Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Games (CIG 2007)

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 re-introduced 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.

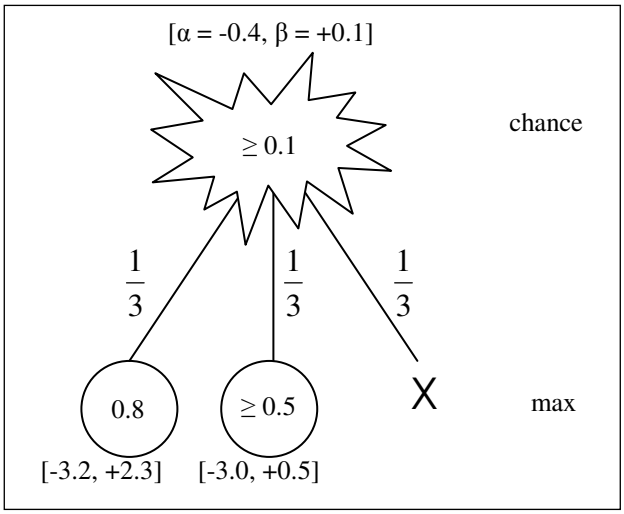

Figure 2.1 – Star1 Example.

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 node 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 ≥0.5 for the middle node will imply that the expectimax value for the chance node is ≥0.1, thus allowing the search to be cut off without exploring the right hand node.

If the stochastic game tree is regular, the Star2 algorithm can be used [2]. Star2 further enhances the Star1 algorithm by augmenting it with a preliminary probing phase that searches one child of each successor node. This cheaply establishes a lower bound on the score of each successor. These lower bounds can be used to either prove that the node's score falls outside the search window, or to further narrow the search windows of each successor for the Star1 part of the search.

Detailed analysis of Expectimax, Star1 and Star2 is given by Ballard in [2] and by Hauk et al in [3] and [1].

III. Transposition Table Usage at Chance Nodes

A. Transposition Cutoffs for Chance nodes

In many games, there are multiple paths to the same node in the game tree. In these games, it is possible to save search effort by caching previous search results. This information can allow us to either directly return a score, or narrow the search window before searching any successors of a node. 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 [α,β]. 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 α, 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.

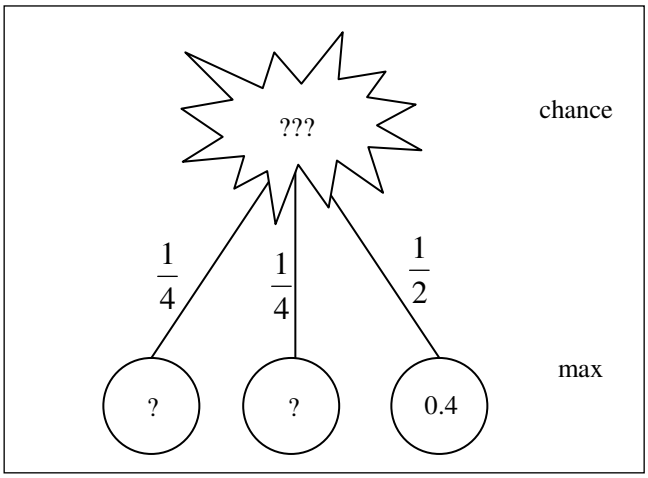

Figure 3.2 – Illustration of Probing Enhancement.

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 [α,β]. 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:

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

```
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.lbound &&
       entry.udepth == entry.ldepth &&
       entry.udepth >= depth) return entry.ubound
   if (entry.ldepth >= depth) {
     if (entry.lbound >= beta) return entry.lbound
     alpha = max(alpha, entry.lbound)
   }
   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) {
     if (entry.ldepth >= depth-1) {
       cinfo[i].LowerBound = entry.lbound
       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, 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
 for (i = 0; i < numChanceEvents(); i++) {
   Score cmin = childMin(event_info, alpha, i)
   Score cmax = childMax(event_info, beta, i)
   Score ax = max(L, cmin)
   Score bx = min(U, cmax)
   applyChanceEvent(brd, i)
   Score search_val = negamax(brd, ax, bx, depth-1)
   undoChanceEvent(brd, i)
   cinfo[i].LowerBound = search_val
   cinfo[i].UpperBound = search_val
   if (search_val >= cmax) {
     store(brd, depth, LB(cinfo), UB(cinfo))
     return LB(cinfo)
   }
   if (search_val <= cmin) {
     store(brd, depth, LB(cinfo), UB(cinfo))
     return UB(cinfo)
   }
 }

 store(brd, depth, LB(cinfo), UB(cinfo))
 return LB(cinfo)  // LB(cinfo) == UB(cinfo) here
}
```

Figure 4.1 Star2-TT Pseudo-code.

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

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 [α,β], we derive the search window from our current successor information. This can be done by computing:

$$ChildMin(n,i) = \frac{\alpha - UB(n) + P_i(n) \times UB_i(n)}{P_i(n)}$$

$$ChildMax(n,i) = \frac{\beta - LB(n) + P_i(n) \times LB_i(n)}{P_i(n)}$$

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

These bounds are chosen so that if the negamax search call returns a value in excess of *ChildMax(n,i)* or less than *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 *ChildMax* routine is used in computing each successor's search window. This is because the *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 I

NUMBER OF NODES SEARCHED TO DEPTH 13 (MILLIONS)

	Expectimax	Star1	Star2	Star2-TT
Min	0.02	0.01	0.01	0.01
Median	13.93	3.99	3.24	1.79
Max	273.8	136.7	31.69	22.81
Mean	37.78	14.76	6.12	3.85
Std	54.67	24.38	7.01	4.70

TABLE II

TOTAL TIME FOR SEARCH TO DEPTH 13 (SECONDS)

	Expectimax	Star1	Star2	Star2-TT
Min	0.44	0.27	0.31	0.27
Median	211	62.25	45.14	28.26
Max	4669	2326	487.8	374.1
Mean	609.1	238.1	88.55	60.83
Std	913.5	408.3	104.6	75.9

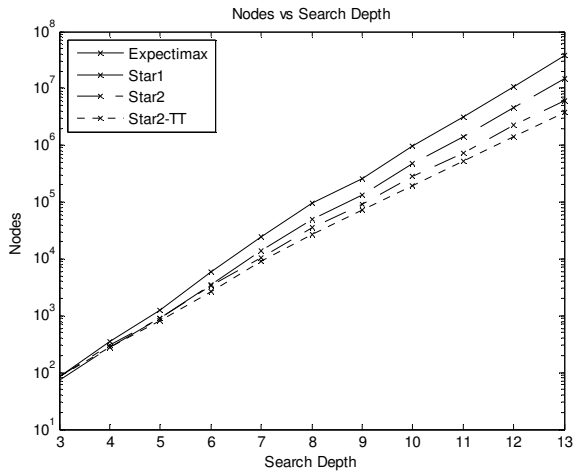

Figure 6.3 – Total nodes searched for each depth.

Figure 6.3 shows the average number of nodes searched over our test positions, as the depth is increased.

Star2-TT shows on average a 37% improvement over Star2 using depth 13 searches, in terms of nodes searched. This is a significant performance improvement, and well worth the implementation hassle for any competitive stochastic game playing program.

Although it is theoretically possible in degenerate cases for Star2 and Star2-TT to search more nodes than Star1, in practice this does not happen. The savings given by the extra probing phase more than outweigh the associated overhead.

The performance results on the time to depth metric are strongly correlated with the number of nodes searched. This is because the leaf node evaluation function takes up almost all of the CPU time. Reducing the number of nodes in the game tree means that fewer calls to the evaluation function need to be made. The overhead associated with using the transposition table is worthwhile. On average, it takes 31% less time for Star2-TT to reach depth 13 compared to Star2.

Although our results generally agree with those found by Hauk in [3], we noticed a larger difference between the performance of Star1 and Expectimax. This could be due to our evaluation being constrained to return scores from a different interval, or due to differences in the alpha-beta implementation.

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.

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

[1] T. Hauk, M. Buro, and J. Schaeffer. Rediscovering *-Minimax. *Computers and Games conference*, 2004.

[2] Bruce W. Ballard. The *-minimax search procedure for trees containing chance nodes. *Artificial Intelligence*, **21**:327-350, 1983.

[3] T. Hauk. *Search in Trees with Chance Nodes*. M.Sc. Thesis, Computing Science Department, University of Alberta, 2004.

[4] T. Hauk, M. Buro, and J. Schaeffer. *-Minimax performance in backgammon. *Computers and Games*, 2004.

[5] Jonathan Schaeffer and Aske Plaat. New advances in alpha-beta searching. In *Proceedings of the 24th ACM Computer Science Conference*, pages 124-130, 1996.

[6] G. Tesauro. Temporal Difference Learning and TD-Gammon. *Communications of the ACM*, **38**(3), March 1995.

[7] J. Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **11**(1):1203-1212, 1989.

[8] A. L. Zobrist. *A new hashing method with applications for game playing*. Technical Report 88, University of Wisconsin, 1970.

[9] R. Hyatt, *A High-Performance Parallel Algorithm to Search Depth-First Game Trees*, Ph.D. Dissertation, University of Alabama at Birmingham, 1988.