an *artificial neural network* for evaluating backgammon positions. Neurogammon was trained with *supervised learning*; it was fed examples labeled by a human expert, and told what the answer should be. The program quickly became the best in computer backgammon, but still only played at the level of a strong human amateur player.

Tesauro went back to the drawing board. One of the first things he changed was the data the program was training on. Instead of using hand-labeled positions, he decided he would rely solely on *self-play* to generate training data – the program would simply play against itself. This has advantages over the previous method since a human expert may label positions incorrectly, or tire quickly (Neurogammon only used selected positions from about 3000 games [11] to train checker play, culled from games where Tesauro had played both sides), but self-play also may lead a program into a *local* area of play. For example, a program can learn how to play well against itself, but not against another opponent. This local minima problem in backgammon is partially overcome due to the fact that the environment is stochastic – dice insert a certain level of randomness – so a program is forced to explore different areas of the state space.

The other thing Tesauro changed was the training method itself. Instead of using a supervised learning approach that adjusted the network after each move (which he could do before because each training example was labeled), Tesauro decided on adapting *temporal-difference learning* for use with his neural network [10][12]. TD learning is based on the idea that an evaluation for a state should depend on the state that follows it. In a game sense, the computer keeps track of each position from start to finish, and then works backward. It trains itself on the last position, with the target score being the outcome of the game. Then it trains itself on the second last position, trying to more accurately predict the score it got for the last position (*not* the final score). The last position is the only position which is given a *reward signal*, or absolute value; all other positions are only trained to better predict the position that followed it. In games, the reward signal is related to the outcome of the game. If the program had lost, the reward signal would be low (to act sort of like a punishment). If the program won, the reward signal would be high. Since backgammon cannot end in a draw, the reward signal could never be zero.

In this manner, Tesauro delayed the final reward signal for the neural network until the game was won or lost, at which point the network would begin adjusting itself. This new program was called TD-Gammon in honour of its training method. Tesauro trained the first version of TD-Gammon against itself for 300,000 games, at which point the program was able to play as well as Neurogammon – quite surprising, considering the program had essentially "discovered" good play on its own, with no human intervention, and zero explicit knowledge. Later versions of TD-Gammon increased the size of the *hidden units* in the network, added hand-crafted *features* to the input representation, trained for longer amounts of time, and included a selective search algorithm to extend the search process deeper than a single ply. TD-Gammon is considered to safely be in the top-3 players of the world. One human expert even ventured to say it was probably better than any human, since it does not suffer from mental exhaustion or emotional play.

TD-Gammon's use of temporal difference learning and a neural network evaluation function has lead to several copy-cat ventures, including the commercial programs Jellyfish [2] and Snowie [3], as well as the open-source GNU Backgammon [1] (also known as Gnubg). Several versions of GNU Backgammon have sprung up on the Internet, and it has quickly become one of the most popular codebases for developers.

4 Search In Top Backgammon Programs

Much effort has been put into creating backgammon programs with increasingly stronger evaluation functions. Compared to other classic games like chess and checkers, however, not much research on improving search algorithms for games with chance nodes has been conducted in the past. Both TD-Gammon and Gnubg use a *forward pruning* approach to search, where some possible moves are eliminated before they are searched in order to reduce the branching factor of the game. Depending on the approach, using forward pruning can be a bit of a gamble, since the program is risking never seeing a good line of play, and therefore never having the chance to take it. Section 5.2 discusses Gnubg's search approach in some detail.

There are two important reasons why improvements in search have not been developed in backgammon. The first is that the current crop of neural network-based evaluation functions are very accurate, but take far too long in processing terms. For example, a complete 3-ply search of an arbitrary position in backgammon can take several minutes to complete. This is clearly undesirable from a performance perspective. The second reason has to do with the game itself. Since there are 21 distinct rolls in backgammon (with varying probability), and often up to 20 moves per roll, the effective branching factor becomes so large that, especially for a slow heuristic, searching anything deeper than a ply or two becomes impractical. It is clear due to these reasons why efforts have concentrated on developing an evaluation function that is as accurate as possible, instead of trying to grapple with the large branching factor inherent in the game.

But search *is* still important. Deeper search allows for the inaccuracies of a heuristic to be reduced, and as mentioned before, the deeper a program can search, the better that program can play. Backgammon is no exception, even with a trained neural network acting as a near-oracle. Still, it is interesting to note that improving search in backgammon programs has not been a priority, to the point where some of the GNU backgammon team are unfamiliar with the concept of Alpha-Beta search. Tesauro thinks that improvements in search will come as a result of faster processors and Moore's Law [13]. In [14] he also considered on-line Monte Carlo sampling or so-called rollout analysis run on a parallel computer. The idea is to play many games starting in a given position using only a shallow search at each decision point and to pick the root move that on average yields the highest value. The results obtained by conducting on-line rollouts on a multi-processor machine are promising and could lead to a top program running on one of today's much faster single processor machines.

5 Overview of GNU Backgammon

GNU Backgammon is an open-source backgammon program developed through the GNU Project. Development began in 1997 by Gary Wong, and has continued up to this time with contributions from dozens of people. The other five primary members today are Joseph Heled, Øystein Johansen, David Montgomery, Jim Segrave and Jørn Thyssen. The current version of Gnubg, 0.14, boasts an impressive list of features, including TD-trained neural network evaluation functions, detailed analysis of matches (including rollouts), a tutor mode, bearoff (endgame) databases, variable computer skill levels and a graphical user interface. Gnubg is also free, and since its exposure to the backgammon community was heightened, it is one of the most popular and strongest backgammon programs available. In fact, in September 2003 the results of a duel between Gnubg and Jellyfish were posted to the `rec.games.backgammon` group on UseNet [8]. Both programs played 5,000 money games, each using their "optimal" settings. Gnubg came out the winner by an average of 0.12 points per game which is a statistically significant indication that Gnubg is stronger than the expensive "professional" backgammon program Jellyfish [15].

5.1 The Evaluation Function

Gnubg has three different neural networks it uses for evaluating a backgammon position, depending on the classification of that position: either contact (at least one checker of a player is behind a checker of the other player), crashed (same as contact but with the added restriction that the player has six or less checkers left on the board, not including any checkers on the opponent's 1 or 2 points) or race (the opposite of a contact position). Since each of the three types of positions are quite different from the others, using three different neural networks improves the quality of the evaluation.

Each neural network is first trained using temporal difference learning, using self-play, similar to TD-Gammon. The input and output representations of the neural networks are also similar to TD-Gammon. The input neurons are comprised of both a raw board representation (with four neurons per point per player) as well as several hand-crafted features, such as the position of back anchors, mobility, as well as probabilities for hitting blots.

After self-play, the networks are trained against a position database (one each for the contact, crashed and race networks). The databases contain "interesting" positions, so-named because a network would return different moves depending on if they searched to either depth=1 or depth=5; and whenever a depth=5 search retains a better result than depth=1, *two* entries are made in the database for that position: the position after the depth=1 move, and the position after the depth=5 move. The positions are a mixture of randomly-generated positions as well as drawn from a large collection of human versus bot or bot self-play games, with the idea that the networks should gain more exposure to "real-life" playing situations than random situations. In total, over 110,000 positions form the position database collection used by the Gnubg team.

There is an entry for each position's cubeless evaluation in the database, along with five legal moves and their evaluations. An evaluation consists of the probabilities of normal win, gammon win, backgammon win, gammon loss and backgammon loss for the player to move (a normal loss is not explicitly evaluated, as it is just equal to $1 - P_{\text{normal win}}$). The moves in the database are chosen by first completing a depth=1 search using Gnubg, taking the top 20 moves from that search, and then searching those to depth=5; the best five moves from the depth=5 search are then kept. These moves are then "rolled out", meaning that the resulting position after the move is then played by Gnubg (doing the moves for both sides) until the game is over. Typically the number of rollouts is equal to a multiple of 36 (say, 1296) by using "quasi-random dice" in order to reduce the variance in the result, where each of the 36 possible rolls after the move is explored, with random dice thereafter. When a race condition is met in the game, the remaining rolls are played using a One-Sided Race (OSR) evaluator. The OSR is basically a table which gives the expected number of rolls needed to bear off all checkers, for a given position. It does not include any strategic elements. By using the OSR, the contact and crashed networks are judged on their own merits, and not based on the luck of the dice in the endgame. This is because race games are generally devoid of strategic play, because there is no interaction between the players anymore, not counting cube actions. Each rollout is performed in a 7-point money game setting, without cubeful evaluations.

A new network is trained against this database so its depth=1 evaluations more closely resemble a depth=5 search, and after the new network is fully trained, it then provides new entries for each position in the database. Gnubg was able to obtain a rating of about 1930 at a depth=1 setting on the First

Internet Backgammon Server (FIBS), which put it roughly at an expert level on the server.

5.2 The Search Algorithm

Gnubg's search is based on heavy use of forward pruning to either completely eliminate or greatly reduce the branching factor at move nodes, and lower the branching factor at the root, in order to keep the search fast. Pruning is based on *move filters* that define how many moves are kept at the root node (and, depending on the depth of the search, at other move nodes lower in the tree). A move filter guarantees a fixed number of candidates that will be kept at a move node (if there are enough moves), plus the addition of n candidates which are added if they are within e equity of the best move. Search is performed using iterative deepening, and root move pruning is done after each iteration. At all other move nodes, the move filter will either limit the number of moves or only keep one move. Candidate moves are chosen by doing a static evaluation of all children of the move node and choosing the n moves with the best scores; in other words, a small depth=1 search is done at all move nodes.

The branching factor at chance nodes can also be optionally reduced by limiting the number of rolls to a smaller set than 21. All roll sets are hard-coded, so no attempt is made to order rolls nor bias roll selection when a reduced set is desired.

Unfortunately, Gnubg has an unusual definition of ply. In Gnubg, a depth=1 search is called "0-ply", a depth=3 search is considered "1-ply", and so on. While most users quickly adapt to this quirk, it makes working with the code potentially tricky, since one must always remember this to avoid bugs.

For depth=1 searches, Gnubg simply performs a static evaluation of all root move candidates (a candidate being a move that has not been pruned by the move filter), and the move with the highest score is chosen. At chance nodes in the search tree, all rolls in the roll set (the set is usually all 21 rolls but it can be reduced for speed) are investigated, and the best move for each roll (chosen by simple static evaluation) is applied and expanded, until the depth cutoff is reached. In homogeneous search trees Expectimax visits $b(nb)^{(d-1)/2}$ leaves, where b is the branching factor at move nodes, n is the branching factor at chance nodes, and d is the odd search depth. By only doing a static evaluation of children at move nodes and then choosing only one for further expansion, the number of leaves of a Gnubg search tree for increasing d is $= b$ $(d = 1)$, $= bnb$ $(d = 3)$, $\leq bnnb + bnb$ $(d = 5)$, $\leq bnnnb + bnnb + bnb$ $(d = 7)$, etc. which can be bounded above by $b^2 \sum_{i=1}^{(d-1)/2} n^i \leq 2b^2 n^{(d-1)/2}$ for $d, n > 1$. Therefore, this pruning technique allows the search tree to be exponentially smaller than the full tree (with savings of at least $b^{(d-3)/2}/2$), but error is also introduced.

6 Implementation Issues and Experimental Setup

In this section we describe important implementation details and the environment in which the experiments were conducted.

6.1 Move Generation

Backgammon is not a trivial game to implement. While the board itself can be fairly easily represented by a two-dimensional array of integers, generating moves is rather complicated to not only do correctly, but also efficiently. Avoiding duplicating moves is also an important consideration because of the large branching factor for some situations (like a doubles roll for a player with checkers on several different points). The use of a transposition table can help reducing the complexity of the move generation algorithm.

6.2 Evaluation Function

Instead of going out and designing a new evaluation function for our experiments, there was already one available for use: the Gnubg codebase, which is a very strong set of trained neural networks.

While many game programs are using integer valued evaluation functions, the Gnubg evaluation function returns a floating point number (the value representing the equity of the player who just moved). Whenever search programs use floating point numbers, there is always the risk of floating point operations having rounding errors; even comparing two (seemingly) identical values may not result in the expected truth value.

To work around the uncertainty presented by floats and the continuous values they may have, we can *discretize* the values by putting them onto a one-dimensional *grid*. This involves taking the floating point number and multiplying it by a large number, and then rounding the value to the nearest integer number. That integer can then be divided by the same large number used for the multiplication. The *granularity* of the grid can be adjusted to meet the desired level of precision. A resolution of 262144 (2^{18}) was used to discretize the floating point numbers in our experiments, to ensure a fine enough granularity without being too fine for the floating point mantissa. Using floating point numbers instead of integers also meant a small performance hit.

6.3 Transposition Table

A transposition table (TT) was used to speed up the search. The TT was implemented as a simple hash table of 128 MB (more or less space could be used, depending on the amount of main memory available). Each entry was 20 bytes large, containing the value for the stored state, a flag to indicate if the entry was in use, an indicator for the depth searched, two flags to determine what kind of value for the state is stored (a lower bound, upper bound, or an exact value), the best move chosen at that state, and the hash key for that state. A Zobrist [16] hashing scheme was used.

6.4 Move Ordering and Probing

Move ordering and probe successor selection are both done with a different heuristic than the evaluation function. This is especially a concern with a heavy evaluation function such as Gnubg's. Probe successors were selected as follows: moves that hit opponent blots were taken first (best quality), moves that formed a point were taken second (good quality), and if no moves met either condition, the first move was chosen. Move ordering worked a little differently. Move sorting was done by scoring a move based on a number of criteria: the number of opponent checkers moved to the bar, the number of free blots it left open to hit, and the number of safe points (2 or more checkers) made. These criteria remained the same for all moves during the game.

6.5 Experimental Design

For obtaining quality results, all experiments were run on relatively new hardware. Two undergraduate labs (one of 22 machines and one of 34 machines) were made available for distributed processing. All machines were identical, each with an Athlon-XP 1.8 GHz processor and 512 MB of RAM, as well as 27 GB of local disk space (to bypass using NFS). Each machine used Slackware Linux with kernel version 2.4.23 and had gcc version 3.2.2 installed. All search software was coded in C.

While all experiments were performed when the labs were largely idle, all experiments were nevertheless subjected to possible skewing if students logged into a host to use it. However, less than a dozen students logged into any one of the machines during the entire experimental phase, so fluctuations in results due to lost CPU cycles are negligible. Since each machine only had a modest amount of free RAM, the transposition table was kept to a relatively small size of 128 MB. Gnubg's codebase was used for the evaluation function and all executables were compiled under gcc with `-O3` optimization.

7 Performance

Using randomly-seeded positions does not make sense for backgammon, since it is difficult to generate random positions which look "reasonable" in backgammon terms. Instead of randomly generating positions, a position database was used. The database came from the Gnubg team, used for training the neural network. It is comprised of several thousands of positions classified into different categories. The contact position database was made available for experiments. The results of searching these positions are therefore more applicable to real-world performance compared to random positions.

500 randomly selected contact positions were used for testing. Each was searched to depths of 1, 3 and 5 by Expectimax, Star1 and Star2. There is a direct relation between time and node expansions, as the Gnubg evaluation function is very heavy in terms of CPU usage (over 90%).

In Figures 1 (CPU time) and 2 (node expansions) graphed on a logarithmic scale, we can see some variation in the amount of effort Star2 requires to complete a search at depth=5, which reflects the variety of backgammon positions during a match. Each of the 25 positions shown were selected at random from the Gnubg

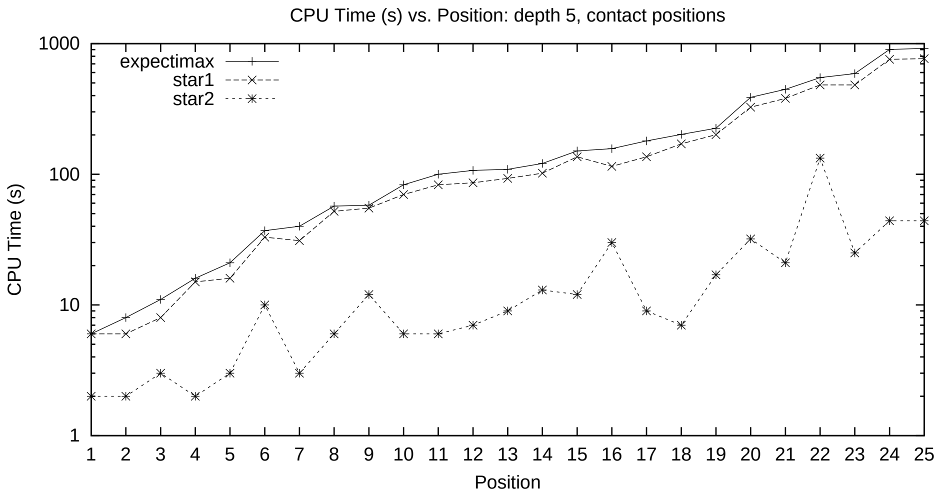

Fig. 1. Time used (s) at d=5 for 25 contact positions

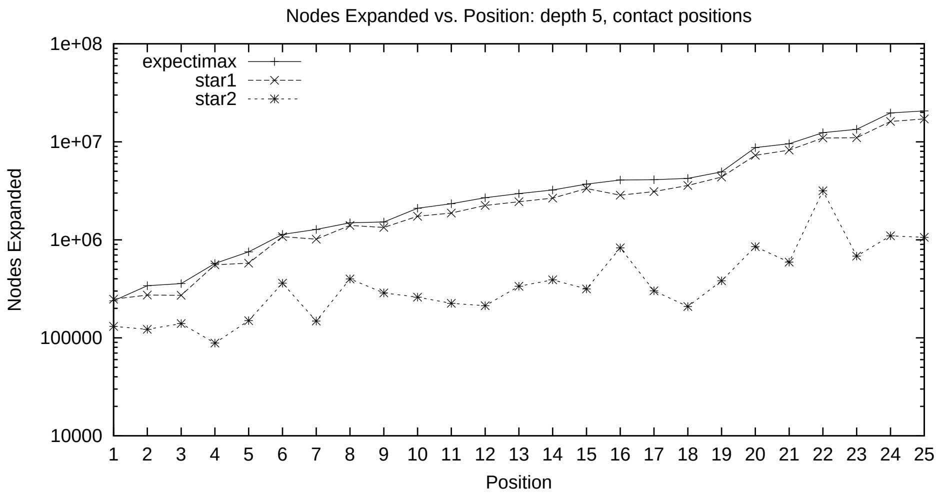

Fig. 2. Node expansions at d=5 for 25 contact positions

contact position database, searched by all three algorithms, and then sorted in order of Expectimax time. The variation in savings for Star2 for the 25 positions goes from about 75% to about 95%. Expectimax and Star1 closely follow each other, where Star1 has only a slight decrease in overall costs.

Table 1 summarizes the time usage over 500 positions. Star2 is clearly the most efficient of the algorithms by over a factor of 10, but even at 21 seconds per search, this would probably still be too slow for tournament play. Figure 3 shows the average number of node expansions over 500 positions, graphed on a logarithmic scale.

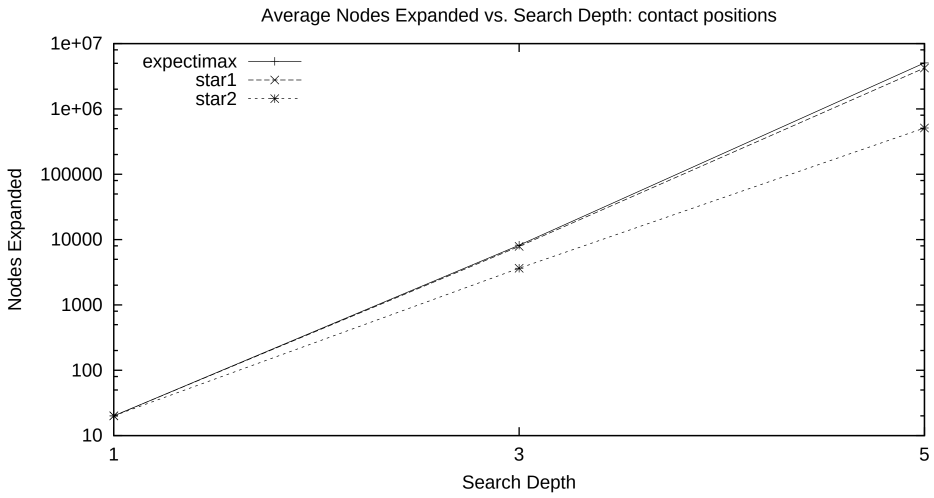

Fig. 3. Average number of node expansions over 500 contact positions (interpolation is used to indicate the trend)

Table 1. Average time (s) over 500 contact positions

	Expectimax	Star1		Star2	
	μ	μ	%	μ	%
$d = 3$	1.1	1.1	100	1.0	91
$d = 5$	315.0	258.6	82	21.0	7

7.1 Probe Efficiency

Table 2 shows the resulting probe efficiency for using Star2. The "quick" successor selection scheme for backgammon is relatively weak because backgammon is a complicated game. Still, these results are better than Ballard's, whose probing was never successful more than about 45% of the time. The improvement here is probably due to better move ordering.

Table 2. Probe efficiency for backgammon

$d = 3$	$d = 5$
68.9%	64.2%

7.2 Odd-Even Effect

Many 2-player game-playing computer programs suffer from what is called the *odd-even* effect, where alternating levels of move nodes will give scores that are either optimistic (for odd-ply searches) or pessimistic (for even plies). For

Table 3. Root value difference statistics for two trials (3200 moves each)

	Average	Abs. Average	Abs. Std. Dev.
d=1 vs. d=3	0.0280	0.0336	0.0397
d=1 vs. d=5	0.0018	0.0134	0.0184

	Average	Abs. Average	Abs. Std. Dev.
d=1 vs. d=3	0.0267	0.0328	0.0389
d=1 vs. d=5	0.0014	0.0126	0.0172

example, a depth=1 search in any game will tend to be optimistic, since we are only investigating the moves currently available to us. The odd-even effect comes from the way in which an evaluation function is created, which generally tries to score the position for the player-to-move.

Table 3 shows the results of two different trials of 3200 backgammon positions. The positions were generated as a continuous sequence of cubeless money games, with the computer playing for both sides. This generated a decent set of "real-world" moves for backgammon. The table shows the average difference, absolute average difference, and absolute standard deviation in the root node value when comparing searches of the same positions to different depths.

Numbers on both tables are very similar. The results show that the evaluation of the root node for a depth=1 search is very close to the evaluation for a depth=5 search, on average. When absolute differences are used instead, depth=1 is not as good as a predictor for a depth=5 search, but the difference is reasonably small (only about 0.01 points).

The differences between depth=1 and depth=3 are much more striking. Both the average and the absolute average difference between them is nearly the same. In fact, the average difference is positive, which means that the depth=3 search value is usually significantly less than the value from a depth=1 search.

These results show a tangible odd-even effect with the Gnubg evaluation function. Even if searches to different depths produce different values for the root, the move chosen at the root usually is the same across searches of different depths. This means the evaluation function itself is very consistent between depths. These results also show that a depth=1 search value is a reasonable predictor for a depth=5 search value for the same position.

7.3 Tournaments

Another way to measure an algorithm's performance is to pit it against itself in a tournament, where each player is searching to a different depth. The question to answer, then, is if deeper search increases real performance in the game. Tournaments were therefore run between combinations of players searching to depths of 1, 3, and 5. Each tournament used a file containing a sequence of seed values, such that they all would then have the same sequence of dice rolls across each tournament. The starting roll for each game was pre-set by a testing script, and went through all combinations sequentially, in order to reduce variance. Furthermore, for each opening roll and sequence of dice rolls, both players were given an opportunity to be the starting player. A file containing 9,000 seeds was used, and therefore a total of 18,000 games per tournament were played. We chose cubeless money games as tournament mode to study *-Minimax checker play performance. Gammons and backgammons only counted as one-point wins. Tournaments were set up between the Gnubg search function and itself, and Star2 against itself. Since Gnubg also has a facility for adding deterministic noise to an evaluation, different noise settings were also investigated.

While we expected that deep search was beneficial for tournament performance just like in chess and Othello, this was not evident in backgammon. Table 4 shows the results of Gnubg playing against itself at different depth settings. We can see from the table that a depth=5 search barely shows any significant improvement over shallower searches. In fact, the three depth settings are nearly identical. This suggests that deeper searches are only finding better moves a small fraction of time, which suggests that the three searches are choosing the same move just about every time. That means the Gnubg evaluation function must be extremely consistent between depth levels. Table 5 A) shows the Star2 performance when playing against itself, in the same manner as Table 4. Deep search is still pretty much irrelevant using the Gnubg evaluation function as-is. Since the evaluation function is so consistent, results were also desired for a less consistent setting. Instead of developing a new evaluation function, noise can just be added to the evaluation function. Gnubg has a built-in noise generator already, which can add either deterministic or non-deterministic noise to each evaluation. Since it is highly desirable that the evaluation for a state be always deterministic, especially when transpositions are possible, another tournament using deterministic noise was added. Only modest amounts of noise were added, consistent with an "intermediate" and "advanced" level of play for Gnubg (noise settings $n = 0.03$ and $n = 0.015$). Tables 5 B) and C) show tournament results in an identical manner to the previous two tables. Now, deeper search is paying off to a significant degree. For $n = 0.03$ a depth=1 search now loses to a depth=5 search 64% of the time. Depth=1 fares slightly better against depth=3 at about 42% winning percentage. Depth=5 wins slightly less than 55% of the time against depth=3, but it is still a tangible amount. Deep search helps to mitigate evaluation function errors by adding more foresight to the move decision process. Adding deterministic noise to the Gnubg evaluation function shows that deep search becomes important again in backgammon.

Table 4. Tournament results for Gnubg with no noise versus Gnubg with no noise, 18,000 games per matchup. Reported are winning percentages and average points per game in view of the top player for search depths 1, 3, and 5.

	1	3	5
1	*	50.92 +0.018	51.79 +0.036
3	49.08 −0.018	*	51.63 +0.037
5	48.21 −0.036	48.37 −0.037	*

Table 5. Tournament results for Star2 vs. Star2 (4000 games per matchup). Reported are winning percentages and average points per game in view of the top player for various evaluation function noise levels and depth pairs.

A) noise level $n = 0.000$

	1	3	5
1	*	50.60 +0.012	51.60 +0.032
3	49.40 −0.012	*	52.52 +0.050
5	48.40 −0.032	47.48 −0.050	*

B) noise level $n = 0.015$

	1	3	5
1	*	55.53 +0.111	54.67 +0.093
3	44.47 −0.111	*	51.57 +0.031
5	45.33 −0.093	48.43 −0.031	*

C) noise level $n = 0.030$

	1	3	5
1	*	59.00 +0.180	64.20 +0.284
3	41.00 −0.180	*	53.40 +0.068
5	35.80 −0.284	46.60 −0.068	*

8 Conclusions and Future Work

Star2 and Star1 both outperform Expectimax on single position searches. Star2 has a significant savings in costs even at depth=5, mostly due to the large branching factor inherent in backgammon. Gnubg's evaluation function is time consuming which means that performance is strongly linked to eliminating as many leaves as possible.

To our surprise strong cubeless money game tournament performance is not much improved by deeper search. The Gnubg evaluation function is sufficiently well-trained and consistent that searches to increasing depths almost always choose the same move at the root. When the searches do not agree on the best move it is usually because they are searching a tactical position. But even the occurrence of tactical positions is relatively infrequent, and the benefits of deep search in these situations is usually washed away by the randomness of the dice rolls. However, when small amounts of deterministic noise are introduced into the search, deep search once again becomes important as the evaluation function becomes less consistent and less accurate. This suggests future *-Minimax experiments in domains less affected by chance events or in which good evaluation functions are unknown. Comparing *-Minimax with Monte Carlo search w.r.t. playing strength vs. search time in various domains would also be interesting.

Gnubg's forward-pruning search method works very well for its evaluation function, since the best move candidate at the root is unlikely to change much from one iteration to the next. Deeper search catches some tactical errors in some situations, but because tactical situations can be thrown completely askew by a single lucky roll, deep search does not pay huge dividends.

With an excellent evaluation function such as Gnubg's set of neural networks, checker play is virtually perfect, even with shallow search. However, since backgammon matches are generally played with a doubling cube, and cube decisions are usually the most important part of the game, this work should be extended to cubeful games and cube decisions. Being able to see farther ahead in these situations and to estimate the winning probability more accurately can make or break a player's chances of winning. The odd-even fluctuations we recognized indicate that although neural networks may be almost optimal in ordering moves, there is still room for improvement w.r.t. absolute accuracy. Looking deeper by using *-Minimax and considering doubling actions in the search is a promising approach.

In view of the close tournament outcomes reported in this paper, future experiments should also consider gammons and backgammons to produce results more relevant to actual tournament play. Moreover, choosing starting positions unrelated to the Gnubg evaluation function tuning could generate results more favorable to deeper search.

Since Star2 is so reliant on successful probing, a more powerful probing function would also increase performance. Right now successors are picked according to some ad hoc rules about good backgammon play for quickly choosing a child, but there are perhaps better techniques for making this decision — e.g. probing functions amenable to fast incremental updating similar to piece–square tables in chess.

There are other stochastic perfect-information games which could benefit greatly from the use of *-Minimax search. One excellent domain would be the German tile-laying game Carcassonne. Being able to see a line of play from even five or six tiles out could result in expert play. Because computers can also keep track of which tiles have been played better than humans, a computer player could also avoid many of the pitfalls which plague humans. However, since the branching factor at chance nodes after the root starts at 40 (when using the most common expansion tileset, Inns & Cathedrals), some form of statistical sampling may be required to jumpstart the computer player.

Acknowledgements

This research was supported by grants from the Natural Sciences and Engineering Research Council of Canada (NSERC) and Alberta's Informatics Circle of Research Excellence (iCORE).

References

1. GNU Backgammon (backgammon software). http://www.gnubg.org/.
2. Jellyfish (backgammon software). http://jelly.effect.no/.
3. Snowie (backgammon software). http://www.bgsnowie.com/.
4. B.W. Ballard. The *-Minimax Search Procedure for Trees Containing Chance Nodes. *Artificial Intelligence*, 21(3):327–350, 1983.
5. H. Berliner. Backgammon Computer Program Beats World Champion. *Artificial Intelligence*, 14:205–220, 1980.

6. T. Hauk. Search in Trees with Chance Nodes. *M.Sc. Thesis, Computing Science Department, University of Alberta*, 2004.
7. T. Hauk, M. Buro, and J. Schaeffer. Rediscovering *-Minimax. *Accepted at the Computers and Games conference*, 2004.
8. M. Howard. The Duel, August 5, 2003. [Online] rec.games.backgammon.
9. D. Michie. Game-playing and game-learning automata. In L. Fox, editor, *Advances in Programming and Non-Numerical Computation*, pages 183–200. Pergamon, New York, 1966.
10. R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.
11. G. Tesauro. Neurogammon: A neural-network backgammon learning program. In *Heuristic Programming in Artificial Intelligence*, pages 78–80, 1989.
12. G. Tesauro. Temporal Difference Learning and TD-Gammon. *Communications of the ACM*, 38(3), March 1995.
13. G. Tesauro. Programming Backgammon Using Self-Teaching Neural Nets. *Artificial Intelligence*, 134(1-2):181–199, 2002.
14. G. Tesauro and G.R. Galperin. On-line policy improvement using Monte Carlo search. *Proc. of NIPS*, pages 1068–1074, 1996.
15. J. Thyssen, 2003. Personal Communication.
16. A.L. Zobrist. A New Hashing Method with Applications for Game Playing. *ICCA Journal*, 13(2):69–73, 1990.