Effective Use of Transposition Tables in Stochastic Game Tree Search

Joel Veness†‡
Joel.Veness@nicta.com.au

Alan Blair†‡
blair@cse.unsw.edu.au

†National ICT Australia
‡School of Computer Science and Engineering
University of New South Wales, Australia

Abstract — Transposition tables are one common method to improve an alpha-beta searcher. We present two methods for extending the usage of transposition tables to chance nodes during stochastic game tree search. Empirical results show that these techniques can reduce the search effort of Ballard’s Star2 algorithm by 37 percent.

Keywords: Transposition Table, Stochastic Game Tree Search, Pruning, Expectimax, Star1, Star2

I. INTRODUCTION

Decades of research into the alpha-beta algorithm have resulted in many enhancements which dramatically improve the efficiency of two-player deterministic game tree search. In contrast, stochastic game tree search has received considerably less attention. Hauk et al [1] recently reintroduced an updated version of Ballard’s almost forgotten Star2 algorithm [2], showing how the fail-soft alpha-beta enhancement can be adapted to chance nodes, and extending the algorithm to games with non-uniform chance events. In the present work, we further explore how existing alpha-beta enhancements can be adapted to improve the performance of stochastic game tree search.

One important enhancement - widely used in Checkers and Chess programs - is a transposition table, which is employed to improve the move ordering of alpha-beta searchers and to allow the algorithm to reuse previous search work. While transposition tables have been used at min and max nodes of a search tree [3,4], their usage at chance nodes has not previously been addressed in the literature.

In the present work, we show in detail how the transposition table improvement can be extended to chance nodes. We introduce two complementary methods that use the information present in the transposition table to reduce the overall search effort at chance nodes.

II. STAR1 AND STAR2

Expectimax is a brute force, depth first game tree search algorithm that generalizes the minimax concept to games of chance, by adding a chance node type to the game tree. Star1 and Star2 exploit a bounded heuristic evaluation function to generalize the alpha-beta pruning technique to chance nodes [2]. Alpha-beta pruning imposes a search window $[\alpha, \beta]$ at each min or max node in the game tree. The search through successor nodes can be terminated as soon as the current node’s value is proven to fall outside the search window. Star1 and Star2 generalize this pruning idea to chance nodes, but an important distinction needs to be borne in mind. At min and max nodes, a single successor is sufficient to terminate the search. However, at chance nodes it is necessary to prove that the weighted sum of all successors will fall outside the search window. Star1 achieves this by starting with a wide search window and narrowing it for each successive node, to a range that would be sufficient to terminate the search, even assuming worst-case values for the remaining unsearched nodes. The narrowed search windows for each successor allow us to prune more heavily at nodes further down the game tree.

Figure 2.1 illustrates the operation of the Star1 algorithm at a chance node with three equally likely successors. We assume that the values returned by the heuristic evaluation function always lie in the range [-1.0, +1.0]. If the search window of the chance (root) node is $[-0.4, +0.1]$, then we are only interested in obtaining an exact score if it lies within this interval. If the score falls outside this window, we only need to establish an upper or lower bound. The intervals shown under each successor indicate the Star1 search window for that node. Once the left hand node has been computed to have an exact value of 0.8, any value $\geq 0.5$ for the middle node will imply that the expectimax value for the chance node is $\geq 0.1$, thus allowing the search to be cut off without exploring the right hand node.
An example cutoff leaving us with upper and lower bounds for algorithms will stop searching the remaining successors, the expectimax score gained from the already searched successors has narrowed since we know that these bounds.

This new scheme allows us to store the usual upper, lower or exact bound information on the expectimax value of a node when we capture the information that is present at chance nodes. This is an upper, lower or exact bound. This is insufficient to fully establish a lower bound on the score of each successor. We can apply these same techniques at chance nodes, provided we have a suitable representation of the information present at chance nodes.

A standard transposition table implementation contains two procedures - store and retrieve. Typical transposition table implementations for alpha-beta based searchers store only a single score, and a flag to indicate whether this score is an upper, lower or exact bound. This is insufficient to fully capture the information that is present at chance nodes. This is because we can have both upper and lower bound information on the expectimax value of a node when we terminate the search of its successors. Figure 3.1 shows a graphical depiction of this situation.

Figure 3.1 – An example cutoff situation.

At a node $n$, we have searched some number of successors. The search window for this node is $[\alpha, \beta]$. The information gained from the already searched successors has narrowed the expectimax score $x$ for this node to the interval $[a, b]$. Since we know that $x$ must be less than $\alpha$, the Star1/Star2 algorithms will stop searching the remaining successors, leaving us with upper and lower bounds for $x$. So that we do not lose information about node $n$, we need to store both of these bounds.

We can solve this problem by modifying the store operation to record both an upper and lower bound on the score for each position, along with their associated depths. This new scheme allows us to store the usual upper, lower or exact bound information at min/max nodes, whilst letting us store independent upper and lower bounds at chance nodes. The retrieve procedure simply returns all of the information about a node from the transposition table.

These two modified routines are sufficient to implement the transposition table cutoff idea. The associated modifications to the Star2 algorithm are described in section IV.

B. Successor Probing

The Enhanced Transposition Cutoff idea augments a normal alpha-beta search with a preliminary transposition table probing phase [5]. The transposition table is probed for each successor. If any one of these entries has enough information for us to exceed beta, we can terminate the search immediately.

We can augment the Star2 algorithm with a similar probing phase. The details for the stochastic case are more complicated, since any bound information we get will be useful, even if it doesn’t allow us to directly cutoff.

The algorithm is implemented as follows. If we are at a chance node $n$ that we are trying to search to depth $d$, we look for all of the information contained in the transposition table for each successor that is based on at least a depth $d-1$ search. Any information that the entry contains is either used to immediately prove that the expectimax score is outside the search window for this node, or stored and then used to tighten the successor search bounds during the subsequent Star2 and Star1 parts of the search.

Figure 3.2 gives an example where this additional probing enhancement can help in practice. Suppose we are at the root node, with a search window of $[\alpha, \beta]$. Since we are using a transposition table, we have a cache of previous search results. Suppose that there exists an entry in the transposition table for the third successor, and that this entry contains the exact score for the successor node. If the expectimax value of the root node is going to be less than $alpha$, then we would like to prove this as early as possible. Retrieving
information from the table is much less expensive than doing a full search, and this information from the transposition table could save us from having to search some successors. The probing enhancement ensures that we retrieve all useful information from our transposition table before searching any successors.

In practice, the probing enhancement could be relatively computationally expensive. A simple way around this problem is to limit the probing enhancement to chance nodes with sufficient depth. This limit will of course be domain and implementation dependent.

IV. IMPLEMENTATION

Figure 4.1 presents pseudo-code for the transposition table enhanced Star2 algorithm, which we shall denote Star2-TT. The additions required to employ transposition cutoffs at chance nodes are shown in bold. The changes that are also italicized constitute the probing phase used to implement our stochastic analogue of the Enhanced Transposition Cutoff idea.

The negamax function called in the Star1 component of the code refers to a negamax formulation of an enhanced alpha-beta search. The negamax-probe function called in the Star2 component is the same as negamax, but with an additional ProbingFactor argument which limits the number of moves searched at the root. Because of this limiting factor, the score returned and stored by negamax-probe is only a lower bound.

We use the constants $U$ and $L$ to denote the absolute upper and lower bounds on the scores returned by the heuristic evaluation function. In our implementation, $U=+1.0$ and $L=-1.0$.

It is important to distinguish how we compute the search window for each successor, compared to the pseudo-code given by Ballard [2] and Hauk [1]: They are able to compute the window incrementally, with a simple update rule. We cannot use such a rule because we can never be sure of what successor information will be determined from the transposition table before we begin searching the successors.

At the beginning of Star2-TT, we initialize an array that stores our current bound information for each chance event. We denote this array as $cinfo$ in the pseudo-code. Each element in the array contains an upper bound, a lower bound and the probability of that particular chance event occurring. The upper bound for each successor is initially set to $U$, whilst the lower bound is set to $L$. As we gain more information about the successors throughout the various stages of the algorithm, the corresponding entry in the array is updated. At any time, we can compute the lower and upper bounds, $LB(n)$ and $UB(n)$, on the current expectimax value of a chance node $n$ with $N$ successors by computing:

$$LB(n) = \sum_{i=0}^{N-1} P_i(n) \times LB_i(n)$$

### Figure 4.1 Star2-TT Pseudo-code.

```plaintext
Score star2-TT(Board brd, Score alpha, Score beta, int depth) {
    if (isTerminalPosition(brd)) return TerminalScore(brd)
    if (depth == 0) return evaluate(brd)
    EventInfo cinfo[numChanceEvents()] // for successor bounds
generateChanceEvents()
    // transposition cutoffs
    if (retrieve(board) == Success) {
        if (entry.ubound == entry.lboud & &
            entry.udepth == entry.ldepth &&
            entry.udepth >= depth) return entry.ubound
    }
    if (entry.ldepth >= depth) {
        if (entry.lboud >= beta) return entry.lboud
        alpha = max(alpha, entry.lboud)
    }
    if (entry.udepth >= depth) {
        if (entry.ubound <= alpha) return entry.ubound
        beta = min(beta, entry.ubound)
    }
    // successor probing using the transposition table
    for (i = 0; i < numChanceEvents(); i++) {
        applyChanceEvent(brd, i)
        if (retrieve(brd) == Success) {
            cinfo[i].LowerBound = entry.lboud
            if (LB(cinfo) == beta) {
                store(brd, depth, LB(cinfo), UB(cinfo))
                return LB(cinfo)
            }
            if (entry.udepth >= depth-1) {
                cinfo[i].UpperBound = entry.ubound
                if (UB(cinfo) == alpha) {
                    store(brd, depth, LB(cinfo), UB(cinfo))
                    return UB(cinfo)
                }
                }\undoChanceEvent(brd, i)
        }
        // Star2 probing phase
        for (i = 0; i < numChanceEvents(); i++) {
            Score cmax = childMax(cinfo[i], beta, i)
            Score bx = min(U, cmax)
            applyChanceEvent(brd, i)
            Score search_val = negamax-probe(brd,
                cinfo[i].LowerBound, bx, depth-1, ProbingFactor)
            undoChanceEvent(brd, i)
            cinfo[i].LowerBound = max(cinfo[i].LowerBound, search_val)
            if (search_val >= cmax) {
                store(brd, depth, LB(cinfo), UB(cinfo))
                return LB(cinfo)
            }
            // Star1 phase
            if (i == 0; i < numChanceEvents(); i++) {
                Score cmin = childMin(event_info, alpha, i)
                Score bx = min(U, cmax)
                applyChanceEvent(brd, i)
                Score search_val = negamax(brd, ax, bx, depth-1)
                undoChanceEvent(brd, i)
                cinfo[i].UpperBound = search_val
                if (search_val <= cmin) {
                    store(brd, depth, LB(cinfo), UB(cinfo))
                    return UB(cinfo)
                }\undoChanceEvent(brd, i)
            }
        }
    }
```
Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Games (CIG 2007)

\[ UB(n) = \sum_{i=0}^{N-1} P_i(n) \times UB_i(n) \]

where \( P_i(n) \) denotes the probability of the \( i \)th chance event, and \( LB_i(n)/UB_i(n) \) denotes our current lower/upper bounds on the expectimax value of the \( i \)th successor.

We know the exact expectimax value of a node \( n \) if \( LB(n) = UB(n) \). A straightforward implementation optimization, that could be worthwhile if the number of chance events is large, is to incrementally maintain the \( LB(n) \) and \( UB(n) \) quantities.

Whenever we recursively call the negamax procedure from a chance node \( n \) with a search window of \([\alpha, \beta] \), we derive the search window from our current successor information. This can be done by computing:

\[
\begin{align*}
\text{ChildMin}(n,i) &= \alpha - UB(n) + P_i(n) \times UB_i(n) \\
\text{ChildMax}(n,i) &= \beta - LB(n) + P_i(n) \times LB_i(n)
\end{align*}
\]

Thus the \( i \)th successor has a search window of \([\text{Max}(L, \text{ChildMin}(n,i)), \text{Min}(U, \text{ChildMax}(n,i))] \).

These bounds are chosen so that if the negamax search call returns a value in excess of \( \text{ChildMax}(n,i) \) or less than \( \text{ChildMin}(n,i) \), then:

\[ S(n) \geq \beta \text{ or } S(n) \leq \alpha \]

respectively, where \( S(n) \) denotes the expectimax value of node \( n \). For a derivation of these equations, see Hauk [1].

It is important to note that during the Star2 probing phase, only the \( \text{ChildMax} \) routine is used in computing each successor’s search window. This is because the \textit{negamax-probe} procedure can only return lower bounds on the expectimax score of each successor.

V. EXPERIMENTAL SETUP

Our experimental framework used the game of Dice created by Hauk et al [3]. Dice is a two player stochastic game where players take turns placing checkers on an \( m \) by \( m \) grid. Before each move, a die is rolled to determine the row or column into which the checker must be placed. The winner is the first player to achieve a connected “run” of \( k \) checkers (horizontally, vertically or diagonally). In our experiments, \( m=5 \) and \( k=4 \). The game of Dice is a prime candidate for the transposition table improvement, since many different lines of play can lead to the same position.

For deterministic game tree search, any monotonic transformation of the evaluation function will result in the same line of best play. However, for stochastic game tree search, the evaluation function must estimate a value directly proportional to the expected final reward (in our case, +1 for win, -1 for loss, 0 for draw). With this in mind, a 2-layer feed-forward neural network with 10 hidden nodes was trained by self play in the same manner as Tesauro’s TD-Gammon [6]. The network had 50 inputs for the raw board encoding plus 4 inputs to store the number of “runs” of length 2 and 3 for each player.

Our search algorithm incorporated several well known move ordering techniques at min/max nodes. We used a combination of transposition tables, the killer move heuristic and the history heuristic in conjunction with iterative deepening [7]. These move ordering techniques increase the efficiency of the alpha-beta component and the Star2 probing phase.

We used a Zobrist hash function [8] to map the Dice game states to entries in the transposition table. The results were generated with a 2.6 gigahertz AMD AthlonFX CPU, using only one processor. 256Mb of memory were used for the transposition table.

VI. RESULTS

We compared Expectimax, Star1, Star2 and Star2-TT. Each of these four algorithms was evaluated on the same set of 50 test positions collected from self-play games. These positions include a representative range of opening, middle and end-game positions.

The results are summarized in Tables I and II:

<table>
<thead>
<tr>
<th>TABLE I</th>
<th>\text{NUMBER OF NODES SEARCHED TO DEPTH 13 (MILLIONS)}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Expectimax</td>
</tr>
<tr>
<td>Min</td>
<td>0.02</td>
</tr>
<tr>
<td>Median</td>
<td>13.93</td>
</tr>
<tr>
<td>Max</td>
<td>273.8</td>
</tr>
<tr>
<td>Mean</td>
<td>37.78</td>
</tr>
<tr>
<td>Std</td>
<td>54.67</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TABLE II</th>
<th>\text{TOTAL TIME FOR SEARCH TO DEPTH 13 (SECONDS)}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Expectimax</td>
</tr>
<tr>
<td>Min</td>
<td>0.44</td>
</tr>
<tr>
<td>Median</td>
<td>211</td>
</tr>
<tr>
<td>Max</td>
<td>4669</td>
</tr>
<tr>
<td>Mean</td>
<td>609.1</td>
</tr>
<tr>
<td>Std</td>
<td>913.5</td>
</tr>
</tbody>
</table>
VIII. Future Work

A. Parallel Search

It is possible to parallelize the negamax algorithm [9]. It would be interesting to investigate methods of splitting up the search work at chance nodes, since the risk of performing redundant work at chance nodes might be much lower than for min/max nodes.

B. Search Window Adjustments

At some nodes, we might strongly suspect that we are going to fall outside the search window for the current node. In these cases, it could prove worthwhile to try to use narrow window searches to prove that the node’s evaluation falls outside the current search window. This has the potential to save search effort because in general it is easier to prove that a successor lies above or below some bound than it is to find its exact score. If we can skip finding exact scores at some nodes, we expect a performance improvement so long as we can reliably determine where to use these narrow window searches.

IX. Acknowledgements

National ICT Australia is funded by the Australian Government’s Department of Communications, Information Technology, and the Arts and the Australian Research Council through Backing Australia’s Ability and the ICT Research Centre of Excellence programs.

References


VII. Conclusion

We have described how to effectively use transposition tables at chance nodes in two-player stochastic game tree search. This addresses an important gap in the stochastic game tree search literature. We have shown that these procedures, in combination with the Star2 algorithm, give a 37% reduction in nodes searched over plain Star2 for the game of Dice at a search depth of 13. These techniques will be useful for stochastic game tree searchers when applied to games where many different lines of play can lead to the same board state.