THE UNIVERSITY OF NEW SOUTH WALES
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Expectimax Enhancements for Stochastic Game Players

Joel Veness

Thesis submitted as a requirement for the degree:

Bachelor of Science (Computer Science) Honours

Submitted: October 30, 2006

Supervisor: Alan Blair

Acknowledgements

My supervisor Alan Blair for all of his time, advice and explanations. It has been fun and I have learnt a lot.

Kee Siong and Will Uther for their advice and help in proof reading.

Felicity Allen for her advice, proof reading, encouragement and support.

Abstract

This thesis develops a state of the art stochastic game tree searcher. It incorporates existing alpha-beta enhancements in conjunction with Ballard's Star1 and Star2 stochastic search algorithms.

Particular attention is paid to the construction of a suitable heuristic evaluation function. We show how the addition of stochasticity considerably complicates this process compared to the deterministic case.

We have developed an experimental framework based around the game of Dice. The framework includes four different stochastic search algorithms incorporating an enhanced alpha-beta search, a $TD(\lambda = 0)$ trained feed-forward neural network for the evaluation function, as well as a graphical interface that allows for batch processing of Dice positions.

We extend the existing literature by introducing the StarETC algorithm. StarETC is a novel enhancement of the Star2 algorithm. It exploits a transposition table at chance nodes to reduce search effort. StarETC is shown to take 37% less search effort to compute the same results as Star2 for depth 13 searches in the game of Dice.

Contents

Chapter 1

Introduction

This thesis investigates how computers can play two-player games of chance. We focus on games where the entire state of the game is visible to both players. This property is known under the more technical name of *perfect information*. The element of chance means that we are considering *stochastic* domains. Some popular two-player, stochastic, perfect information games include Backgammon and Carcassonne. Backgammon is stochastic due to the dice rolls, and Carcassonne is stochastic due to the random drawing of game tiles. Both are perfect information games since the game state is completely visible at all times.

For many games, it is possible to construct a heuristic *evaluation function*. This function takes a game state and returns a score that reflects the utility of being in that state. These heuristic functions are typically correct at terminal game states, but are only estimates for the non-terminal game states. Coupled with a move generator, a heuristic evaluation function can be used by the computer to determine what move to make in a given position.

However, for some games it is very difficult to construct a good heuristic evaluation function by only looking at static game state information. One way to improve the quality of a heuristic evaluation function is by using a *game tree search* algorithm. Game tree search algorithms work by looking at many possible future game states. These future states can be visualised as trees of possibilities. Figures 1.1 and 1.2 show two different types of game trees. Figure 1.1 shows a two-player deterministic game tree, that could be used in a game such as Chess or Checkers. Figure

1.2 shows a two-player stochastic game tree, that could be used in a game such as Backgammon or Carcassonne. This thesis is focused on efficiently searching stochastic game trees.

The *minimax* algorithm is a well known method for playing two-player deterministic games. It has been studied extensively and has many enhancements. We will go through minimax and its enhancements in detail in the next chapter.

Whilst the minimax algorithm has been extensively studied, the natural extension of it to stochastic games, the *expectimax* algorithm, has not. We will be reviewing Ballard's expectimax improvements and later on introducing an improved algorithm for stochastic game tree search.

The layout of this thesis proceeds from the theoretical to the practical. We begin by surveying the previous work done in two player game tree search, and introduce Ballard's improved expectimax based algorithms. We then introduce our improvements to expectimax based searching. The rest of the thesis is spent introducing our experimental framework, which we use to justify our expectimax enhancements.

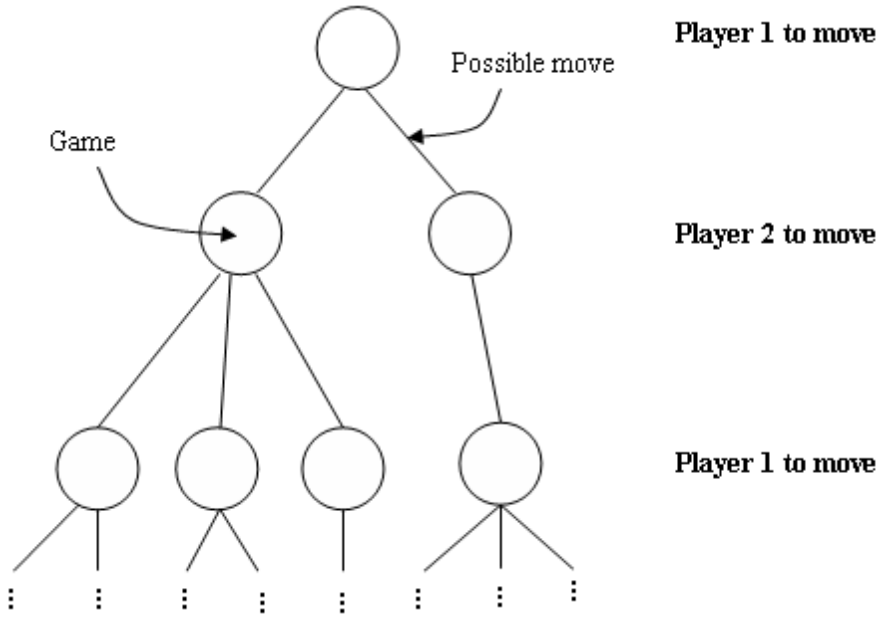


Figure 1.1: A two-player deterministic game tree

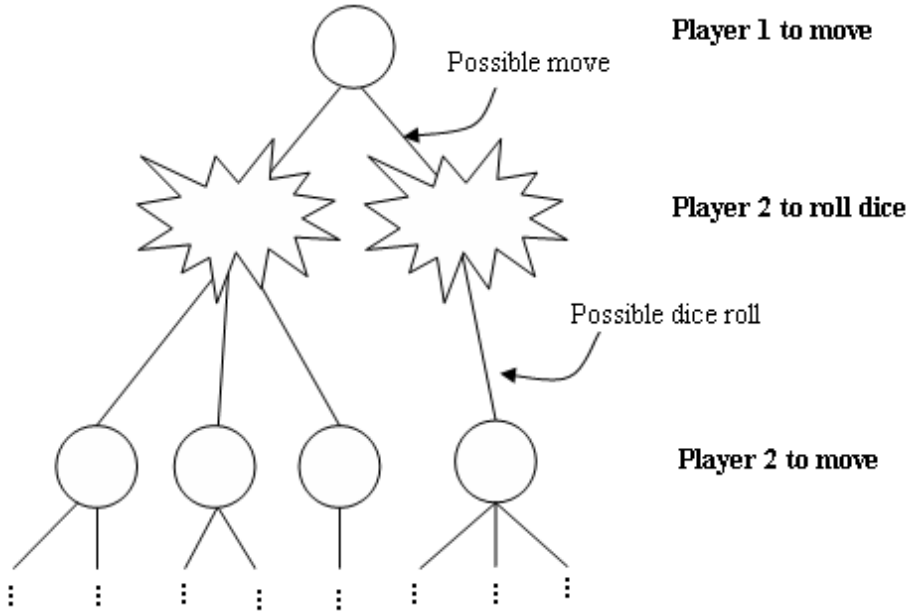


Figure 1.2: A two-player stochastic game tree

Chapter 2

Background

In this chapter, we give an overview of two-player game tree searching. We assume that the reader has been previously exposed to the minimax and alpha-beta algorithms.

We begin by giving a brief introduction to two-player deterministic search, and introduce the terminology associated with it. We then go on to describe some well-known extensions to the alpha-beta algorithm.

We then introduce the expectimax algorithm. Expectimax is a generalisation of the minimax algorithm that is applicable to games of chance. Ballard's enhancements to expectimax are then introduced. These enhancements allow for alpha-beta style cutoffs in stochastic game tree search.

2.1 Minimax

Minimax is a recursive game tree search algorithm. It computes the value of a state in a two-player deterministic game. It does this by constructing a tree of possible player actions and opponent responses. For simple games, minimax will correctly compute the best move in any situation by searching the entire game tree. However, most games are too complicated to search the entire game tree, so typically we limit the number of levels of recursion to some fixed depth.

To determine the value of the game tree, every node is traversed in a depth first fashion. At the leaf nodes we apply a heuristic evaluation function. The values from this heuristic evaluation function are then backed up to the nodes closer to the root, using the following process:

The player to move is known as the max player, and the opponent is the min player. As we progress from the root to the leaves, we alternate between decisions that need to be made by the max and min players respectively. We assume that at any position, each player will perform the action that maximises their utility. This means that the max player will perform the action that maximises their score, and the min player will perform the action that minimises the score for max player.

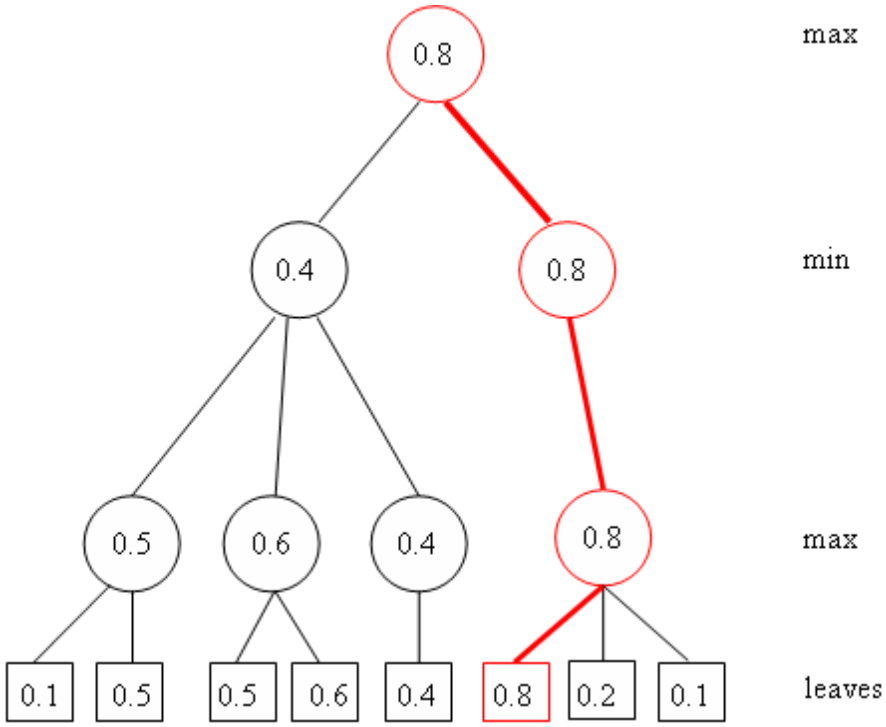


Figure 2.1: A minimax search tree. The line of best play is shown in red.

The minimax algorithm is a computationally expensive process. The *branching factor* of a game is the average number of possible moves at a game state. If the branching factor is b, and we wish to perform a search to depth d, then the runtime complexity of minimax is $O(b^d)$.

Some people argue that the minimax assumption is suboptimal, because it assumes the opponent is perfect and doesn't take advantage of the opponent's propensity to make mistakes. This is a reasonable objection, but in practice minimax based searchers perform well in domains such as Chess and Checkers.

2.1.1 Pseudocode

The pseudocode for minimax is given below. Notice how we have two separate inner loops, one for the max player and one for the min player. Later on, we will show how to rewrite this procedure without having to handle each case separately.

```
Score minimax(Board board, int depth) {

    if (isTerminal()) return terminalScore()
    if (depth == 0) return evaluate(board)

    generateMoves()

    if (maxNode()) {
        Score best = -Infinity
        for (i = 0; i < numMoves(); i++) {
            board.makeMove(move(i))
            val = minimax(board, depth-1)
            board.unmakeMove()

            if (val > best) best = val
        }
    } else if (minNode()) {
        Score best = Infinity
        for (i = 0; i < numMoves(); i++) {
            board.makeMove(move(i))
            val = minimax(board, depth-1)
            board.unmakeMove()
            if (val < best) best = val
        }
    }

    return best
}
```

2.2 Alpha-Beta Pruning

Alpha-beta pruning is a well known way to drastically reduce the branching factor of a minimax search.

At any node during an alpha-beta search, we have something called a search window. This denotes the interval $[\alpha, \beta]$ of scores we are interested in. Using alpha-beta pruning allows us to compute the exact minimax score of a node, so long as this score falls within the search window. If the true value of the node lies outside the search window, then the alpha-beta algorithm will only give us bound information.

Thus, the three types of information we can gather from an alpha-beta search are:

1. a *lower bound* - the score is at least this value
2. an *upper bound* - the score is at most this value
3. an *exact value* - the score is exactly this value

As the tree is searched, the window is narrowed, which allows us to skip searching many sub-branches of the game tree. How this bound is narrowed depends on the order the successors of a node are searched. We achieve the best case performance of alpha beta pruning when the best successor branch at any node is searched first. In alpha-beta based game playing programs, a lot of effort is spent developing heuristics that allow one to search better looking moves earlier. This sorting process typically comes under the general term of *move ordering*.

With optimal move ordering, alpha-beta pruning reduces the asymptotic complexity of the minimax search procedure to $O(b^{d/2})$ [6]. However, the worst case performance of alpha-beta is the same as minimax. In practice, near optimal move ordering can be achieved by using a combination of transposition tables, killer moves tables, the history heuristic, iterative deepening and domain specific heuristics. These alpha-beta enhancements will be explained in detail shortly.

2.2.1 Negamax Formulation of Alpha-Beta Pruning

If we construct our evaluation function so that it always returns a score with respect to the side to move, then we can reformulate the minimax algorithm with alpha-beta pruning in a way that allows us to avoid having to handle the min and max cases separately. This is known as the *negamax* formulation of alpha-beta pruning.

Pseudocode

The negamax formulation of the alpha-beta algorithm is given below. If we are searching node with a search window of $[\alpha, \beta]$, then each successor is searched with a window of $[-\beta, -\alpha]$. Before we can use the result of the recursive call, we need to negate the score so that it reflects the utility of the current side to move.

```
Score negamax(Board board, Score alpha, Score beta, int depth) {

    if (isTerminal()) return terminalScore()
    if (depth == 0) return evaluate(board)

    generateMoves()
    applyMoveOrderingHeuristics()

    for (i = 0; i < numMoves(); i++) {
        board.makeMove(move(i))
        val = -negamax(board, -beta, -alpha, depth-1)
        board.unmakeMove()
        if (val > alpha) {
            if (val >= beta) return beta
            alpha = val
        }
    }

    return alpha
}
```

2.2.2 Terminology

We now revise some terminology that will make our life easier when explaining the more complicated algorithms.

Failing High / Failing Low

Whilst searching a node with a window of $[\alpha, \beta]$, we often say that a node has *failed high*. This means that we have proven that the score for the node is at least β. Similarly, *failing low* means that the score for a node is at most α. We often make use of the term *cutoff*, which means to either fail low or fail high.

Narrowing the Search Window

This refers to modifying the search window by bringing the bounds closer together. Many different alpha-beta enhancements rely on narrowing the search window for a reduction of search effort.

Zero-width Window Search

This refers to a search window that can never return an exact score because the search window is too narrow. If our evaluation function returns only integers, an example zero-width window is $[\alpha, \alpha + 1]$ for some α.

Depth / Ply

This refers to how many moves ahead the alpha-beta procedure is looking, relative to the current node. A ply refers to one level of search work. When we say that we are performing a k ply search, we mean that the root node of our alpha-beta search has a depth of k.

2.3 Alpha-Beta Enhancements

The following are some well known enhancements to the previously presented alpha-beta algorithm. All the enhancements we consider are *sound* in the sense that the correct minimax value is still computed, just with less computational effort.

2.3.1 Fail-Soft Alpha-Beta Pruning

In classical alpha-beta, when a score falls outside the alpha-beta window, a score of alpha or beta is returned. This method is known as fail-hard alpha-beta pruning. In fail-soft alpha-beta, when a node's score x exceeds beta or is lower than alpha, we simply return the score. The algorithm still correctly computes the minimax value of a position, only now we retain some extra information when we cutoff. This allows other alpha-beta search enhancements to work more effectively.

2.3.2 Iterative Deepening

This is a driver routine to the alpha-beta search. It works by first performing a search to depth 1, then a search to depth 2, etc. This process continues until a time threshold is exceeded, whereupon the best move and score are recorded. This technique has two main benefits. The first is an effective way to manage your search time - i.e. it is guaranteed that you get a score and best move after the depth 1 search is complete. The second benefit is slightly counterintuitive. Initially it seems natural to assume that searching to depth 1, then depth 2, and so on would be a waste of computational effort. However, because of alpha-beta being so sensitive to move ordering, and the ability of shallower searches to prime various move ordering heuristics, using iterative deepening to search to depth N is typically faster than just invoking a standard alpha-beta search to depth N.

2.3.3 Internal Iterative Deepening

This technique is a move ordering heuristic, which although computationally expensive, can give large performance increases if judiciously applied. Since the performance of alpha-beta is so sensitive to the order of moves searched, in situations where you doubt that you have a good move to search, it is worthwhile to use a lot of CPU time to try and guess it. If we are doing a depth N search, the internal iterative deepening heuristic would say to take the best move from a depth $N - R$ search, where R is the depth reduction factor.

2.3.4 History Heuristic

This is a computationally cheap move-ordering heuristic that improves the performance of the alpha-beta algorithm [7]. Statistics are collected about the frequency of moves which cause cutoffs. These type of statistics are generally domain specific. Before processing the possible moves from a search node, these statistics are used to order moves from most likely to cause a cutoff to least likely.

A good overview of the history heuristic can be found at [7].

2.3.5 Killer Move Heuristic

This is another computationally cheap move ordering heuristic that works well in practice [7]. If we find that a move is sufficient to cause a cutoff at a position d moves from the root, then often this same move will achieve a cutoff at other positions in the search tree that are also d moves from the root.

2.3.6 Principle Variation Search

Modern alpha-beta searchers typically have very good move ordering and this property can be exploited by adjusting the alpha/beta bounds during the recursive alpha-beta calls. The idea is that in a program with good move ordering, typically the best move in a position will be the first move searched. So after this first child is searched, we can initially search with a zero-width window to try and prove that all the subsequent moves are no better. Of course we are taking a gamble, since if another move turns out to be better than the first, then we need to re-search this move with a wider search window. However, in practice this technique gives non-trivial performance gains in most game positions, so it is widely used.

2.3.7 Transposition Table

A transposition table is essentially a large cache of previous search results. It has many uses, and can give massive performance improvements to alpha-beta searchers [7]. The Zobrist hashing scheme [10] is used by many game playing programs, since it provides a very efficient way to incrementally update the hash-key for a given board position.

One use is to suggest a best move for the search to try before any other moves. For example, if you are searching a position to depth N, then often you will have the best move, or the move which caused a cutoff at depth $N - R$ stored inside the transposition table. A good move ordering heuristic is to search the move stored in the transposition table first.

Another use is to save and cache search results. In many games, various sequences of game moves transpose to the same position. Having a transposition table allows one to avoid a lot of

redundant work. More detailed information can be found in [3] and [2].

2.3.8 Enhanced Transposition Cutoffs

This is a computationally expensive, but potentially worthwhile technique to generate cutoffs before any moves are searched. The idea is to first generate the hash-keys of the children of the current node, then check the information contained in the transposition table for each key. If any transposition table entry allows you to cutoff, then you can terminate the search immediately.

Because this technique is computationally expensive, it is usually only applied at nodes with sufficient remaining depth. What is sufficient depends on both the domain and the implementation of the game.

For a more detailed description of the idea, see Shaeffar et al [8].

2.4 Stochastic Game Tree Search

2.4.1 Expectimax

Expectimax is a brute force, depth first game tree search algorithm that generalises the minimax concept to games of chance. By looking into the future, we can compute the expected value of a particular game state. This means averaging the minimax values across all of the possible chance events, taking into account the probability of each event occurring. This generalisation amounts to adding a chance node type to the minimax tree. The successors of a chance node are all of the possible stochastic events that can occur according to the game rules. A sample expectimax game tree is shown in figure 2.2.

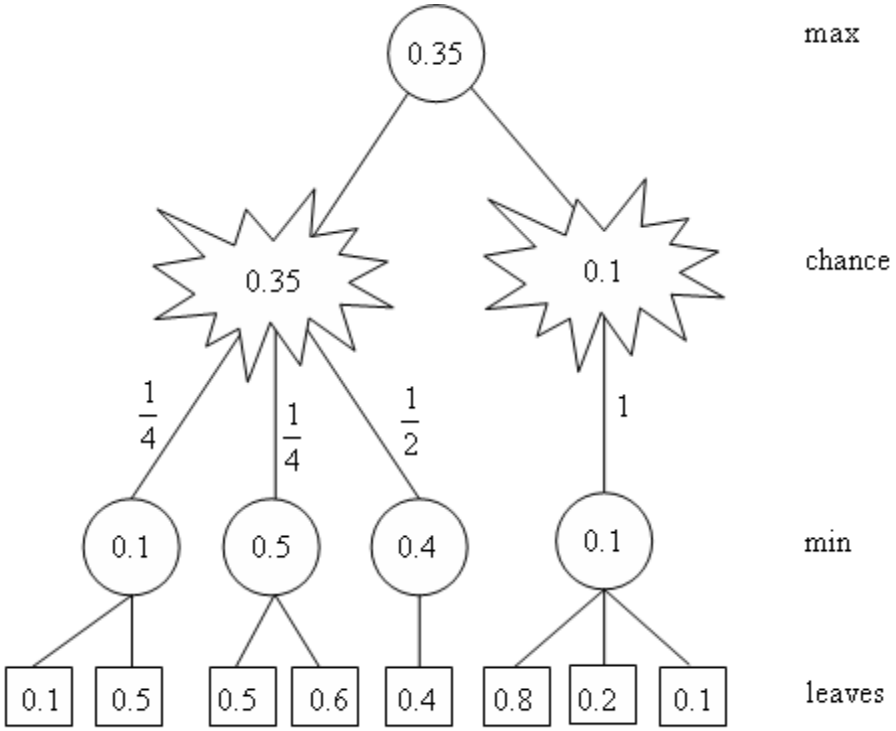


Figure 2.2: A sample expectimax game tree

There are three different non-leaf node types, max, min and chance. The value of a max node is defined to be the highest value of it's children. Similarly, the value of a min node is defined

to be the lowest value of its children. The value of a chance node is defined to be the weighted sum of its children.

More precisely, if b is a game state, b_e is the game state after event e has occurred and $Utility(x)$ represents the utility of being in state x, then:

$$Expectimax(b) = \sum_{e \in Events} Probability(e) \times Utility(b_e) \tag{2.1}$$

Like minimax, expectimax makes use of a heuristic evaluation to compute the utility of a game state at the leaf nodes. In minimax, any monotonic transform of the evaluation function will cause the same move to be chosen. However for expectimax there is an extra restriction on the function that estimates the utility of a state. Because the expectimax score is computed by multiplying probabilities of event occurrences with heuristic evaluation scores, the evaluation scores need to be directly proportional to the likelihood of winning.

Pseudocode

A straightforward modification of the above equation gives a recursive algorithm, which uses a utility function at the leaf nodes of the search tree. We use a driver function, *chanceSearch* to alternate between the chance and min/max search procedures, according to the game rules.

```
Score chanceSearch(Board board, int depth) {
    if (board.needChanceEvent()) return expectimax(board, depth)
    return minimax(board, depth)
}
```

At chance nodes, we use the expectimax algorithm. At min/max nodes, we use an algorithm which computes the exact minimax score for a node.

```
Score expectimax(Board board, int depth) {

    if (isTerminalScore()) return terminalScore()
    if (depth == 0) return evaluate(board)

    Score sum = 0
    for (i = 0; i < numChanceEvents(); i++) {
        board.makeChanceEvent(i)
        Score val = chanceSearch(board, depth-1)
        board.unmakeChanceEvent(i)
        sum += eventProbability(i) * val
    }
    return sum
}
```

2.4.2 Incorporating Alpha-Beta Bounds

At non-chance nodes, the best score is determined by the value of only one move. Thus it is sufficient to fall outside the search window by determining the value of only one successor. We cannot do this at chance nodes. Since expectimax is computing a weighted sum at every node, we need to show that this weighted sum falls outside the search window.

In computing the expectimax value, we note that each recursive call to *chanceSearch* must return an exact score. If we were to naively use *negamax* to compute this score, than we would need to use a search window of $(-\infty, \infty)$. However, if we can place an upper and lower bound on the scores returned by the evaluation function, and modify the expectimax algorithm to compute a score within the interval $[\alpha, \beta]$, then it is possible to compute the expectimax value more efficiently. Intuitively, as we search the successors of a chance node, we gain more information about the final expectimax value of the node. As we shall see, both Star1 and Star2 use this information to reduce the search effort necessary to compute the expectimax value.

2.4.3 Star1

Star1 is a sound pruning technique, introduced by Ballard [1], that can be used at chance nodes. It allows us to use the $\alpha\beta$ bounds passed in from the min/max nodes.

There are two main reasons why Star1 is an improvement over expectimax. If we can prove that the value will fall outside the search window, then we can skip searching the remaining successors and cutoff immediately. Even if we cannot cutoff, as we search each successor, Star1 narrows the search window used for each recursive call. This increases the likelihood of cutoffs further down the search tree.

Notation

We will now introduce the notation which will allow us to formally reason about the Star1 and Star2 algorithms.

Let:

L denote the lower bound on the value of any node.
U denote the upper bound on the value of any node.
$S(n)$ denote the expectimax value of a node n.
$Search_i(n, a, b)$ denote the search value of the $i'th$ child of node n, using a window of $[a, b]$
$S_i(n)$ denote the exact value of the $i'th$ child of node n.
$P_i(n)$ denote the probability of event i occurring at node n.
$N(n)$ denote the number of children of node n.
k denote the number of children currently searched at node n.
$LB(n, k)$ denote the lower bound on the value of node n, after searching k children.
$UB(n, k)$ denote the upper bound on the value of node n, after searching k children.

Observation

At chance nodes, we are computing a weighted average of values of the child nodes. However, if we are only interested in a score between $[\alpha, \beta]$, and we know that the value of each child must be in the interval $[L, U]$, then we have:

$$LB(n,k) = \sum_{i=0}^{k-1} P_i(n) \times S_i(n) + \sum_{i=k}^{N(n)-1} P_i(n) \times L \tag{2.2}$$

$$UB(n,k) = \sum_{i=0}^{k-1} P_i(n) \times S_i(n) + \sum_{i=k}^{N(n)-1} P_i(n) \times U \tag{2.3}$$

Therefore, if $LB(n,k) \geq \beta$ then $S(n) \geq \beta$. Similarly, if $UB(n,k) \leq \alpha$ then $S(n) \leq \alpha$.

Algorithm Derivation

The Star1 algorithm exploits this observation to narrow the $\alpha\beta$ window in the recursive calls to *Search*. This narrowed search window makes it easier for successor nodes to skip redundant search work. However, we need to be careful about how much we narrow this window. If we narrow the window too much, we will not have enough information to compute the expectimax value. We will now analytically derive the smallest window for each recursive call to *Search*, so that we can always compute the exact expectimax value or fall outside the search window for any given node.

Assume we are at a node with a search window of $[\alpha, \beta]$. When searching the $k'th$ successor, we want a minimal window $[a_k, b_k]$ so that we either get an exact score, or an α or β cutoff. More specifically, we want an $[a_k, b_k]$ such that if $Search_k(n, a_k, b_k) \leq a_k$ then $S(n) \leq \alpha$, and if $Search_k(n, a_k, b_k) \geq b_k$ then $S(n) \geq \beta$.

We can derive this window by noting that:

$$LB(n,k) = LB(n,k-1) + P_k(n) \times S_k(n) - P_k(n) \times L \tag{2.4}$$

$$UB(n,k) = UB(n,k-1) + P_k(n) \times S_k(n) - P_k(n) \times U \tag{2.5}$$

So, for no cut-off to occur, $LB(n,k) < \beta$ and $UB(n,k) > \alpha$, therefore

$$LB(n,k-1) + P_k(n) \times S_k(n) - P_k(n) \times L < \beta \tag{2.6}$$

$$UB(n,k-1) + P_k(n) \times S_k(n) - P_k(n) \times U > \alpha \tag{2.7}$$

Upon re-arranging, we have:

$$S_k(n) < \frac{\beta - LB(n,k-1)}{P_k(n)} + L \tag{2.8}$$

$$S_k(n) > \frac{\alpha - UB(n,k-1)}{P_k(n)} + U \tag{2.9}$$

Hence

$$\frac{\alpha - UB(n,k-1)}{P_k(n)} + U < S_k(n) < \frac{\beta - LB(n,k-1)}{P_k(n)} + L \tag{2.10}$$

and since all scores must lie between $[L,U]$, we have

$$a_k = max(L, \frac{\alpha - UB(n,k-1)}{P_k(n)} + U) \tag{2.11}$$

$$b_k = min(U, \frac{\beta - LB(n,k-1)}{P_k(n)} + L) \tag{2.12}$$

Pseudocode

The following routines implement the above equations. They are used extensively by the Star family of algorithms.

We start by defining a data structure which will be used to hold the bound information for a single chance event. At every chance node, we use an array of these EventInfo structures to store the information we gather about each successor.

```
type EventInfo {
    LowerBound defaults to L
    UpperBound defaults to U
    Event
    Probability defaults to probability of this event occurring
}
```

We then show how to compute the lower, upper and exact bounds on the expectimax score for this node, given the current information we have about the successors.

```
Score lowerBound(EventInfo[] event_info) {
    Score total = 0
    for (i=0; i < numChanceEvents(); i++) {
        total += event_info[i].LowerBound * event_info[i].Probability
    }
    return total
}

Score upperBound(EventInfo[] event_info) {
    Score total = 0
    for (i=0; i < numChanceEvents(); i++) {
        total += event_info[i].UpperBound * event_info[i].Probability
    }
    return total
}

Score exactValue(EventInfo[] event_info) {
    assert(lowerBound(event_info) == upperBound(event_info))
    return lowerBound(event_info)
}
```

The following pseudocode computes the most narrow search window that can be used to determine the value of a successor, given what we currently know about the node.

```
Score computeSuccessorMin(EventInfo[] event_info, Score alpha) {
    Score cur_alpha = alpha - event_info.upperBound()
    cur_alpha += event_info[i].Probability * event_info[i].UpperBound
    cur_alpha /= prob
    return cur_alpha
}

Score computeSuccessorMax(EventInfo[] event_info, Score beta) {
    Score cur_beta  = beta - event_info.lowerBound()
    cur_beta += event_info[i].Probability * event_info[i].LowerBound
    cur_beta /= prob
    return cur_beta
}
```

We now present the Star1 algorithm. It is worth noting that it would be possible to efficiently compute the ax and bx quantities incrementally by using identities 2.4 and 2.5.

```
Score star1(Board board, Score alpha, Score beta, int depth) {

    if (isTerminalPosition()) return TerminalScore()
    if (depth == 0) return evaluate(board)

    EventInfo event_info[numChanceEvents()]
    generateChanceEvents()

    for (i = 0; i < numChanceEvents(); i++) {
        Score cur_alpha = computeSuccessorMin(event_info, alpha)
        Score cur_beta = computeSuccessorMax(event_info, beta)
        Score ax = max(L, cur_alpha)
        Score bx = min(U, cur_beta)

        board.applyChanceEvent(i)
        Score search_val = negamax(board, ax, bx, depth-1)
        board.undoChanceEvent(i)

        event_info[i].LowerBound = search_val
        event_info[i].UpperBound = search_val

        if (search_val >= cur_beta) return lowerBound(event_info)
        if (search_val <= cur_alpha) return upperBound(event_info)
    }

    return exactValue(event_info)
}
```

2.4.4 Star2

Star2 is an extension of Star1 that has a best case complexity that is an order of magnitude better than expectimax [1].

Star2 exploits the regularity in many domains where stochastic game tree search is useful. If we know that a chance node will always be followed by a min/max node, then we can use the Star2 algorithm. This type of stochastic game tree is called a *regular *-minimax tree* by Ballard [1].

The Star2 algorithm augments the Star1 procedure with a preliminary probing phase. Using a negamax framework, a chance node from a *-minimax tree will always be followed by a max node. If we search one child from each successor of a chance node, we establish a lower bound for each successor. If at any stage we determine that we will fall outside the $[\alpha, \beta]$ window, we immediately return the score. If the probing stage fails to exceed beta, then we re-search all of the successors, using the same procedure as Star1.

This preliminary probing phase has two main benefits. The best possible outcome is that the lower bounds we establish for the chance events are large enough so that we can prove the expectimax score exceeds beta. Failing that, we can use the extra lower bound information to narrow the search window when evaluating successor nodes in the Star1 part of the algorithm. Note that it is not possible to determine any kind of upper bound during the probing phase, since this requires searching all of the children of each successor.

A straightforward generalisation of Star2, mentioned by Ballard [1], is to consider k moves of each successor rather than just one. We denote the number of children searched from each successor as the *ProbingFactor*.

Move ordering

In a normal alpha-beta search, it is important to have good move ordering. It is even more important to have good move ordering at min/max nodes in Star2, since this is what determines

the quality of the lower bounds found by the probing phase.

Pseudocode

We have deliberately left the negamax-probe function undefined. This is because it is defined exactly the same as negamax, with the addition of a *ProbingFactor* argument. This argument limits the number of moves searched at the root. Because of this limiting factor, the scores returned by negamax-probe are only lower bounds.

```
Score star2(Board board, Score alpha, Score beta, int depth) {
    EventInfo event_info[numChanceEvents()]
    if (isTerminalPosition()) return TerminalScore()
    if (depth == 0) return evaluate(board)

    generateChanceEvents()
    for (i = 0; i < numChanceEvents(); i++) {
        Score cur_beta = computeSuccessorMax(event_info, beta)
        Score bx = min(U, cur_beta)
        board.applyChanceEvent(i)
        Score search_val = negamax-probe(board, L, bx, depth-1, ProbingFactor)
        board.undoChanceEvent(i)
        event_info[i].LowerBound = search_val
        if (search_val >= cur_beta) return lowerBound(event_info)
    }

    for (i = 0; i < numChanceEvents(); i++) {
        Score cur_alpha = computeSuccessorMin(event_info, alpha)
        Score cur_beta = computeSuccessorMax(event_info, beta)
        Score ax = max(L, cur_alpha)
        Score bx = min(U, cur_beta)
        board.applyChanceEvent(i)
        Score search_val = negamax(board, ax, bx, depth-1)
        board.undoChanceEvent(i)
        event_info[i].LowerBound = search_val
        event_info[i].UpperBound = search_val
        if (search_val >= cur_beta) return lowerBound(event_info)
        if (search_val <= cur_alpha) return upperBound(event_info)
    }
    return exactValue(event_info)
}
```

Chapter 3

StarETC

StarETC is a new stochastic game tree search algorithm that is an extension of Star2. It contains two enhancements over the Star2 algorithm.

Transposition tables are frequently used by alpha-beta searchers at min and max nodes. We show how to generalise this enhancement to stochastic game tree search. Hauk mentions that a transposition table is used in [4], but it is not clear whether its usage was extended to chance nodes as well. The implementation is non-trivial, so we have provided detailed pseudocode for the enhancement.

The second improvement is the addition of a novel probing phase based on the Enhanced Transposition Cutoff idea described in section 2.3.8. This has not been previously described in the literature.

3.1 Transposition Table Enhancement

A transposition table (see section 2.3.7) is typically implemented as a large hash table, where we overwrite entries in the event of a collision.

In each entry we store:

1. A score, which could be an upper, low or exact bound

2. The depth the information is based on
3. An optional suggested best action
4. The game state

This information is typically stored at min and max nodes during an enhanced alpha-beta search. We notice that the Star2 algorithm either returns an exact or bounded score. This information can be stored in the transposition table as well. To take advantage of this information, all that remains is to specify the implementation details for chance nodes.

There are two main operations we can perform with a transposition table, *store* and *retrieve*.

Store

This operation is invoked whenever we return a value from a chance node. We cache the depth, score and whether the score is an exact, lower or upper bound. If the Star2 probing phase determines that we have a β cutoff, we store a lower bound. If the Star1 part of the search fails high, then we store a lower bound. If it fails low, we store an upper bound. If no cutoff occurs, then we store an exact bound.

Often two game states will hash to the same position within the transposition table. When this occurs, it is up to the *replacement scheme* to determine what happens. The replacement scheme is the algorithm which determines exactly what entries get replaced or updated within the table. There are a lot of different replacement schemes, with their own advantages and disadvantages. See [2] for a comparison of the different methods.

Retrieve

Before searching or probing any successors, we check for a matching entry in the transposition table. If we find an entry that matches or exceeds the current required depth, we use the bound information to either tighten the $\alpha\beta$ window, or prove that the score for this node lies outside the $[\alpha\beta]$ interval.

3.2 Probing Enhancement

The Enhanced Transposition Cutoff idea (see section 2.3.8) augments a normal alpha-beta search with a preliminary transposition table probing phase. The transposition table is probed for each successor. If any one of these entries has enough information for us to exceed beta, we can cutoff the search immediately.

We can augment the Star2 algorithm with a similar probing phase. The details for the stochastic case are a bit more complicated, since any bound information we get will be useful. We look for a matching transposition table entry, with minimum depth of $depth - 1$ for each successor. Any information that the entry contains is either used to immediately prove that the expectimax score is outside the $[\alpha, \beta]$ interval, or stored and then used to tighten the successor search bounds during the subsequent Star2 and Star1 parts of the search. These tightened bounds can reduce the search effort for this node.

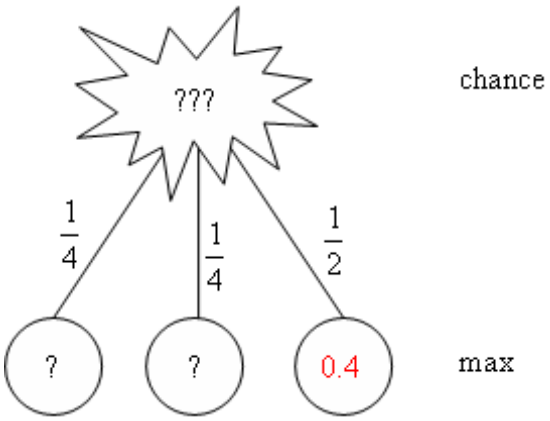


Figure 3.1: A situation where the probing enhancement can help

Suppose we are at the root node in figure 3.1, with a search window of $[\alpha, \beta]$. Since we are using a transposition table, we have a cache of previous search results. Suppose there exists an entry in the transposition table for the third successor. Suppose this entry contains the exact score for the successor node. If the expectimax value of the root node is going to be less than α, 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.

An efficient hashing scheme is an important requirement for the StarETC algorithm to outperform Star2. Since StarETC relies on extra probing of the hash table to tighten search bounds, if the hash key was expensive to compute, then the reduction in nodes searched might not justify the probe cost. Zobrist hashing [10] will be the method of choice for many game domains, since the hash key can be efficiently computed during the search.

3.3 Pseudocode

The details of correctly using the transposition table make the pseudocode long and complicated. We present the full algorithm, since correctly using the transposition table is non-trivial.

The code is broken up into four sections. We first see whether the transposition table has any cached information about this node. We then try the probing enhancement. If this stage fails to find a cutoff, we try the Star2 probing phase. Notice that we use the lower bound established by the transposition table probing enhancement in the Star2 probing phase. If the two probing phases fail, we then proceed to the Star1 part of the search.

```
Score starETC(Board board, Score alpha, Score beta, int depth) {

    if (isTerminalPosition()) return TerminalScore()
    if (depth == 0) return evaluate(board)

    EventInfo event_info[numChanceEvents()]
    generateChanceEvents()

    // probe transposition table for node information
    if (transtbl.probe(board, depth) == Success) {
        if (entry.boundType() == Exact) return entry.score()
        if (entry.boundType() == Lower) {
            if (entry.score() >= beta) return entry.score()
            alpha = max(alpha, entry.score())
        }
        if (entry.boundType() == Upper) {
            if (entry.score() <= alpha) return entry.score()
            beta = min(beta, entry.score())
        }
    }

    // probe transposition table for successor information
    for (i = 0; i < numChanceEvents(); i++) {
        applyChanceEvent(i)
        if (transtbl.probe(board, depth-1) == Success) {
            if (entry.boundType() == Lower ||
                entry.boundType() == Exact) {
                event_info[i].LowerBound = entry.score()
                if (lowerBound(event_info) >= beta) {
                    transtbl.update(board, depth, LowerBound,
```

```
                                lowerBound(event_info))
                return lowerBound(event_info)
            }
        } else if (entry.boundType() == Upper ||
                   entry.boundType() == Exact) {
            event_info[i].UpperBound = entry.score()
            if (upperBound(event_info) <= alpha) {
                transtbl.update(board, depth, UpperBound,
                                upperBound(event_info))
                return upperBound(event_info)
            }
        }
    }
    undoChanceEvent(i)
}

// modified Star2-like probing phase
for (i = 0; i < numChanceEvents(); i++) {
    if (lowerBound(event_info) != node_info[i].UpperBound) {
        Score cur_beta = computeSuccessorMax(event_info, beta)
        Score bx = min(U, cur_beta)
        board.applyChanceEvent(i)
        Score search_val = negamax-probe(board, event_info[i].LowerBound,
                                         bx, depth-1, ProbingFactor)
        board.undoChanceEvent(i)
        event_info[i].LowerBound = search_val
        if (search_val >= cur_beta) {
            transtbl.update(board, depth, LowerBound, lowerBound(event_info))
            return lowerBound(event_info)
        }
    }
}
```

```
    // Star1 search phase
    for (i = 0; i < numChanceEvents(); i++) {
        Score cur_alpha = computeSuccessorMin(event_info, alpha)
        Score cur_beta = computeSuccessorMax(event_info, beta)
        Score ax = max(L, cur_alpha)
        Score bx = min(U, cur_beta)
        board.applyChanceEvent(i)
        Score search_val = negamax(board, ax, bx, depth-1)
        board.undoChanceEvent(i)
        event_info[i].LowerBound = search_val
        event_info[i].UpperBound = search_val
        if (search_val >= cur_beta) {
            transtbl.update(board, depth, LowerBound, lowerBound(event_info))
            return lowerBound(event_info)
        }
        if (search_val <= cur_alpha) {
            transtbl.update(board, depth, UpperBound, upperBound(event_info))
            return upperBound(event_info)
        }
    }

    transtbl.update(board, depth, ExactBound, exactValue(event_info))
    return exactValue(event_info)
}
```

Chapter 4

Experimental Framework

In this chapter, we introduce our experimental framework that is based around the game of Dice. The implementation of this framework took a significant amount of time and effort. The framework includes four different stochastic search algorithms incorporating an enhanced alpha-beta search, a $TD(\lambda = 0)$ trained feed-forward neural network for the evaluation function, as well as a graphical interface that allows for batch processing of Dice positions.

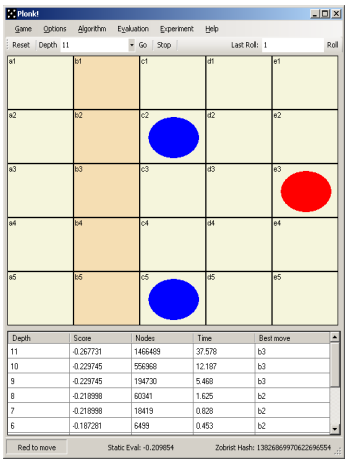


Figure 4.1: A game of Dice in progress

4.1 The rules of Dice

Dice is a simple two-player stochastic game played on an $n \times n$ board. It was first introduced by Hauk in [4]. Players take turns placing checkers on the board. Checkers cannot be placed in locations where an existing checker resides. At the start of every turn, an n sided die is rolled to determine where the player to move can place a checker.

This works as follows. One player is the column player, and the other is the row player. At the start of each turn, the player to move rolls a die. If this player is the column player, then the result of the die roll determines which column they are permitted to place a checker in. Similarly, the die roll for the row player determines which row they may play in. If the die roll indicates a row or column that is already completely filled, it is re-rolled. The players alternate turns until the game ends in a win, loss or when the entire board is filled, in which case the game ends in a draw.

Checkers are considered to be connected if they are horizontally, vertically or diagonally adjacent to one another. The aim of the game is to get some $m \leq n$ connected checkers of your own colour in a row.

The experimental framework supports 5×5, 9×9, 13×13, 17×17 board sizes. The win length can be either 4, 6 or 8. Blue denotes the row player and red denotes the column player.

4.2 Heuristic Evaluation Function

4.2.1 Introduction

Some care needs to be taken with the construction of the heuristic evaluation function, if we want to use it in conjunction with an expectimax searcher. Recall from section 2.4.1, our evaluation function needs to be a positive linear transform of the likelihood of winning.

4.2.2 Why This Restriction?

Here is an example of how the expectimax search gives incorrect results if this restriction is lifted. Suppose we had an evaluation function that was based on some domain specific heuristics, with the properties specified in the table below. Consider the game tree specified by figure 4.3.

Evaluation Score	True Probability of Winning
0	0
1	0.65
2	0.95
3	0.955
4	0.96

Figure 4.2: A sample heuristic evaluation function

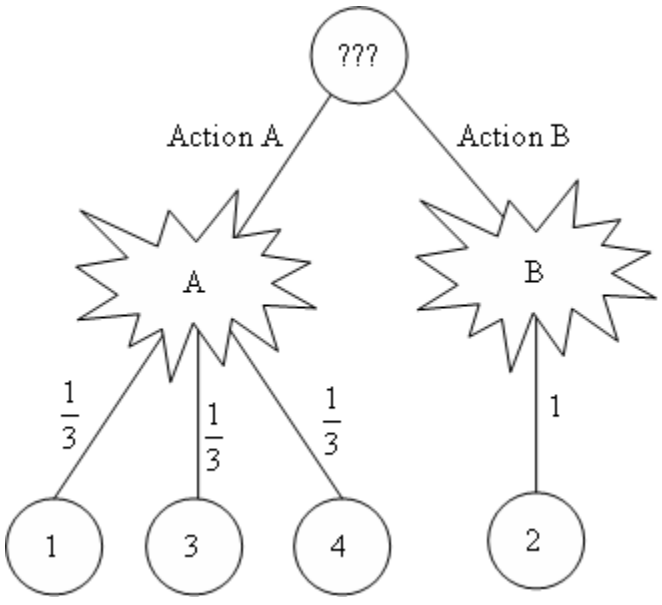


Figure 4.3: An example stochastic game tree

The chance node A would have a value of $2\frac{1}{3}$, greater than the value of 2 for B. Therefore the expectimax search would choose action A. However, this is incorrect. Consider what would happen if we computed the value of node A and node B using the true winning probabilities. Node A would have a value of $0.65 \times \frac{1}{3} + 0.955 \times \frac{1}{3} + 0.96 \times \frac{1}{3} = 0.85\bar{3}$, compared to node B having a value of 0.95. Therefore, action B is best for this position.

Hauk reported negligible performance improvements by Dice players using deeper searches versus shallower searching Dice players in 7×7, 4-in-a-row Dice [4]. He states that *"the evaluation function consisted of simply counting pairs of squares. A pair is two squares filled for the same player, next to each other in the same row, column, or diagonal. Two squares separated by a single empty square on a row, column or diagonal were also considered to be a pair. The evaluation function counted pairs for both opponents and took the difference."* There was no mention of a non-linear transfer function. Although this is just speculation, it is quite possible that the deeper searches did not help Hauk because his evaluation function was not returning scores that were directly proportional to the likelihood of winning.

4.2.3 Our Solution

Our evaluation function consists of two parts, a hand-coded feature recognition module, and a non-linear function approximator. Our non-linear function approximator is a standard multi-layer perceptron. In computing the value of a state, we first determine the active board features, normalise them, and then feed them through the neural network. We use self play and TD-learning to train the evaluation function, similar to the process used by Tesauro's TD-Gammon [9].

We feel that this is a reasonable choice, for a number of reasons:

1. The stochastic nature of Dice forces many different game states to be explored during self play.

2. Dice games have fixed maximum length, which means that we do not have to worry about games repeating forever, or getting into unproductive cycles.

3. Draws are possible, but very rare, so the estimated reward of a state gives us a reasonable estimate of the likelihood of winning.

Feature Recognition

Our goal is to get m checkers in a row. A simple heuristic that gives a rough measure of progress to this goal is the number of streaks of length $k < m$, for each side to move. A key advantage of this heuristic is that it is applicable to many different Dice board-size/win-length configurations.

For example, consider a game where we are trying to get a run of 4 checkers in a row. Suppose blue has 3 streaks of length 2 and two streaks of length 3. If red has only 1 streak of length 2, and no streaks of length 3, then the heuristic suggests that blue has a much higher chance of winning.

Network Design

We are using a feed-forward neural network with a single hidden layer, and one output unit. The number of input and hidden units depends on the board dimension. Because of this, we have a different network for every different configuration of Dice.

The first twelve inputs to the network, irrespective of the board dimension, are reserved for the hand coded evaluation features. The remaining inputs are for the board state. There are two network inputs for every game cell. These two inputs can take on any one of three possible values:

1. $< 1.0, 0.0 >$ - indicating that the cell is occupied by a blue checker
2. $< 0.0, 1.0 >$ - indicating that the cell is occupied by a red checker
3. $< 0.0, 0.0 >$ - the cell is empty

The following table summarises the different network topologies:

Board Dimension	No. Inputs	No. Hidden units
5	62	10
9	174	15
13	350	20
17	590	30

Figure 4.4: The network topologies for different Dice configurations

Training

Initially the weights of the neural network are set to small random values near zero. We can let the engine play itself, by using 1 ply searches to determine the engine actions. It is important to point out that a 1-ply search will return a win score if there is a move that allows a side to win. After the engine makes a move on the game board, we use the $TD(0)$ algorithm to compute the temporal difference of the position. This is simply $T_k = P_k - P_{k-1}$, where P_k denotes the evaluation of the k'th game state. We then invoke the backpropagation algorithm, using T_k as the error term.

300,000 self play matches were used to train the neural networks used by our Dice evaluation function.

4.2.4 Evaluation Quality

It is clear from playing the Dice opponent before and after training, that the training procedure makes a substantial improvement in playing strength. The network learns that controlling the centre is of vital importance, and that checkers on the edges of the board have limited utility. The moves suggested by shallow searches rarely change as we search deeper, and the scores stay

very consistent across the different search depths, even in the endgame where we can search the entire game tree.

4.3 Search

The framework allows you to choose a search algorithm to examine the position. This search algorithm is used by an iterative deepening driver routine. All of these algorithms were introduced previously. Each algorithm uses exactly the same negamax procedure to search the non-chance nodes.

4.3.1 Negamax Enhancements

The negamax search also includes the following previously described alpha-beta enhancements:

1. Fail-Soft Alpha Beta (see section 2.3.1)
2. Transposition Table (see section 2.3.7)
3. History Heuristic (see section 2.3.4)
4. Killer move heuristic (see section 2.3.5)

4.3.2 Algorithms Implemented

We have implemented the following two-player stochastic game tree search algorithms:

1. Expectimax (see section 2.4.1)
2. Star1 (see section 2.4.3)
3. Star2 (see section 2.4.4)
4. StarETC (see chapter 3)

4.4 Search output

It is important to understand how to interpret the search results. These results are generated by the search process and fed back to the GUI. They are then displayed in a table beneath the game board.

Depth	Score	Nodes	Time	Best move
9	-0.128549	56956	1.891	d3
8	-0.213644	23411	0.875	d3
7	-0.213644	7491	0.532	d3
6	-0.196547	2792	0.375	d3
5	-0.196547	807	0.297	d3
4	-0.17037	272	0.25	d3
3	-0.17037	77	0.235	d3
2	-0.144769	21	0.204	d3
1	-0.144769	3	0.188	d3

Figure 4.5: Sample search results

Depth

The current search depth. Because we use iterative deepening, if we have a result for depth N, then we have all of the preceding results for depths 1 to $N-1$. The results of the deepest search are shown first, since we make the assumption that a deeper search yields more accurate results.

Score

The score reflects the side to move's utility. Scores lie in the interval $[-1, 1]$. Positive scores mean that the search thinks that the position favours the side to move, negative scores mean the search thinks the position favours the side not to move. A score of 1 means that the current side to move has a definite win. A score of -1 means the side to move has definitely lost.

Nodes

The total number of nodes needed by the search to reach this depth. This includes all of the searches done to shallower depths by the iterative deepening driver.

Time

The total amount of time required to reach this depth, in seconds.

Best Move

The coordinates of the move that maximises the utility of the side to move.

4.5 Dice Position Notation

To compare the search efficiency of the various algorithms, we need to collect data across a wide range of positions. To further this aim, a simple, text-based, position description language was developed. We shall make use of the abbreviation *DPN* to refer to *Dice Position Notation*.

4.5.1 DPN Grammar

DPN : <dimension> <win condition> <stm> <roll state> {<coords>, ... }
dimension : { 5, 9, 13, 17 }
win condition : { 4, 6, 8 }
stm : { b, r }
roll state : { -1, 1, 2, 3, ... , 17 }
coord : { -, b, r }

There are $dimension^2$ coordinates. A roll state of -1 is used to indicate that the die needs to be rolled before a move can be made. A coordinate that is a dash denotes an empty square.

Example

A graphical representation of a game state is shown in Figure 4.6.

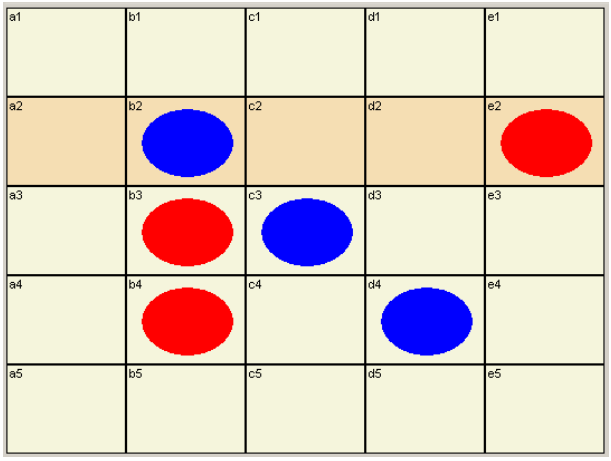


Figure 4.6: sample game position

It's corresponding DPN representation is:

5 4 b 1 - - - - - - b - - r - r b - - - r - b - - - - - -

4.6 Testsuites

It is possible to perform a batch run across a set of positions. One or more algorithms can be tested at various search depths. All of the results are then logged to disk. This was the method we used to obtain the data for our experimental results.

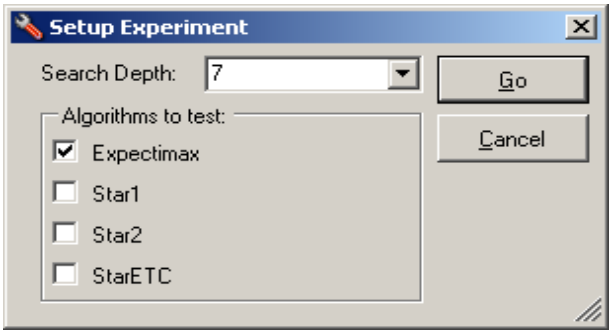


Figure 4.7: Creating a new batch run

Chapter 5

Experimental Results

In this chapter, we provide an empirical performance analysis of the Expectimax, Star1, Star2 and StarETC algorithms using 5x5 Dice.

5.1 Evaluating Search Efficiency

We use two separate metrics to measure search efficiency.

5.1.1 Nodes Searched

This refers to how many nodes we examine when traversing the game tree. This metric allows us to compute the potential benefit of tree pruning schemes, irrespective of their implementation overhead.

5.1.2 Time to Depth

A more practical metric is how long it takes to complete a search to a particular depth. So long as the overhead for the various pruning mechanisms is negligible, an algorithm that searches less nodes compared to another algorithm should also spend less time reaching a particular depth.

5.1.3 Setup

The experiments were conducted on an AMD AthlonFX60, running WindowsXP 64-bit edition. 64 megabytes of memory were used for the transposition table.

Each result is based on the search performance across 100 positions. These positions are listed in Appendix A.1. The positions were chosen randomly from the log files of a TD learning run, using an already well trained evaluation function. This gives a representative sample of search positions that would typically occur in real games.

5.2 Results

Overview

The following graph shows the average number of nodes searched for depths up to 13. The graph is using a logarithmic scale.

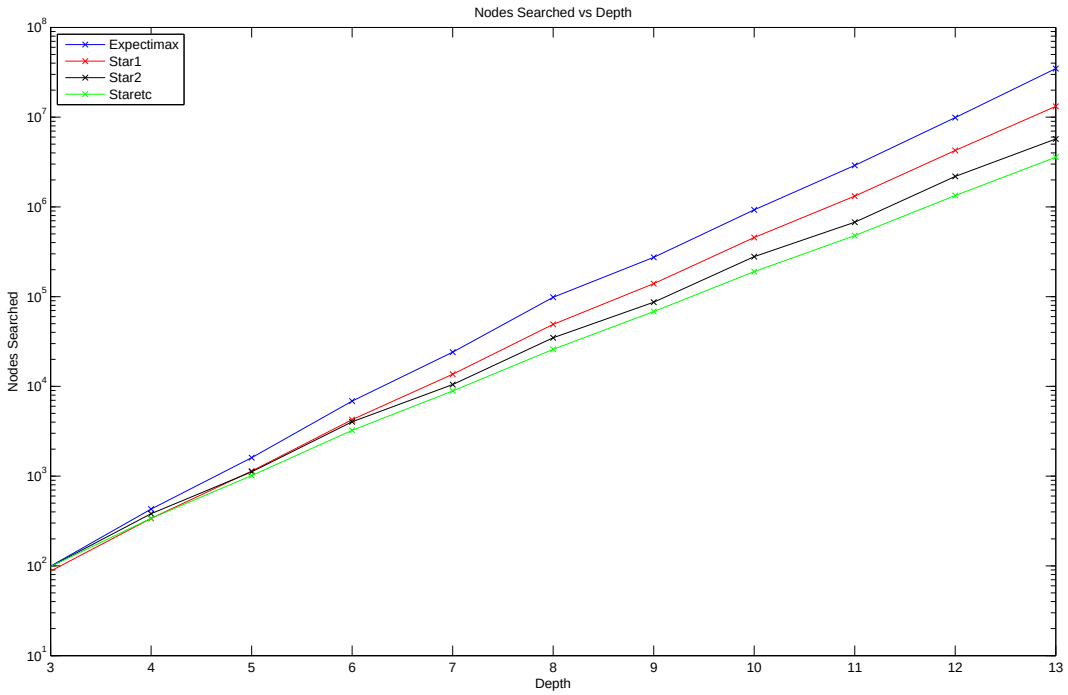


Figure 5.1: Nodes Searched vs Search Depth

Our next results will give a much more detailed breakdown of the relative search performance for each algorithm.

Nodes Searched Metric

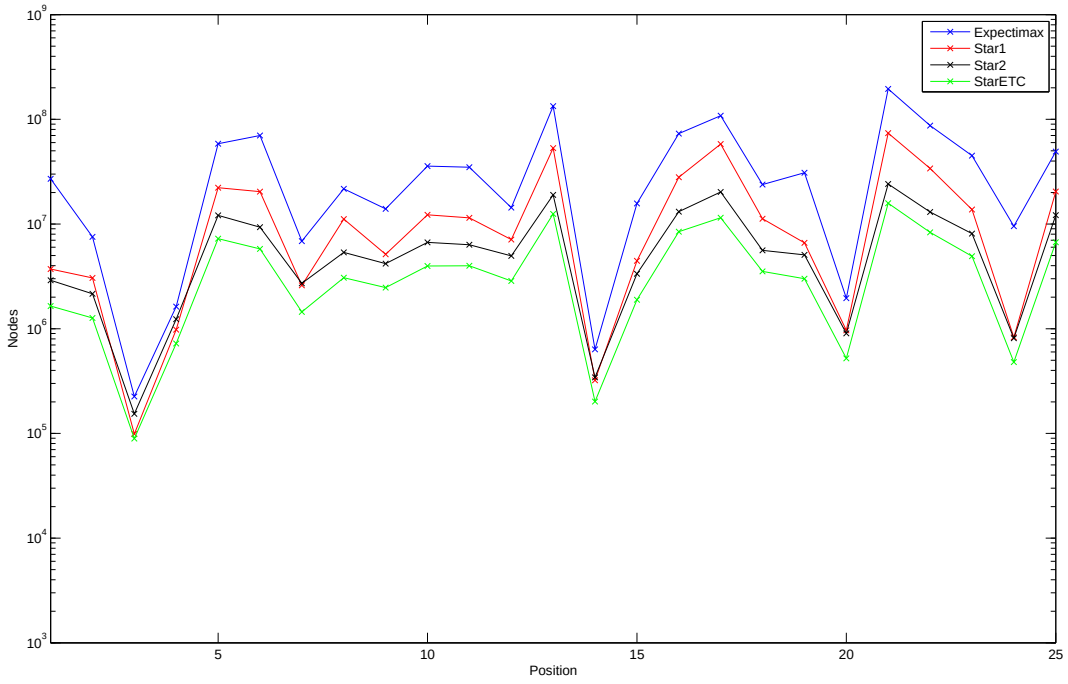
This series of tables and figures summarise the relative performance of the four algorithms across our test positions. Performance numbers are given on a logarithmic scale. Each position was searched to 13 plies.

The table in figure 5.2 summarises our nodes searched results:

	Expectimax	Star1	Star2	StarETC
min	5813	3997	8720	5536
max	2.73e+08	1.35e+08	3.23e+07	2.20e+07
mean	3.48e+07	1.32e+07	5.71e+06	3.58e+06
median	1.28e+07	3.91e+06	2.95e+06	1.83e+06
std	4.99e+07	2.06e+07	6.31e+06	4.12e+06

Figure 5.2: Number of nodes searched to depth 13

The results for each of the 100 positions are presented across the following two pages:



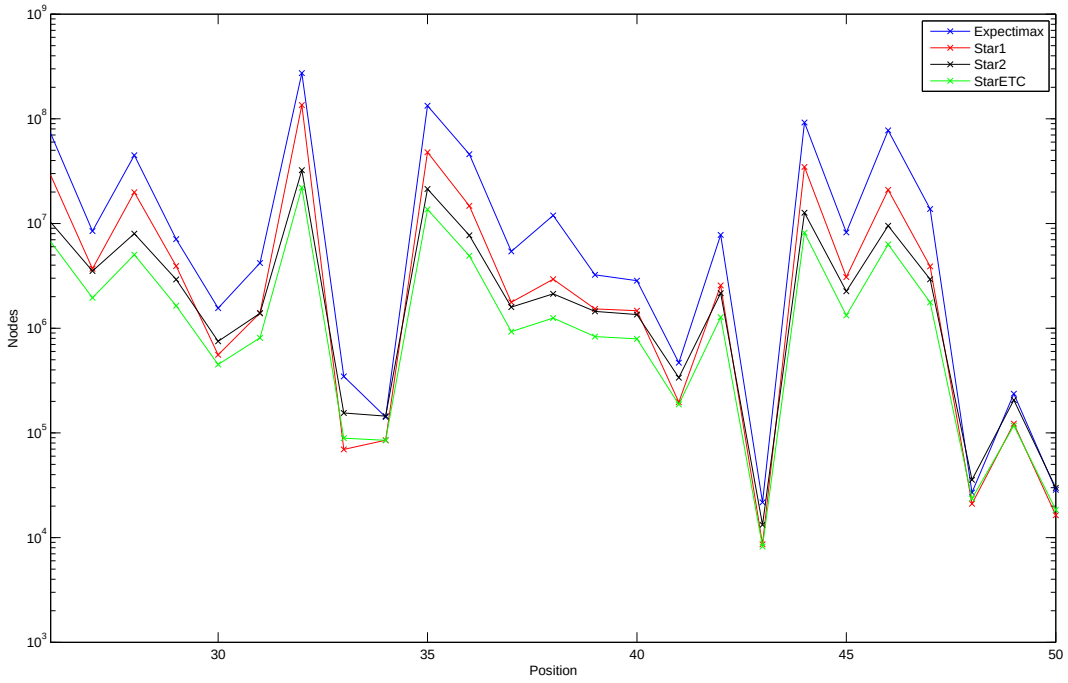
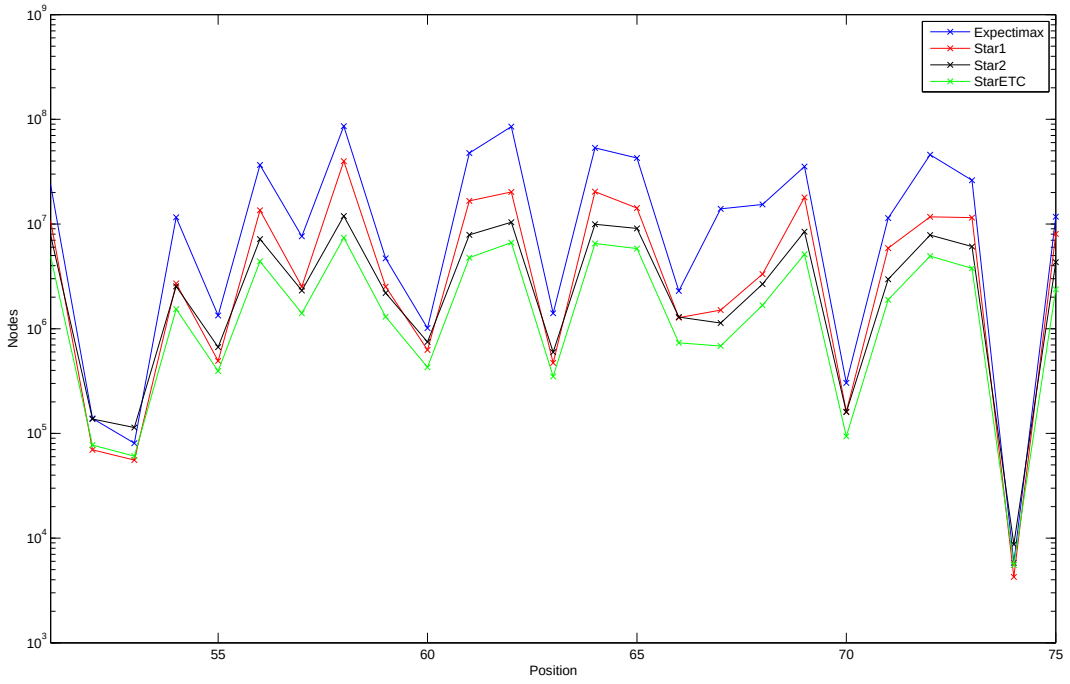


Figure 5.3: Nodes searched for positions 1-50



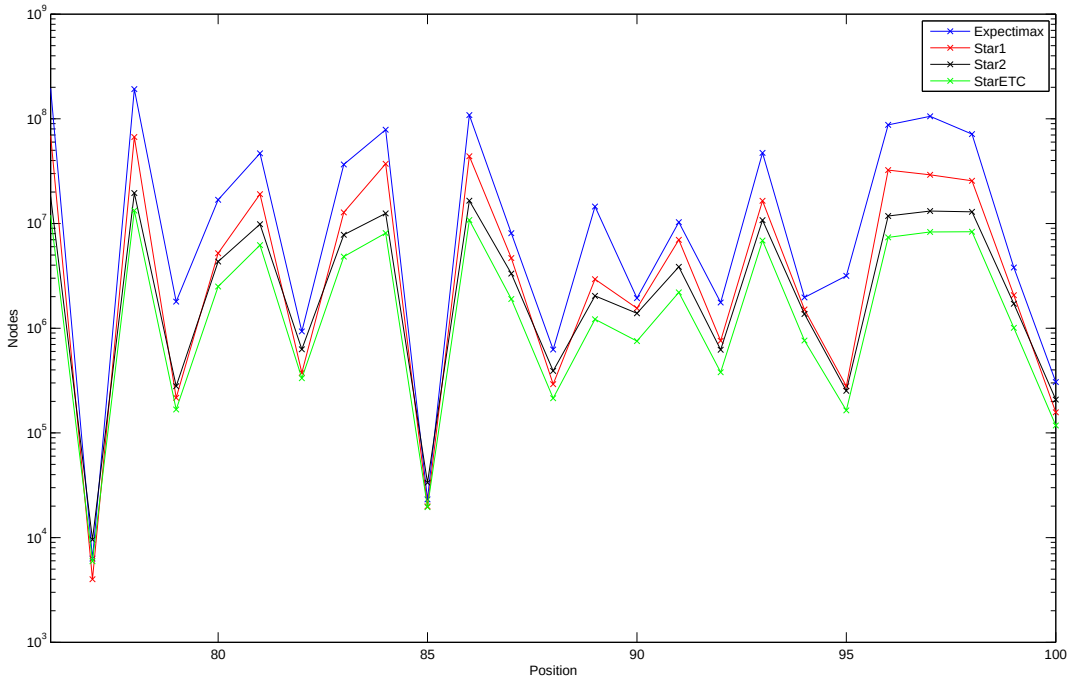


Figure 5.4: Nodes searched for positions 51-100

Time to Depth Metric

We now show the time taken to search 13 plies ahead on our 100 test positions. As we shall see, an algorithm's performance using the time to depth metric is highly correlated with its performance using the node count metric. All times are given in milliseconds.

The table in figure 5.5 summarises our time to depth results:

	Expectimax	Star1	Star2	StarETC
min	219	187	250	219
max	4.53e+06	2.26e+06	4.89e+05	3.65e+05
mean	5.51e+05	2.10e+05	8.16e+04	5.62e+04
median	1.92e+05	6.02e+04	4.18e+04	2.76e+04
std	8.15e+05	3.39e+05	9.25e+04	6.63e+04

Figure 5.5: Time (ms) to depth 13

The results for each of the 100 positions are presented across the following two pages:

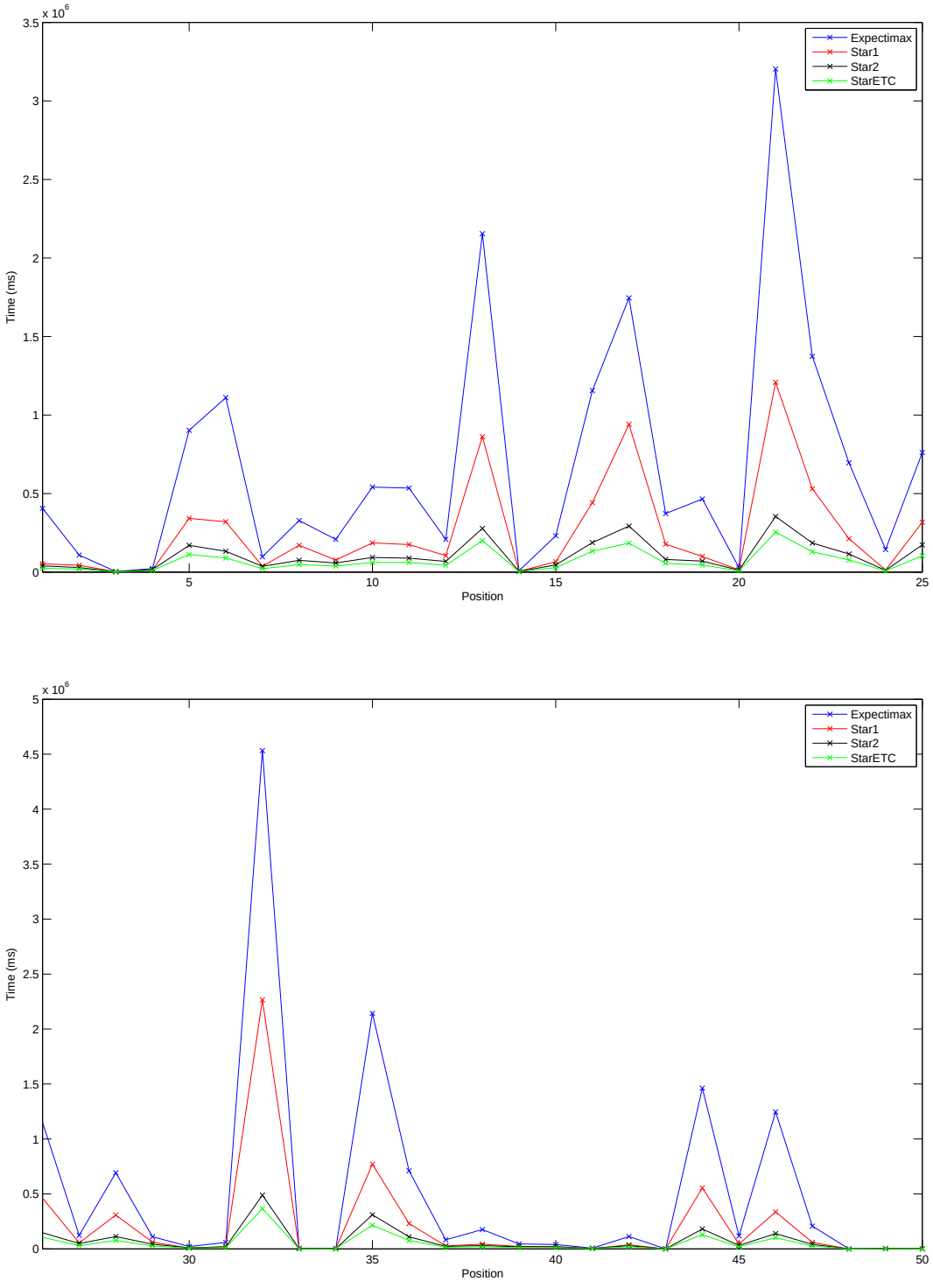


Figure 5.6: Time to depth 13 for positions 1-50

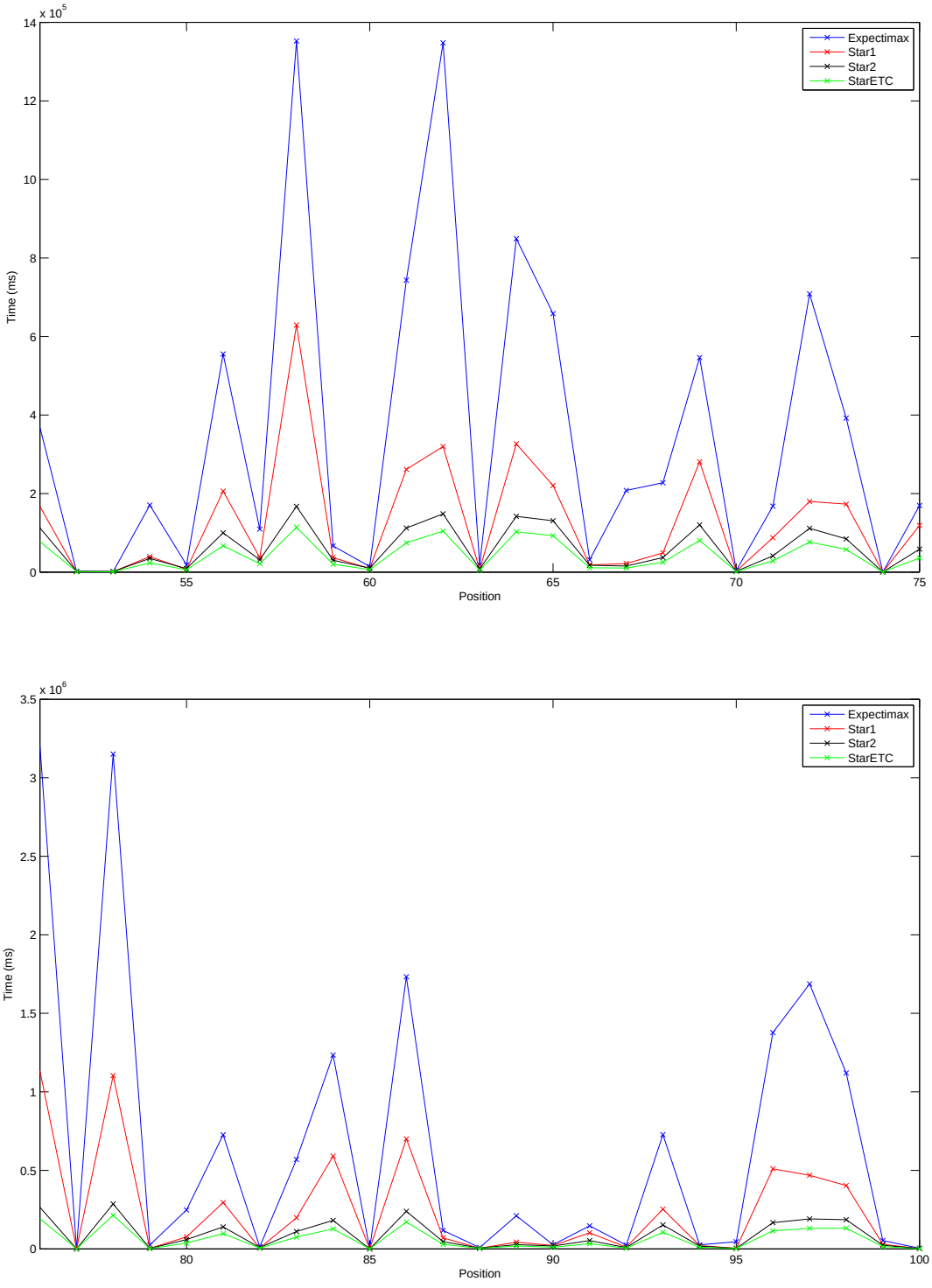


Figure 5.7: Time to depth 13 for positions 51-100

5.3 Discussion

The Star2 algorithm is a significant improvement over Star1, and Star1 is a significant improvement over Expectimax. StarETC searches on average 37% less nodes to depth 13 than Star2. This is a significant performance improvement, and well worth the implementation hassle for any competitive stochastic game playing program.

Although it is possible in degenerate cases for Star2 and StarETC 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 less calls to the evaluation function need to be made. The extra probing phase in StarETC is worth the effort. On average, it takes 31% less time for StarETC to reach depth 13 compared to Star2.

The wide variance in search time for each different position is mainly due to the nature of Dice. The branching factor of the game is maximum at the opening position, and it steadily reduces as more checkers are placed on the board.

Chapter 6

Conclusion

We have described in detail how to construct a state of the art, two player stochastic game tree search algorithm. We paid special attention to the construction of a heuristic evaluation function, and how it differs from an evaluation function that only needs to be applied by a minimax searcher.

We have demonstrated two improvements to the Star2 algorithm, which when combined reduce the number of nodes searched in the game of 5x5 Dice by 37% when looking 13 plies ahead. The first is the utilisation of transposition tables at chance nodes. The second enhancement augments the Star2 algorithm with a novel preliminary probing phase.

Furthermore, we have completed an easy to use testing framework which could prove useful to other people who want to experiment with stochastic search algorithms. We have listed some areas for future work below.

6.1 Future Work

More comprehensive results

It would be interesting to see performance numbers for games with different branching factors. The current experimental framework supports games of Dice on boards larger than 5, so there is

a lot of scope for extension. It is possible some search enhancements may only become practical when the branching factor at chance nodes is high. This is because the relative importance of each individual successor to the final value of a chance node decreases as the branching factor becomes higher.

It would also be interesting to see some results with other games. Carcassonne would make an ideal next target. This game is popular all over the world, and requires much more sophisticated play compared to Dice. A key difference to Dice is that the chance events in Carcassonne are not uniformly distributed.

Move ordering at chance nodes

There could be potential gains by heuristically ordering the stochastic events at chance nodes. We tried a couple of simple heuristics, but could not produce any evidence of statistically significant savings of search effort for 5x5 Dice. Perhaps the importance of move ordering would take on more significance if the branching factor at chance nodes was higher, or if the chance events were not uniformly distributed.

$\alpha\beta$-window adjustments

The Star family of algorithms work by requiring that each successor fall inside a dynamically computed window or directly produce a cutoff. At some nodes, we might strongly suspect that we are going to fall outside the $[\alpha\beta]$ interval. In these cases, it could prove worthwhile to try to prove that we must cutoff using inexact bound information, and only try to establish exact successor scores once we determine that the bound information is insufficient to cause a cutoff.

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 to find its exact score. If we can skip finding exact scores at some nodes, then we expect a performance improvement so long as we can reliably guess where to apply the windowing adjustments.

We have very recently obtained some promising preliminary results using a method based on this idea. There was no time to include it in this thesis, however we are looking to include it in an upcoming paper.

Forward pruning

There is lots of scope for trying theoretically unsound pruning methods at chance nodes. We did not examine any methods in this thesis, but there is lots of scope for work here.

One idea that could be promising is the *Lucky/Unlucky Roll* heuristic. If you find a position where even if you could pick the outcome of the dice roll, you still couldn't make it inside the $[\alpha, \beta]$ interval, then you return the relevant bound on the node's score. To determine whether this criteria holds, one could try using a depth $N - R$ search, where R is the depth reduction factor.

Parallel Search

It is possible to parallelise the negamax algorithm[5]. 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 lower than for min/max nodes.

Bibliography

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

[2] D. Breuker, J. Uiterwijk, and H. Herik. Replacement schemes for transposition tables.

[3] Dennis M. Breuker and Jos W.H.M. Uiterwijk. Transposition tables in computer chess.

[4] Thomas Hauk. Search in Trees with Chance Nodes. Master's thesis, University of Alberta, January 2004.

[5] B W; Nelson H L Hyatt, R M; Suter. A parallel alpha/beta tree searching algorithm. *Parallel Computing*, 10(3):299–208, 1989.

[6] D.E. Knuth and R.W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

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

[9] Gerald Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Comput.*, 6(2):215–219, 1994.

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

Appendix A

Appendix 1

A.1 Test Positions

The following is a listing of the 100 positions used to evaluate search efficiency.

```
5 4 b 1 - - - - - - - b - - r - r b - - - r - b - - - - - -
5 4 b 2 - - - - b r - - b r r - b - - r - - - - - - - b -
5 4 r 2 - - b - - b r b b r - - b r - r - r r - - b - b -
5 4 b 3 - - b - - r r - - - r r - - - r b - b - - - b b -
5 4 b 0 - - - - - r - - - - r - r - - - b - b - - - - b -
5 4 r 1 - - - - - - - b - b - - - b r - - - - r - - - - - -
5 4 b 0 - - - b - - r r b - - b r r - - r b b - - - - - -
5 4 r 3 - - - b - r r - - r - - b - - - - - - - - - b b -
5 4 b 3 - - - b - - r - - - r b b r - - r - b - - - - - -
5 4 r 4 - - - b - - - r - - - - b r - - b r - - - - - b -
5 4 b 2 - - - b - - - r - - r - b - - - - r b - - - - - -
5 4 r 0 - b b - - - r - - - - r b r - - - r b - - - - b -
5 4 b 4 - - - b - - r - - - - r b - - - - - - - - - - - -
5 4 b 4 - - - b - - - - - r - r - b r r r b b r - b - b -
5 4 b 4 - - - - - - - - - b r - b b r - r - r - - - - - b -
5 4 r 1 - - - b - r b - b - - - - r - - - - - - - - - - -
```

5 4 b 4 - - - - - - - - - b r - - r - - - - - - - - - - - b -

5 4 b 1 - - - - - - - r - b r - - - r - - - - - b - - - - b -

5 4 r 4 - - - b - - - r - - r - r b - - b - b - - - - - - -

5 4 r 3 - - - b - - r b b - - r - - - r r - b - - - - b -

5 4 r 2 - - - b - - r - - - - - - - - - - - - - - - - b -

5 4 r 4 - - - - - - - - - - - - r - r - - b - b - - - - b -

5 4 b 2 - - - - - - - - r - - - - r - - r b - b - - - - b -

5 4 r 1 - - - - - - - b - b - - r b r - - r - b - - - - - -

5 4 b 1 - - - - - - - r - - - r - b r - - - - - - - b - b -

5 4 r 1 - - - - - r b - b - - - r - - - - - - - - - - b -

5 4 r 1 - b - b - r r r b - - - b r - - - - - - - - - b - -

5 4 b 2 - - - b - - r - - r - - - - - r - b - b - - - - - -

5 4 r 3 - - - - - r r - b - - - - r - - - - b - - b - b -

5 4 r 3 r - b b - b r r b r - - r - - - b - b - - - - - -

5 4 b 0 - - b - - - r r - r - b b b r - r - - - - - b - -

5 4 b 1 - - - - - - - - - - - - - - - r - - - - - - - - b -

5 4 r 3 - - b - - - r - b r r r - b - r b b b r - b - - -

5 4 b 4 - - r - b r r r b - - b b - - - r r - - b b r b -

5 4 b 0 - - - - - r - b b - - r - - - - - - - - - - - - -

5 4 b 0 - - - b - - r - b - - - r r - - - - - - - - - b -

5 4 r 2 - - - b - - - r b - - - b b - - - r - - - - r - -

5 4 b 2 - - - b - - r b b - - b - r - - - - r - - - - r -

5 4 r 4 - b r - - - - b b - - - r r b r - r b - - - b - -

5 4 b 3 - - - - - r r b b - r - - r - r - - b - - b - b -

5 4 r 0 - - b r - r b b r - b - r r - r - - b - - b - b -

5 4 r 3 - - - b - - r - - r r b b - - - r b b - - - - - -

5 4 r 4 - r - - - b r r b r b b b r b r r b b b - r - - -

5 4 r 2 - - - b - - - - - - - - r - - - r - - b - - b - - -

5 4 r 4 - - - - - - - - r - - - b r - r r b - b - - b - b -

5 4 b 2 - - - b - - - - - b - - r r - - - - - - - - - - - -

5 4 r 3 - - - b - r - - b r - - r - - - b - b - - - - - -

5 4 b 2 - - - - - r r r b b - b r r r r b b b r - b - b -

5 4 r 2 - - b - - - - r b - r - r r - - r b b b r b - b -

5 4 b 1 - r r - b r r r b - - b b - - - r r - - b b r b b

5 4 b 2 - - - - - - - - - - r - b b r - r - - b - - - - - -

5 4 b 1 - b b b - r r - - - r b r r b r - - - - - r b b -

5 4 r 1 - b b - - r - - b r r b r r r r - b b - - b - b -

5 4 b 2 - - b b - r - - - r - - b r - - - r b - - - - - -

5 4 r 1 - b r b - - r - - r r - b - - r b b b - - - - - -

5 4 r 0 - - b b - - r - b r - - r - - - - - - - - - - b -

5 4 r 2 - b - b - r - r - - - r b - - r - - b - - b - - -

5 4 r 4 - - - - - - - - - r - - - r b - - - - - - - b - b -

5 4 b 0 - - - - - b r r b b - b r r - - - r b - - - - - -

5 4 b 3 - - - b - b r b b r r - - r - r - r b - - b - - -

5 4 b 3 - - - - - - - - - b - r - r b - - - - r - - - - b -

5 4 r 2 - - - - - - - b - b - r - - r - - - - b - - - - - -

5 4 r 1 - b - b - r - - b - r - - r - r b - r - - - b b -

5 4 r 1 - - - b - - r - - - r - - - - - b - b - - - - - -

5 4 b 4 - - b - - - - - - - r - - b r - - - r - - - - - b -

5 4 r 0 - - - - - - - r - b r r b r r - - b b b - - b - - -

5 4 r 1 - - - - - r b - b r - - b - r - b - - - - - - - -

5 4 r 1 - - - b - - r - b - r r b b - - - - - - - - - - -

5 4 b 2 - - - - - - - r - - r - - b r - - - - - - - - b b -

5 4 r 3 - - b b - r r r b b - - r - - - r - - - r b b b -

5 4 b 0 - - b b - - - - r r - - - r - - - - b - - - b r -

5 4 b 2 - - b - - - r b b - - r - - - r - - - - - - - - -

5 4 r 4 - - - - - r - r - r - b b - - - b - b - - - - - -

5 4 r 0 - - b b r r r b r b - b r r r - b r b b - b r b -

5 4 r 4 - - - - - - - r - b - - r b r - - r - b - - b - b -

5 4 r 2 - - - - - - - r - - - - - - - - - - - - b - - b - - -

5 4 b 4 - - - - b r r - b r r b b r r r r b b b - r b b -

5 4 r 0 - - - b - - - - b - - r - - - - - - - - - - - - -

5 4 b 1 - - - b - - b - b r r r b r - r b - - - - - - - -

5 4 b 0 - - b - - r b - b r r - - r - - - - b - - - - - -

5 4 b 1 - - - - - - - - - - r - r b b - r - - - - - - - b -

5 4 r 0 - b r - - - r b b b - - r - - - r r b - - b r b -

5 4 b 1 - - - b - - - r r - - - b b - r - - - - - - - - -

5 4 r 4 - - - - - - - - - - - r b b b - r - - - - - - - - -

5 4 b 2 - b b b - - b r b - - b r r r - r - r b - r b r -

5 4 b 0 - - - b - r - - - - - - - b r - - - - - - - - - -

5 4 b 2 - - b - - - - r - r - - b b r - - r - - - - - b -

5 4 r 4 - - - r - - - - - r r b b b r - r b r b - - b - - b

5 4 b 3 - - - - - - - r - b r - - b r - - - r - - - - b b -

5 4 b 1 - b - b - r - - b r - - r - r - - - - r - b - b -

5 4 b 1 - - - - - - - - - b r - - - r - r - - r - - b b b -

5 4 r 1 - - - b - - b b b r - - - r r r - - b - - b - r -

5 4 b 3 - - b - - r r r b - - - - - - - - - - b - - - - - -

5 4 b 3 - - b - - r r - - r r - - - - r - - b - - b b b -

5 4 r 3 - - - - - - - - - b - r r b r b r b - - - - - - b -

5 4 r 0 - - - b - - - - - - - - - r r - - - - b - - - - b -

5 4 b 0 - - - b - - - - - - - - r - r - - - - b - - - - - -

5 4 r 2 - - - - - - - r - b - - b r - - - - - - - - - - b -

5 4 b 4 - - b - - r - - r - r b b r - - - r - - - b - b -

5 4 b 4 - - - b - b r b b r r b r r r - - - b - - b - r -

5 4 r 3 - - - - - - - r - b - - - - - - - - - - - - - - - b -