Rediscovering *-Minimax Search

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. The games research community has devoted little effort to investigating search techniques for stochastic domains. The predominant method used in these domains is based on statistical sampling. When full search is required, EXPECTIMAX is often the algorithm of choice. However, EXPECTIMAX is a full-width search algorithm. A class of algorithms were developed by Bruce Ballard to improve on EXPECTIMAX's runtime. They allow for cutoffs in trees with chance nodes similar to how ALPHA-BETA allows for cutoffs in MINIMAX trees. These algorithms were published in 1983—and then apparently forgotten. This paper "rediscovers" Ballard's *-MINIMAX algorithms (STAR1 and STAR2).

1 Introduction

Games can be classified as requiring skill, luck, or a combination of both. There are many games which involve both skill and chance, but often simple games of chance do not involve much strategy. For example, chess is clearly a game of skill, but luck only factors into the equation when we hope that our opponent makes a mistake. On the other hand, games of chance like roulette offer little opportunity to use skill, besides, perhaps, knowing when to quit. Games that involve chance usually involve dice or cards. Competitive card games combine skill and chance by requiring players to use strategic thinking, while they manage the uncertainly involved; since card decks are shuffled, the game's outcome is not certain. Since a player's cards tend to be hidden in card games, each player will have *imperfect information* about the game state. There are few *perfect information* games which blend skill and chance – games where nothing is hidden, yet nothing is certain. Backgammon is one such game.

For games where chance is a critical component of the game, statistical sampling has been often used to factor the random element out of the search. The method involves repeated trials where the outcome of the chance events are randomly decided before the search begins, and then search is run normally on the resulting tree. Since each chance event has one successor, they just become intermediary nodes in the tree. For example, for games that involve dice, the chance events can be determined in advance or on-demand when a chance node is met in the tree. To get a good statistical sample, the number of trials must be high enough to approximate the true distribution (for backgammon, this is often in the tens or hundreds of thousands of trials). Although the method is popular (it

is easy to implement), it is not without its limitations (see, for example, [3]). The technique has been successfully used in backgammon (for post-mortem roll-outs [8]), bridge (for the play of the hand [3]), poker (computing expected values for betting decisions [2]), and Scrabble (anticipating opponent responses [7]).

Perfect information games without chance benefit from deep search. A natural question is whether there is a counterpart to the ALPHA-BETA algorithm for perfect information games with chance. Backgammon programs typically do almost no (or limited) search as part of their move decision process [4]. Would these programs benefit from an ALPHA-BETA-like algorithm?

Bruce Ballard enhanced ALPHA-BETA search to traverse trees consisting of min, max and chance nodes [1]. The ideas centered around a family of algorithms that he called *-MINIMAX, with STAR1 and STAR2 being the main variants. His work was published in 1983, but has received very little attention. Indeed, it appears that his work has been all but forgotten (there are few references to it, and the paper is not online).

The main contributions in this paper are as follows:

1. Re-introducing Ballard's ideas to the research community.
2. Negamax pseudo-code for STAR2.
3. Updating the algorithms to reflect non-uniform probabilities for chance nodes.
4. An understanding of the relative strengths of STAR1 and STAR2.
5. Updating the algorithms to take advantage of 20 years of ALPHA-BETA search enhancements.

This paper "re-discovers" Ballard's work. The algorithms are important enough that they need to be updated and re-introduced to the games research community. This work shows that STAR1 and STAR2 are worthy of consideration as a full-width search-based approach to dealing with stochastic domains. A companion work presents experimental results for these algorithms for the game of backgammon [5].

2 Search in Stochastic Domains

Most games extensively studied in the AI literature are non-stochastic games of perfect information. With the addition of a random element (roll of the dice; dealing of cards), chance events are introduced into the game tree. Hence, we need to add a new kind of node to our game tree: a chance node. A chance node will have successor states like minimization (min) or maximization (max) nodes, but each successor is associated with a probability of that state being reached. For example, a chance node in a game involving a single dice would have six successor nodes below it, each representing the state of the game after one of the possible rolls of the dice, and each reachable with the same probability of $\frac{1}{6}$.

The element of chance completely changes the landscape that search algorithms work on. In games of chance, we cannot say for certain what set of legal moves the opponent will have available on their turn, so we cannot be certain to avoid certain outcomes. The introduction of chance nodes means that we can no

longer directly apply standard ALPHA-BETA to games of chance. Chance nodes act as intermediaries, by specifying the state the game will take before a choice of actions becomes available. Before we can search trees with chance nodes, we have to figure out how to handle them.

2.1 Expectimax

The baseline algorithm for trees with chance nodes (analogous to MINIMAX for games without chance nodes) is the EXPECTIMAX algorithm [6]. Just like MINIMAX, EXPECTIMAX is a full-width, brute-force algorithm. EXPECTIMAX behaves exactly like MINIMAX except it adds an extra component for dealing with chance nodes (in addition to min and max nodes). At chance nodes, the heuristic value of the node (or EXPECTIMAX value) is equal to the weighted sum of the heuristic values of its successors. For a state s, its EXPECTIMAX value is calculated with the function:

$$Expectimax(s) = \sum_i P(child_i) \times U(child_i)$$

where $child_i$ represents the ith child of s, $P(c)$ is the probability that state c will be reached, and $U(c)$ is the utility of reaching state c. Evaluating a chance node in this way is directly analogous to finding the utility of a state in a Markov Decision Process.

EXPECTIMAX is given in Figure 1, which makes use of the following functions:

1. `terminal()` that returns true if and only if a given state is terminal,
2. `evaluate()` that returns the heuristic evaluation of a state,
3. `numChanceEvents()` to specify how many different values the chance event can take,
4. `applyChanceEvent()` to apply the chance event to the state,
5. `eventProb()` to determine the probability of the chance event taking that value, and
6. `search()` calls the appropriate search function depending on the type of node that follows the chance node (min, max, or chance).

```
float Expectimax(Board board, int depth, int is_max_node) {
   if(terminal(board) || depth == 0) return (evaluate(board));
   N = numChanceEvents(board);
   for(sum = 0, i = 1; i <= N; i++) {
      succ = applyChanceEvent(board,i);
      sum += eventProb(board,i) * search(succ, depth-1, is_max_node);
   }
   return (sum);
}
```

Fig. 1. The EXPECTIMAX algorithm

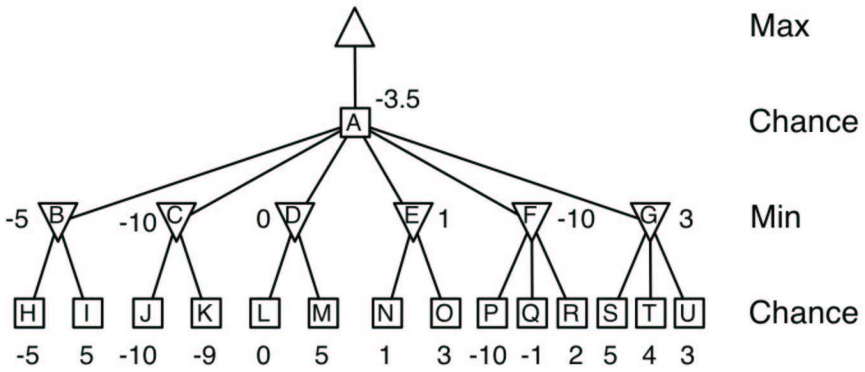

Fig. 2. An EXPECTIMAX tree

For games where chance nodes alternate with player turns, we can use MINIMAX for searching, with the modification that MINIMAX's recursive call uses EXPECTIMAX instead of itself. We also use floating point numbers instead of integers for return values, since probabilities are real numbers and the sum may have a fractional component.

Figure 2 illustrates how EXPECTIMAX works. If we assume that each of the 6 branches at the chance node have the same probability (such as would be the case for a single dice), then each child contributes 1/6th of the value of the node: $value = -5 \times \frac{1}{6} + -10 \times \frac{1}{6} + 0 \times \frac{1}{6} + 1 \times \frac{1}{6} + -10 \times \frac{1}{6} + 3 \times \frac{1}{6} = -3.5$.

Assume that the search tree has a fixed branching factor B, and a search is being conducted to depth D (where a depth, or ply, consists of a min, max, or chance node). While the worst-case time complexity for MINIMAX is $O(B^D)$, the worst-case for EXPECTIMAX (for trees with alternating levels of chance nodes) is $O(B \times B^{\frac{D-1}{2}} \times N^{\frac{D-1}{2}})$ (for D odd), where N is the branching factor at chance nodes (for example, in backgammon's case, $N = 21$ since there are twenty-one distinct rolls). As an example of the explosive effect of chance nodes even on shallow searches, there would be approximately 3.5 million nodes in a 3-ply search of an arbitrary backgammon position. If an evaluation function took 0.05 ms to complete (about the speed of GNU backgammon's neural network on a modern computer), then a 3-ply search would take about 3 minutes to complete, a 4-ply search would take about 21 hours, and a 5-ply search would be roughly a year.

2.2 *-Minimax

Bruce Ballard was the first to develop a technique, called *-MINIMAX, for enabling chance node cutoffs [1]. He proposed two versions of his algorithm, called STAR1 and STAR2. He also further refined the second algorithm to handle more general cases and have parameters to control functionality, and called the new

version STAR2.5. All the experiments that Ballard performed were in a rather abstract domain. He did not use a real domain to validate his results.

The basic idea of EXPECTIMAX is sound, but slow. Just as we can derive a strategy for obtaining cutoffs in MINIMAX to obtain ALPHA-BETA, so too can we derive a strategy for obtaining cutoffs in EXPECTIMAX. Since there are three different types of nodes in a game tree for games with chance, there are three cases we need to consider for cutoffs. Since max and min nodes work the same way in trees with chance nodes as they do in trees without chance nodes, we get the cutoff strategies for those nodes for "free". All we need to concern ourselves with are chance nodes. If we pass alpha and beta values to chance nodes as we do min and max nodes, and we pass alpha and beta values from chance nodes to min and max nodes, all that is left to consider is exactly what values we can pass, and how they will be used.

In the first case, chance nodes can have a search window just like min and max nodes, using alpha and beta values to determine if further search below the node is relevant. However, these alpha and beta values cannot be used just like they are used in min or max nodes, because the child of a chance node cannot be chosen deterministically (unless there is only one child). We can obtain a cutoff, however, if the EXPECTIMAX value of a chance node falls outside the alpha-beta window. The problem is that we cannot know the exact EXPECTIMAX value of a chance node before we search all of its children. However, if we know bounds on the range of values leaf nodes can take (called L and U, respectively, using Ballard's notation), we can determine bounds on the value of a chance node based on the worst-case conditions for the alpha and beta values.

If we have reached the ith successor of a chance node, after having searched the first $i-1$ successors and obtained their backed-up values (which we will call $V_1 \ldots V_{i-1}$), then we can determine a bound for the value of the chance node. In the worst case, all the unsearched children will have a value of L, and in the best case, all the unsearched children will have a value of U. Therefore, the lower bound of a chance node's value, where V_i represents the true value of successor i and there are N different equally-likely chance events, is equal to

$$\frac{1}{N}((V_1 + \ldots + V_{i-1}) + V_i + L \times (N - i))$$

and the upper bound is equal to

$$\frac{1}{N}((V_1 + \ldots + V_{i-1}) + V_i + U \times (N - i))$$

These bounds determine the range in which the EXPECTIMAX value for a chance node must lie. We can use this range to generate cutoffs. Recall that the chance node itself was passed alpha and beta values. We can cut off our search if the lower bound of the EXPECTIMAX range for the chance node ever exceeds or equals beta,

$$\frac{1}{N}((V_1 + \ldots + V_{i-1}) + V_i + L \times (N - i)) \geq beta$$

or the upper bound is ever less than or equal to alpha,

$$\frac{1}{N}((V_1 + \ldots + V_{i-1}) + V_i + U \times (N - i)) \leq alpha \quad (1)$$

where $(V_1 + \ldots + V_{i-1})$ are the accurate values for the first $i - 1$ children of a node, V_i is the value for current node being searched, and $(U \times (N - i))$ and $(L \times (N - i))$ represent the best/worst-case assumptions for the values of the remaining nodes. In either equation, we can solve for V_i, and use the value as either an alpha or a beta value for the next child.

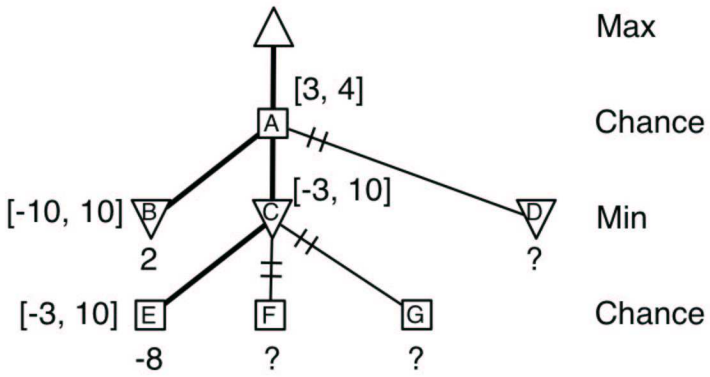

Fig. 3. Fragment of a *-MINIMAX tree

Take the following example shown in Figure 3, where heuristic values range from $L = -10$ to $U = 10$, inclusive. The top-most chance node, A, is entered with a window of alpha=3 and beta=4 (we will write this as [3,4]). Because we have not searched any of its children yet, we know its value lies in the range [-10,10], and the alpha and beta values for the first child are equal to $\frac{1}{3}(3 \times L) = L$ and $\frac{1}{3}(3 \times U) = U$, which is also [-10,10]. Assume that the first child (B) is searched and a value of 2 is returned. We now know the EXPECTIMAX range for the chance node is between $\frac{1}{3}(2 + 2 \times L) = \frac{1}{3}(-18) = -6$ and $\frac{1}{3}(2 + 2 \times U) = \frac{1}{3}(22) = 7\frac{1}{3}$. Since -6 is not greater than 4 and $7\frac{1}{3}$ is not less than 3, this child did not create a cutoff. Before we search the next child, we need to recalculate the alpha and beta values we want to pass down to it: $\frac{1}{3}(2 + V_i + (1) \times L) \geq beta \Rightarrow V_i \geq 20$, and $\frac{1}{3}(2 + V_i + (1) \times U) \leq alpha \Rightarrow V_i \leq -3$.

We will call the V_i value associated with alpha A_i, and the V_i value associated with beta B_i, at chance nodes, and so we will pass a window of $[A_i, B_i]$ to successor i when we search it.

Since the upper bound on a leaf node is 10, we will pass a window of [-3, 10] to the next child, C. Assume the next node searched at the bottom, E, has a value of -8. This will trigger a cutoff at C, because -8 lies outside the lower

bound of the window (which is -3). The cutoff at C will also trigger a cutoff at the topmost chance node A. In fact, this could also trigger further cutoffs along this branch all the way up to the root; the possibility for two or more cutoffs to occur without intervening leaf searches is unique to trees with chance nodes, and not found in typical MINIMAX trees.

2.3 Star1

When we translate the ability to obtain chance node cutoffs into a procedural representation, we end up with STAR1, Ballard's first version of the *-MINIMAX algorithm. By re-arranging equations (1) and (2), the alpha value for the ith successor, A_i, can be determined with

$$A_i = N \times alpha - (V_1 + \ldots + V_{i-1}) - U \times (N - i)$$

and the beta value for the ith successor, B_i, with

$$B_i = N \times beta - (V_1 + \ldots + V_{i-1}) - L \times (N - i)$$

where alpha and beta are the respective values passed to the chance node. These equations can be rewritten to be more efficient by initializing the two values as:

$$A_1 = N \times (alpha - U) + U; \quad B_1 = N \times (beta - L) + L$$

and updating them with

$$A_{i+1} = A_i + U - V_i; \quad B_{i+1} = B_i + L - V_i$$

where $i = 2 \ldots N$. When a chance node only has one successor ($N = 1$), the initial A and B values for the chance node take on the alpha and beta values initially passed to the node.

Figure 4 shows the resulting STAR1 algorithm. The algorithm makes use of the following additional functions:

1. `numSuccessors()` that returns the number of successors a state has,
2. `successor()` that returns a new state, and
3. `search()` which calls the appropriate function, either STAR1 for a chance node or ALPHA-BETA for a min or max node.

This assumes that all values for the chance event have equal probability.

Note that our version of STAR1 extends the algorithm to include the fail-soft ALPHA-BETA enhancement. When further search at a node is unnecessary, rather than returning the window bound (alpha or beta) the code returns the lowest upper bound or highest lower bound that would be achievable if the remaining successors were searched.

An example of STAR1 cutoffs is shown in Figure 5. The uppermost chance node is initially passed bounds of [-2,2]. The initial value for A is equal to $N \times (alpha - U) + U = 6 \times (-2 - 10) + 10 = -62$ and B is equal to $N \times (beta - L) + L =$

```
float Star1(Board board, float alpha, float beta, int depth) {
   if(terminal(board) || depth == 0) return (evaluate(board));
   N = numSuccessors(board);
   A = N*(alpha-U) + U;
   B = N*(beta-L) + L;
   for(vsum = 0, i = 1; i <= N; i++) {
      AX = max(A, L);
      BX = min(B, U);
      v = search(successor(board,i), AX, BX, depth-1);
      vsum += v;
      if(v <= A) { vsum += U*(N-i); return (vsum/N); } // Fail soft
      if(v >= B) { vsum += L*(N-i); return (vsum/N); } // Fail soft
      A += U - v;
      B += L - v;
   }
   return (vsum/N);
}
```

Fig. 4. The STAR1 algorithm, adapted from [1]

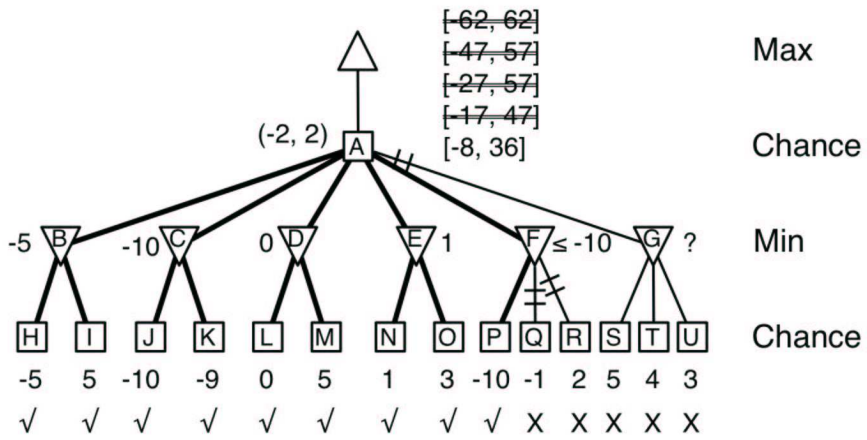

Fig. 5. A STAR1 tree

$6\times(2+10)-10 = 62$. After searching the root's first successor, the A and B values are adjusted for the second successor (C), where A becomes $-62+10+5 = -47$ and B becomes $62-10+5 = 57$. As we continue to search the children of the root sequentially, we can see that the root node's $[A,B]$ window is equal to [-8,36] by the time it reaches its fifth child F, who gets an ALPHA-BETA window of [-8,10]. After searching P, which has a value of -10, F gets an immediate cutoff and returns this value to its parent A, the uppermost chance node, which triggers another cutoff because -10 falls outside its lower bound of -8. The other children of F, as well as the sixth successor G, do not need to be searched, as we can prove that the EXPECTIMAX value of A must be less than -2 (it is in fact $-3\frac{1}{2}$, which we can read from Figure 2).

2.4 Star2

While STAR1 results in an algorithm which returns the same result as EXPECTIMAX, and uses fewer node expansions to obtain the same result, its results are generally not very impressive. One reason is that STAR1 is agnostic about its successors; it has no idea what kind of node (min, max or chance) will follow it, but even if it did, it would not be able to take advantage of that knowledge. However, game domains are fairly regular; for example, in a standard MINIMAX tree, min and max nodes are on levels that strictly alternate. Min always follows max, and max always follows min. In games like backgammon, where each player rolls the dice, then moves, we end up with a tree like a MINIMAX tree, except we insert a chance node immediately after any non-terminal min or max node. In other words, we add a layer of chance nodes between each layer of nodes in a standard MINIMAX tree. Ballard refers to trees with this structure as *regular* *-MINIMAX trees, an example of which is shown in Figure 6, where +, - and * refer to max, min and chance nodes, respectively. The regular structure assumed for STAR2 is not essential, as any tree can be transformed into such a form.

Another drawback to STAR1 is due to its pessimistic nature. We may potentially search nearly all the children of a chance node before a cutoff is obtained, because we assume that all unseen children have a worst-case evaluation. However, children of a successor of a chance node will tend to have values which are highly correlated. Instead of searching each child of a chance node fully and sequentially, and give a value of L to any children we haven't seen yet, we can

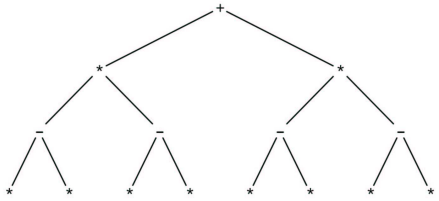

Fig. 6. A regular *-MINIMAX tree

get a more accurate picture just by searching a single successor of each child. This value we get for the child then becomes a bound on the true value for the child (a lower bound if the child is a max node, and an upper bound if the child is a min node). It is likely that the bound will be much better than L, especially if we chose the child well. We will therefore introduce this phase of speculative search (which we will call the *probing phase*) before sequentially searching each child, in order to obtain a quicker cutoff.

We need to modify the equations used to generate A and B in STAR1 to reflect the new use of a probing phase in STAR2. For STAR2's probing phase, we derive the bounds for A and B just like we do in STAR1's search phase, except we do not have alpha cutoffs at chance nodes followed by min nodes (since we can only get an upper bound on those children), and we do not have beta cutoffs at chance nodes followed by max nodes (since we can only get a lower bound on those children).

We obtain a cutoff in STAR2's search phase if

$$\frac{(V_1 + \ldots + V_{i-1}) + V_i + (W_{i+1} + \ldots + W_N)}{N} \leq alpha$$

or

$$\frac{(V_1 + \ldots + V_{i-1}) + V_i + (W_{i+1} + \ldots + W_N)}{N} \geq beta$$

where $(W_1, ..., W_N)$ are the probed values for the N children of a node, obtained during the probing phase.

The alpha value for the ith successor, A_i is now obtained with

$$A_i = N \times alpha - (V_1 + \ldots + V_{i-1}) - (W_{i+1} + \ldots + W_N) \tag{2}$$

and the beta value for the ith successor, B_i with

$$B_i = N \times beta - (V_1 + \ldots + V_{i-1}) - (W_{i+1} + \ldots + W_N) \tag{3}$$

Like with STAR1, these equations can be rewritten:

$$A_1 = N \times alpha - (W_2 + \ldots + W_N); \quad B_1 = N \times beta - (W_2 + \ldots + W_N)$$

and updated by

$$A_{i+1} = A_i + W_{i+1} - V_i; \quad B_{i+1} = B_i + W_{i+1} - V_i$$

where $i = 2 \ldots N$.

Figure 7 shows the resulting STAR2 algorithm using a Negamax formulation (hence, a chance node is always followed by a max node). To get values for the probing phase, we need a procedure similar to ALPHA-BETA since successors are min or max nodes. Figure 8 shows the Probe algorithm. Figure 9 shows the PickSuccessor algorithm used by `Probe`, which is explained in more detail below.

Consider the tree in Figure 10, to see STAR2's strength. It is the same tree used in the previous example with STAR1. For the probing phase, the alpha

```
float nStar2(Board board, float alpha, float beta, int depth) {
   if(terminal(board) || depth == 0) return (evaluate(board));
   N = numSuccessors(board);
   A = N*(alpha-U);
   B = N*(beta-L);
   AX = max(A, L);
   /* Probing phase */
   for(vsum = 0, i = 1; i <= N; i++) {
      B += L;
      BX = min(B, U);
      w[i] = nProbe(successor(board,i), AX, BX, depth-1);
      vsum += w[i];
      if(w[i] >= B) { vsum += L*(N-i); return (vsum/N); }
      B -= w[i];
   }
   /* Search phase */
   for(vsum = 0, i = 1; i <= N; i++) {
      A += U;
      B += w[i];
      AX = max(A, L);
      BX = min(B, U);
      v = nAlphaBeta_MM(successor(board,i), AX, BX, depth-1);
      vsum += v;
      if(v <= A) return { vsum += U*(N-i); return (vsum/N); }
      if(v >= B) return { vsum += L*(N-i); return (vsum/N); }
      A -= v;
      B -= v;
   }
   return (vsum/N);
}
```

Fig. 7. Negamax formulation of the STAR2 algorithm

value changes just like with STAR1 but the beta value does not. In this case, we only need to search five leaves: H, J, L, N and P, because by the time we reach child F, we give it a window of [-8,10]. Since P has a value of -10, this causes a cutoff at F. It also causes a cutoff at A since F returns a value of -10, which is less than or equal to A. In this example our Probe function did a good job and we always chose the best child for probing (fortuitously), so we obtained a cutoff after searching about half the nodes STAR1 searches.

As the branching factor increases, probing becomes even more effective, because sequential searching of children becomes more and more time-consuming. But even with small branching factors, probing can still be effective.

In his paper, Ballard did not specify how `Probe` should choose a successor besides to say it could be done "at random or by appeal to a static evaluation function" [1]. Since the domain he used was limited to a depth=3 tree, all the probes done in his experiments were on leaf nodes. His domains also only had

```
float Probe_Min(Board board, float alpha, float beta, int depth) {
   if(terminal(board) || depth == 0) return (evaluate(board));
   choice = PickSuccessor(board);
   return (AlphaBeta(successor(board,choice), alpha, beta, depth-1));
}
```

Fig. 8. The Probe algorithm

```
int PickSuccessor(Board board) {
   choice = 1;
   N = numSuccessors(board);
   if(N < 2) return (1)
   else {
      for(i = 1; i <= N; i++) {
         if(hasBestQuality(successor(board,i))) return (i);
         else if(hasGoodQuality(successor(board,i))) choice = i;
      }
   }
   return (choice);
}
```

Fig. 9. The PickSuccessor algorithm

chance nodes at depth=1 (the nodes at depth=3 are technically chance nodes, but since they are leaves, they are just statically evaluated), so probing was always relatively inexpensive.

For STAR2 to be successful, `Probe` must search a "good" child. We can abstract the selection process away from `Probe` and create another function, which we will call `PickSuccessor`. `PickSuccessor`, shown in Figure 9, will take a set of nodes and return the node it thinks is the "best". We want this selection process to be relatively fast and not use much overhead, so `PickSuccessor` may not want to use the evaluation function used for leaf evaluations, but instead use domain-specific knowledge to heuristically select a child. For example, in backgammon we may first select moves that result in hitting the opponent's blots, moves that form primes, or moves that form points. As soon as we see a successor that meets the best quality, we can simply exit with that successor as the choice. Failing that, we can keep track of a successor that has the next best quality. If no successors have either quality, then the first can just be chosen.

Even if we do not obtain a quick cutoff during the probing phase, we will have a tighter window for the search phase, which in itself will lead to quicker cutoffs, because we have better estimates of the values of the children. Reconsider once more the tree we have been using, but this time we will see what happens if Probe does a bad job. Figure 11 represents this situation. Assume that at the min nodes, we probe with the child that has the worst score for helping obtain a cutoff (the child with the maximum score). Now the probing phase will finish

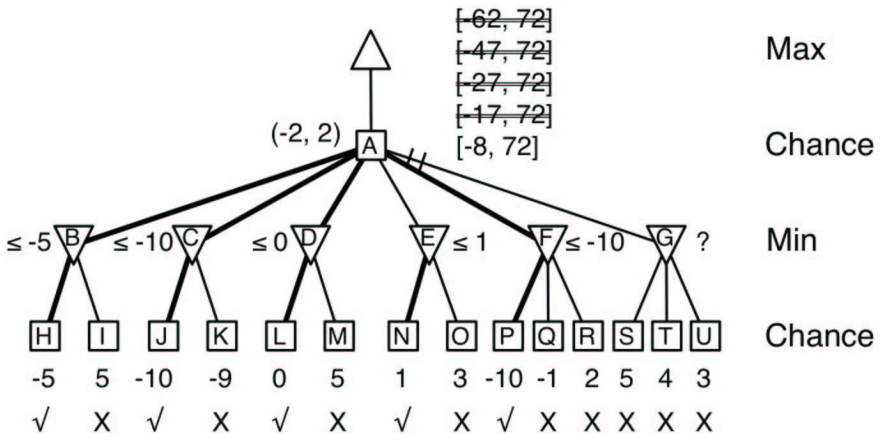

Fig. 10. A STAR2 tree, with good probing

before we have obtained a cutoff, and so we will end up searching almost half of the leaves already. However, before the searching phase begins, notice that the window has been almost halved, because we have better upper bounds for the childrens' values. Instead of starting with a window of [-62,62] as STAR1 would, we start searching sequentially with a window of [-8, 62]. Now, by the time we start to search the third successor D, we have passed it a window of [3, 10]. If we assume that the leaf node L is searched first, then we get a cutoff at D (because 0 is less than 3) as well as at A. We end up searching six leaves in the probing phase, and an additional three leaves in the search phase, for a total of nine leaves. In this particular situation, even the worst-case probing resulted in the same number of leaves expanded as STAR1.

2.5 Non-uniform Chance Event Probabilities

For many applications (including backgammon), the probability of each chance event is not uniform. The formulas used to derive the equations for A and B need to be modified to accommodate this generalization. Ballard mentions the modifications needed but does not go into detail [1]. Note that this process doesn't affect EXPECTIMAX, just STAR1 and STAR2.

Recall equation 1 for obtaining A_i. The entire left hand side of the inequality is divided by N because each of the N values has an equal chance of occurring. For non-uniform chance probabilities, this inequality changes to

$$(P_1 \times V_1 + \ldots + P_{i-1} \times V_{i-1}) + P_i \times V_i + U \times (1 - P_1 - \ldots - P_i) \leq alpha$$

or

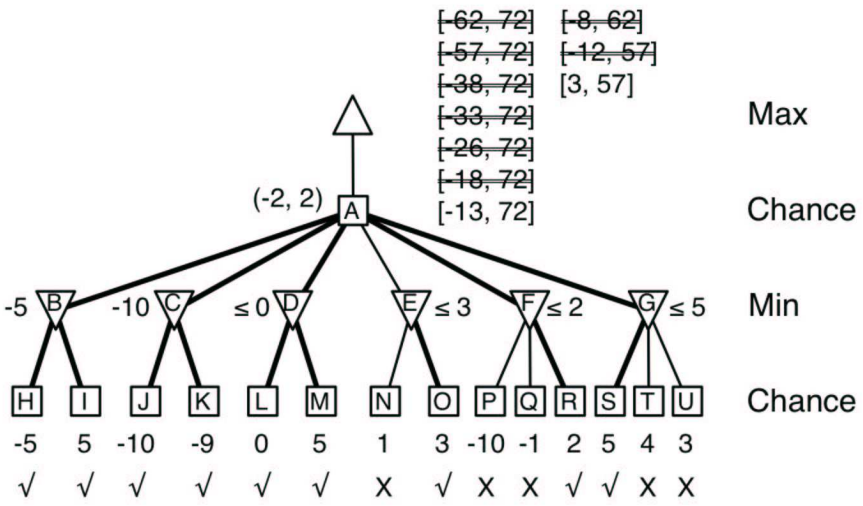

Fig. 11. A STAR2 tree, with bad probing

$$A_i = \frac{alpha - U \times (1 - P_1 - \ldots - P_i) - (P_1 \times V_1 + \ldots + P_{i-1} \times V_{i-1})}{P_i}$$

where P_i is the probability that the ith chance occurs, for A. Similarly,

$$B_i = \frac{beta - L \times (1 - P_1 - \ldots - P_i) - (P_1 \times V_1 + \ldots + P_{i-1} \times V_{i-1})}{P_i}$$

We will make the substitution $Y = (1 - P_1 - \ldots - P_i)$, which can be computed incrementally, where $Y_0 = 1$ and updates are made with $Y_i = Y_{i-1} - P_i$. We will make another substitution $X = (P_1 \times V_1 + \ldots + P_{i-1} \times V_{i-1})$, which can also be computed incrementally where $X_1 = 0$ and updates are made with $X_{i+1} = X_i + P_i \times V_i$. We can then calculate A_i and B_i with

$$A_i = \frac{(alpha - U \times Y_i - X_i)}{P_i}; \qquad B_i = \frac{(beta - L \times Y_i - X_i)}{P_i} \tag{4}$$

When there is only one successor, $A = alpha$ and $B = beta$, as desired.

These equations can be used for STAR1 and also for STAR2's probing phase. When calculating A and B values in STAR2's search phase, we can still use equation 4 to get A, but for B we need to modify equation 3:

$$B_i = \frac{(beta - W_i - X_i)}{P_i}$$

where $W_i = (W_{i+1} + \ldots + W_N)$, the sum of the probed values for nodes not yet searched.

2.6 Star2.5

Ballard proposed variations on the probing done by STAR2. For example, having probed one child of each node and not obtained a cutoff, additional probing effort could be invested. For example, each child could have a second probe performed. Ballard called the number of probes done at each node the *probing factor*. STAR1 can be viewed as having a probing factor of 0, while STAR2 has a probing factor of 1. Ballard proposed several algorithm variants with probing factors greater than 2, and called this family of algorithms STAR2.5.

2.7 Enhancements

ALPHA-BETA search has numerous enhancements that can greatly improve search efficiency. Here we briefly mention the enhancements that will have the most impact on *-MINIMAX algorithms:

- Transposition table. Besides the usual transpositions and move ordering benefits, transposition tables can help by re-using the results from STAR2's probing phase.
- Move ordering. Move ordering is always critical in any ALPHA-BETA-based search program. For STAR2, it is even more critical since it is needed to identify a "best" candidate for probing.
- Iterative deepening. Iterative deepening can be used to improve move ordering, both for the search and the probing.
- Fail soft. This is a simple enhancement that essentially comes for free (and has been added to the STAR1 and STAR2 pseudo-code). It helps narrow the search window bounds, resulting in earlier cutoffs.

These ideas have been implemented in a backgammon program and shown to be important enhancements to *-MINIMAX search [5].

3 Conclusions

Backgammon is the obvious domain for exploring performance issues of *-MINIMAX. The results are very encouraging, typically giving an extra ply or two of search. These results are reported in a companion article [5].

Besides games, the *-MINIMAX algorithms seem to also be applicable to Markov Decision Processes (MDPs), especially in the area of multi-agent MDPs. While solving MDPs usually involves an EXPECTIMAX-type evaluation of states one step away during value iteration, perhaps that component could be changed to a depth-N search of states, where the action at any given state would be determined by the current policy at that iteration. This may produce quicker convergence, or in the case of multi-agent MDPs, a better method for choosing actions that lead to higher rewards.

A general approach to solving games that combine elements of skill and chance will remain an open research problem for a while to come, but they provide some of the most interesting domains as they often have elements at which

computers excel but humans do not (optimization, uncertainty calculation), and vice-versa (long-term planning, opponent modeling). Games that combine skill, chance, imperfect information and opponent interaction are the most difficult domains for computers, so cross-disciplinary approaches involving combining elements of heuristic search, machine learning, agent theory, game theory and even psychology may prove the most fruitful in the years to come.

Acknowledgments

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. Bruce Ballard. The *-Minimax search procedure for trees containing chance nodes. *Artificial Intelligence*, 21(3):327–350, 1983.
2. Darse Billings, Aaron Davidson, Jonathan Schaeffer, and Duane Szafron. The challenge of poker. *Artificial Intelligence*, 134(1–2):201–240, 2002.
3. Matt Ginsberg. GIB: Steps toward an expert-level bridge-playing program. In *International Joint Conference on Artificial Intelligence*, pages 584–589, 1999.
4. Thomas Hauk. Search in trees with chance nodes. Master's thesis, Computing Science, University of Alberta, 2004.
5. Thomas Hauk, Michael Buro, and Jonathan Schaeffer. *-Minimax performance in backgammon. In *Computers and Games*, 2004. Under submission.
6. Donald 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.
7. Brian Sheppard. *Towards Perfect Play of Scrabble*. PhD thesis, Computer Science, Universiteit Maastricht, 2002.
8. Gerald Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995.