University of Alberta

Search in Trees with Chance Nodes

by

Thomas Gordon Hauk 

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Spring 2004

Your file Votre référence
ISBN: 0-612-96482-5
Our file Notre référence
ISBN: 0-612-96482-5

Abstract

While much of the research done in heuristic search has concentrated on deterministic domains, not much work has been done to investigate search techniques in stochastic domains other than statistical sampling methods. When full search is required, Expectimax is often the algorithm of choice. However, Expectimax is a full-width search algorithm. A class of algorithms called *-Minimax were developed by Bruce Ballard to improve on Expectimax's runtime. They allow for cutoffs in trees with chance nodes similar to how Alpha-Beta allows for cutoffs in Minimax trees. This thesis presents new performance results for Expectimax, as well as Star1 and Star2 (the two main *-Minimax algorithms), in real-world domains. Ballard's work is verified and new insights into move ordering and probe successor selection are presented.

"After evaluating millions of pieces of data in the blink of an eye,
the Gamble-Tron 2000 says the [Superbowl] winner is...
Cincinnati by 200 points!? Why, you worthless hunk of junk!"
–Professor John Frink, The Simpsons (Epsiode 8F12)

Acknowledgements

I would like to thank many people, including:

My two supervisors, Jonathan Schaeffer and Michael Buro, for support and encouragement over the last eight months, for providing many insights into the search problem and posing enough interesting questions to keep me going for another year.

The GAMES research group at the U of A for providing much-needed nights of fun playing Settlers of Catan or Carcassonne, and for being a group of people I could talk to in the AI lab when I needed a break.

Dominique Parker for all the help troubleshooting my code and providing the answers to programming questions when I had them.

The GNU Backgammon team, especially Jørn Thyssen, Jim Segrave, Ian Shaw, and Gary Wong, for their help with understanding their code, getting a position database for testing, and giving insight into Gnubg's history.

The Instructional Support Group (better known as Labadmin) for hardware and software support, without whom my experimental results would not have been possible.

Joerg Richter, who made the backgammon font I used for describing the game.

And finally, last but not least, my mother Dr. Marie Hauk. Thanks for putting a roof over my head and food in my stomach while I worked all those months!

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Search in Real Life

The topic of search is not just one of the main areas of study in artificial intelligence (AI) today, but also one of the most important areas in computing science. Research to make computers search better and faster has led to improvements in various other areas of AI (better planning systems), telecommunications (more efficient network routing), industrial applications (more efficient scheduling systems) and even e-commerce (creating travel itineraries). Pathfinding is a research area that has a great deal of academic and industry interest (especially in games, robotics and military research). Search has also been used to improve constraint satisfaction systems.

Search is concerned with finding the best solution to a problem, commonly called an *optimal* solution. Finding optimal solutions to most problems is generally not trivial, because most "interesting" problems are at least NP-hard. With regards to the previous examples, we would like to find the smallest plan, the shortest path between two nodes, the cheapest schedule, a trip with the fewest number of connecting flights, or a solution to a constraint satisfaction problem in a "reasonable" amount of time. We want to find an answer, the best answer if possible, and we want to find it quickly.

Many real-life problems are also *stochastic*, meaning some steps are based on an element of uncertainty. For example, a person driving a car needs to not only pay attention to keeping the car on the road, but on various other factors around them like other cars, pedestrians and signal lights. Choices are often based on making assumptions about these factors. For example, we may not want to speed down a street if we notice children playing on the sidewalk up ahead, because we won't be able to stop quickly enough if they run on the road. At the same time, we won't just stop the car and wait for them to go inside before continuing on, because we might end up waiting for hours. While there is a chance the children will run onto the road, the chance is not likely, if they've been warned by their parents. So we'll probably drive down the road at a reasonable speed, while keeping one eye

on them. In other words, we need to balance the risks and rewards for doing actions based on possible future worlds.

A popular model for these types of stochastic domains is called a *Markov Decision Process*[22], or MDP. MDPs are graphs with states that are linked together by actions whose outcomes are deterministic. Being in a given state may give us a penalty or reward, which can also be based on the last action we took. The transition probabilities to another state are based only on the state we are currently in, forgetting everything else that happened in the past. This constraint is called the *Markov property.*

MDPs are often used in single-agent problems, where one wants to find the best policy for every given state. MDPs can be used for businesses to maximize sales, or in risk management in order to find optimal expenditures. For example, a retailer may classify their clients into two groups, people who spend a lot of money, and people who spend little money. Based on the classifications, the retailer can then consider whether or not they should incur the cost of sending that person a catalogue of their products, since the catalogue may encourage the client to spend more money. The retailer can then decide what the best catalogue-sending policy to use is, based on the risks (cost of printing and shipping) and rewards (extra client spending). MDPs can also be adapted for multi-player games, where states can define different rewards for different players, and players take turns choosing actions to make transitions between states.

1.2 Games in Research

Games are often used as a testbed for new search algorithms. Programs designed to play games usually make their decisions using searches in the problem *state space*, which is the set of all possible *states*, or possible configurations, of a game. Since games have been used as a testbed for search algorithms for almost as long as computing science has been around, most games used are generally *well-understood* domains. These days, it is rare to find a search problem that has not been explored in the context of a game, or a game which has not been investigated using search. Of course, a fringe benefit[1] of working with games as a research domain is that games are *fun.*

1.3 Games of Skill, Games of Chance

Games are usually classified as games of skill, or chance. There are many games which involve both skill and chance, but often simple games of chance do not involve much strategy. For example, chess is clearly a game of skill, but luck only factors into the equation when we hope that our opponent makes a mistake. On the other hand, games of chance like

[1]Some might say the *primary benefit!*

roulette offer very little opportunity to use skill, besides, perhaps, knowing when to quit losing. Games that involve chance usually involve dice or cards. Competitive card games combine skill and chance by requiring players to use strategic thinking, while they manage the uncertainly involved; since card decks are shuffled, the game's outcome is not certain. Since a player's cards tend to be hidden in card games, each player will have *imperfect information* about the game state. There are not many *perfect information* games which blend skill and chance – games where nothing is hidden, yet nothing is certain. Happily, there does exist a game that fits this description, a game that is both well-known and relatively popular: backgammon.

1.4 Contributions of this Thesis

Whenever chance is introduced in a domain, Expectimax is the usual algorithm of choice. While Expectimax is sound, it builds large trees in complex domains, and searching these trees deeply may require too much time. Instead of doing brute-force search in these domains, we can use techniques in order to prune away parts of the tree which are irrelevant. In 1983, Bruce Ballard developed a class of algorithms, called *-Minimax, which can be used to search state spaces in games with chance, but require less time per search than Expectimax. Ballard investigated the algorithms in an artificial domain at relatively shallow depths.

The main contributions of this thesis are as follows:

1. Re-implementation of Expectimax as well as the two main *-Minimax algorithms, Star1 and Star2, and their application to two real-world domains: a game called Dice (developed for this thesis), and the ancient game of backgammon,
2. Verification of Ballard's work,
3. Investigation of the relative performance of Expectimax, Star1 and Star2 when applied to Dice (at varying branching factors) and Dice and backgammon (at varying search depths),
4. Investigation of the relative performance of Star2 at various depth settings in Dice and backgammon tournaments,
5. Presentation of new insights into the performance of Star2 with various move ordering schemes as well as probe successor selection schemes.

Most importantly, this thesis will provide results at depths much deeper than Ballard was able to attempt, and in more complex domains than Ballard used, thanks to the improvements made in computing resources since the early 1980s.

1.5 Organization of Thesis

Chapter 2 presents an overview of games research, provides an introduction to the two most common algorithms used in two-player search (Minimax and Alpha-Beta), and summarizes some of the most common search enhancements. **Chapter 3** focuses on stochastic domains and the adaptation of Alpha-Beta to work when chance is introduced into the search space, including an explanation to the two *-Minimax algorithms, Star1 and Star2. **Chapter 4** focuses on applying *-Minimax to a game domain called Dice (invented for this thesis), to investigate how branching factor and search depth effect performance. **Chapter 5** explores the game of backgammon and its implementation using the GNU Backgammon codebase. **Chapter 6** summarizes observations and makes conclusions about the experimental data collected, as well as suggests some new avenues for further research.

Chapter 2

Background

2.1 Games as a Research Domain

When research is done with search algorithms, games are used often as the domain for testing new algorithms or improvements to old algorithms. Games are used because they are often well-understood domains and relatively "real-world". Games also have a good built-in performance measure: a win.[1]

There has been a rich history of games research in computing science over the last fifty years. While some games have been solved outright, other games are still proving to be difficult challenges for computers. This chapter will explore various games and at what level of skill computer programs of today have, and provide an overview of two important search algorithms and popular enhancements for them.

2.1.1 Solved

Some games have been solved, meaning that computers play perfectly. Since they do not make mistakes, solved games are usually no longer of much interest to researchers. They are still useful for developing new approaches to AI, because a perfect opponent is available as a performance metric.

Awari

Awari, also known as Mancala, is an ancient game from Africa involving pits and stones. It is a popular game (especially among children) because its rules are simple. While Awari's state space is not as large as most other games used in research, it has been enough of a factor in past years to prevent outright solving, so most Awari programs used search to find the best move. However, John Romein and Henri Bal, researchers in the Netherlands, were able to finally prove Awari as a draw with perfect play[21]. They used an impressive array of hardware to perform brute-force search, and were able to solve and store the game,

[1] We could also build programs that lose, but that would be much less fun.

starting from possible end positions, all the way back to the initial start.

Connect-4 and Gomoku

Both Connect-4 and Gomoku have similar rules: players take turns placing their stones on a grid (or in Connect-4's case, dropping them into a grid), and the first player to connect enough pieces (four for Connect-4 or five for Gomoku) wins the match. Both games have a fairly low degree of *decision complexity*, meaning most moves are irrelevant because of the structure of the game, or there aren't many move choices during play. For these two games, countering opponent threats is generally the most important part of the game. In 1988, Victor Allis solved Connect-4[1], using strategic rules to guarantee a win for the first player, and to guarantee at least a draw for the second player (if the first player does not start in the middle column). Allis solved Gomoku in 1993[17] using a threat-based search technique.

2.1.2 Super-human Play

Programs that play at a super-human level may not play a perfect game, but usually the game is complex enough that humans cannot hope to challenge programs in this domain any more.

Checkers

Checkers[2] is a popular game played by people of all ages around the world, using a 8×8 board. The first success in computer checkers was obtained by Arthur Samuel in 1959[23], using a simple learning algorithm to evaluate board positions. Currently, the Chinook program created by Jonathan Schaeffer and his team at the University of Alberta is the top program in the world. Chinook won the world checkers championship by beating the best human in 1994[25]. One of Chinook's weapons is its large end-game database (approximately 18 trillion positions and counting), which allows it to play a perfect game, once enough checkers are removed from the board. Checkers may soon move into the solved category.[26]

Othello

Othello is a popular territory capture game. The first computer Othello program was developed at Caltech in the late 1970s and the first computer tournament took place in 1979. Program skill has developed continuously since then, and in 1997 a program written by Michael Buro, Logistello, took on and routed the world champion at the time, Takeshi Murakami[9]. Logistello used machine learning techniques to create and tune an extremely complex and powerful evaluation function for Othello positions, and used a new, sophisticated search enhancement called ProbCut[8] to ignore irrelevant lines of play.

[2]Or draughts, if you are British.

Scrabble

Scrabble is a word game, where players use tiles with letters on them to score points. It requires not only a good vocabulary, but also a keen tactical sense. Computers can easily digest dictionaries, but strategic play is the hard part. Because of the randomness in which players draw new tiles, the brute-force approaches that are successful for chess, checkers and Othello cannot be used. One of the most successful computer Scrabble programs is Maven, written by Brian Sheppard[27], which beat a Scrabble grandmaster in a match in 1998. Maven has extensive dictionaries, and uses a "simulator" technique to perform searches by randomly "dealing" out tiles to each player and then playing through a few turns.

2.1.3 World Championship Level

Some programs play games at a world championship level, meaning only the best can hope to compete, but they are not without their flaws. Some games like chess are still complex enough for computers that human masters still have a reasonable chance. Other games like backgammon involve an element of luck that computers can model, but not control.

Chess

If there is one game that has captured the attention of AI researchers since the early days of computing, it is chess. Chess has been referred to as the *Drosophila*[3] of AI, the most popular game domain used in research. The most famous achievement in computer chess was the 1997 man-machine match between then-world champion Garry Kasparov, and Deep Blue, a massive custom-made chess computer designed by a team from IBM[20]. Kasparov won the first game but ultimately lost the match to Deep Blue, which was then promptly dismantled. As chess programs continue to get stronger and hardware gets faster and cheaper, beating them has become out of reach for all but the best players.

Backgammon

Backgammon is considered to be one of the oldest board games still played, perhaps the oldest of recorded history[29]. It is also a *non-deterministic* game because there is a dice roll at the beginning of each turn to determine what legal moves a player has. As such, search algorithms that have been successful for deterministic games (like chess) cannot be directly applied. By the late 1980s, computer backgammon was mired at an amateur level. In 1990, Gerald Tesauro applied a new machine learning technique, reinforcement learning, to create a backgammon program that could learn the game solely by self-play[33]. The resulting program, TD-Gammon, has become one of the top backgammon programs in the world, and is considered to be as strong as any human grandmaster. In recent years, several

[3]Drosophila, or the common fruit fly, is used in biology to study the effects of inheritance.

new projects have sprung up based on Tesauro's work, including GNU Backgammon[2], Jellyfish[3], and Snowie[4], all of which are considered to be near the same performance level as TD-Gammon.

2.1.4 Strong Play

Often, there are programs that reach a strong level of play, but not expert, because the game may have complex rules, simple rules but a high degree of decision complexity, require long-term abstract planning, hide information about the game state from the players, or involve outright deception.

Poker

Poker is a game played by two or more people. It has a multitude of variants, but most feature the same basic play, where one or more rounds are played which consist of cards being dealt followed by a round of betting. In most variants, betting is the most important part of the game, because you do not necessarily need a strong hand to win – you just need to be the last person standing. Poker research has concentrated on two different schools of thought. The first has adopted a mathematical approach, using game theory to discover optimal strategies for play for simplified poker variants[7]. The second school of thought has been to tackle a complicated variant of poker head-on, and develop a program with all the necessary skills at once – skills like opponent modeling and developing betting strategies. One of the most successful programs that has adopted this approach is the Poki program[7], developed at the University of Alberta by a number of researchers.

Bridge

Bridge is a popular four-player card game, played by two teams of two players. It has two phases: the bidding phase, in which each team wagers how many rounds ("tricks") they can win, and the playing phase, which has thirteen rounds of play. Both phases are a challenge for computers. The bidding phase is difficult because players can only communicate via their bids. The playing phase is difficult because cards are randomly distributed and not known until they are played. Bidding strategies can be subtle, and playing strategies based on precise timing, which makes the game difficult for computers to play. One of the current top bridge computer programs is Matt Ginsberg's GIB[11].

2.1.5 Weak Play

Computer programs have a long way to go in some game domains. Games that involve social interaction between players, explosive search spaces, or require long-term planning remain difficult domains.

Go

Go is an ancient game, and the focus of much research over the last thirty years. It is a game of creating territories played on $N \times N$ boards, where players place stones on alternate turns. The major problem with Go is the difficulty in designing strong evaluation functions. Selecting a good move is therefore not a trivial task. While humans are able to adapt strategies and plan in the long-term, this is still a difficult task for computers. Although some major strides have been made in the last few years as some computer Go programs (such as Handtalk and KCC Igo) are beginning to approach competent levels of play, they can still be beat by humans of moderate skill[19].

Settlers of Catan

A German board game that has become a world-wide hit, Settlers of Catan is a 2-4 player game of colonization of a hexagon-shaped island. What makes the game so interesting is that players are free to (and encouraged to) trade resources they gather among themselves. Although only one person can win the game, trading is a vital aspect because it is a win-win situation for both players involved in the trade. It is nearly impossible to win without interacting. And while computers may find it "easy" to play a game like chess, much less progress has been made in developing computer agents capable of negotiation and opponent modeling, abilities that humans seem to have naturally or learn easily. The Columbus project lead by Robert Thomas and Kristian Hammond at Northwestern University[34] has developed a free game server for Settlers, as well as their own Settlers agent, playable by humans. The agent usually fares poorly in games with human players because of its limited ability to communicate.

2.1.6 The Future

Even with all the progress that has been made in AI games research, many computer programs can still be beaten. For example, research still has not been able to create the "perfect" chess program, simply because the state space for chess – the number of different possible positions in chess – is too large. A wide variety of different algorithms and algorithm extensions have been used in computer game programs, proving that games can be excellent domains for new ideas in search, knowledge, and machine learning. While some games have been solved, many interesting games continue to be huge challenges, because the size of state spaces or high degree of decision complexity in most games remains intractable to brute-force search. Games with high degrees of complexity (like Go) or even modest degrees of player interaction (like Settlers of Catan) will remain open problems for years to come.

2.2 Heuristic Search

2.2.1 Game Trees

When a computer program wants to decide on which move to play in a deterministic game, it will typically build a *game tree* (see Figure 2.1). A game tree is a directed graph where the root node represents the current game *state*, and its successors represent all possible game states after playing a move, which are represented by edges between states. States are *terminal* when they represent a game state that is a win, a loss or a draw. States at the bottom of the tree are called *leaf* nodes. All other states are *internal* to the tree. In a single-player game (like a puzzle), a game tree is usually built in order to find an optimal solution; in other words, to find the minimal sequence of moves that leads the player to the best terminal state. In multi-player games, game trees usually consist of alternating levels of nodes, where each level of nodes is associated with the player whose turn it is to move. In such games the computer must not only consider its possible options, but also consider what options its opponent(s) have as well.

2.2.2 The Search Problem

Most interesting problems have state spaces which are exponentially large in the branching factor. Using exhaustive search techniques on these domains – going through the space in some sequential order and checking each state in turn – is not a reasonable method to find a solution, because it would take too long to find any solution, even sub-optimal ones. While the intractability of such large state spaces may preclude a brute-force algorithm from finding a solution quickly, we can still speed up the process considerably. Instead of iterating through the entire state space, we need only search the part of the state space relevant to finding the answer. The current state can be considered the root of a tree, and we can generate successors for the root by going through each possible action that modifies the state. For example, a node located two steps from the root would represent a state separated from the root by two actions. Any state representing a solution would be a terminal node. The general method, then, is to continue adding to our tree starting from the root until we find a path that leads to the best solution. We will use an indicator called a *heuristic* to help guide the search process.

One-Player Domains

The most common heuristic search algorithm, which has spawned many variations, is the A* algorithm[13]. A* requires the ability to put the entire state space it may explore in memory, and guarantees to perform asymptotically less than or equal to the number of *node expansions* of any other algorithm, when finding the optimal solution to a problem. This quality makes it extremely appealing, but its memory requirements still outstrip even

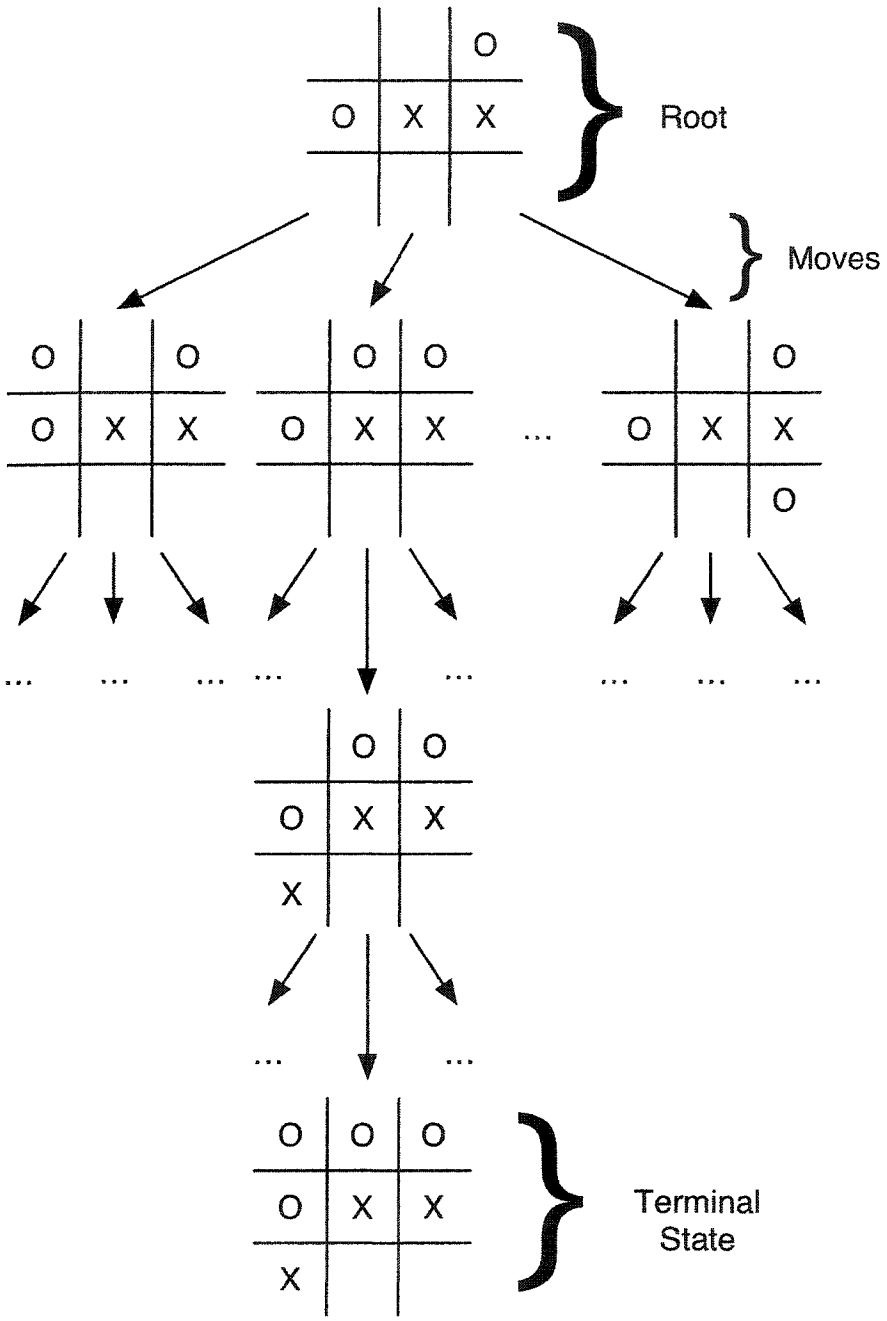

Figure 2.1: A game tree for tic-tac-toe

the latest computers on "difficult" problems. In response to this problem comes the IDA* algorithm[16], which still guarantees an optimal answer, but uses *iterative deepening* search in order to minimize memory requirements.

One of the most popular application domains used in heuristic search research are sliding-tile puzzles. These puzzles have a number of square tiles mixed up in a square area, with one tile missing (the "blank" square). They are solved when each tile is placed in numerical order, with the blank at the end. Actions can be performed by sliding a tile adjacent to the blank into the blank space, thus shuffling tiles around. A* uses a heuristic to score every non-terminal state. The greater the score, the greater the estimated cost to reach a goal state from that node. A* is a *best-first search* algorithm, because it always searches nodes in order of most- to least-likely to succeed.

Two-Player Domains

We cannot apply A* to two-player games. While actions can be controlled in single-agent games, in two-player games we can only control one of the sides of play. The opponent will generally be out to win just as we are out to win, and will not make moves that raise our chances of success, or lower their chances of winning. This adversarial setup requires a different approach; instead of finding a sequence of moves to lead us from a starting position to a solution, our job is to decide what move we should make on our turn. If we cannot find a move to lead us to a winning state, we want a move leading us to the "best" state possible. We will use a heuristic to help us make our decision.

In two-player domains as in single-player domains, heuristics come in the form of *evaluation functions*, which are functions that take a state (usually a leaf node) and map it to a value. Most evaluation functions will map a terminal node to an extreme value: a large positive number for a winning position, and a large negative number for a losing position. Non-terminal states are usually given a value based on their *utility* or benefit for the player to move, such that an even game may be mapped to a 0, or a state that is nearly a win is mapped to a large positive number. The exact numbers are always implementation-specific. For example, a backgammon program may simply map a state to a real value between -1 and +1, to represent the position's *equity* to the player to move; an Othello program might simply map the state to the material difference in discs between the two players. In practice, good evaluation functions will map states to a "reasonable" range of values. If the range is too small, then a program may end up choosing bad moves over average moves simply because it is not able to tell the difference. Program performance is directly related to the quality of the evaluation function, which is often the component of a computer game that takes the most work to get gradual improvement.

While we are concerned with the depth in the tree at which we find a solution in single-

player games (the less deep we have to search, the quicker we find the answer), in multi-player games the search depth is just as important, because of the adversarial nature of games. Searching to depth=1 only considers the immediate states which follow from the root; in other words, the player to move is only looking ahead one step (one *ply*) to see what would happen immediately after playing a move. Search a step deeper, and now the opponent's replies are considered. If we continue this process of considering replies to replies to replies, we "look into the future". We can avoid traps set up by our opponent, and lead our opponent into traps. We can find lines of play that lead us to wins. Deep search in games allows us to be more informed in our decision making, and hopefully increases our chances of winning.

2.3 Minimax

Searching depths greater than one ply requires that we take the opponent's choices into consideration. When we search, we will make the assumption that the opponent will always make the best move possible on their turn (based on the same evaluation function as we use for our turns), one that lowers our score as much as possible (and improves their score as much as possible). In game tree terms, we will have an alternating sequence of levels where it is our turn at even depths (with the root at depth 0), and the opponent's turn at odd depths. Since we are trying to *maximize* our score, and our opponent can be seen as trying to *minimize* our score, these levels are often called *Max* and *Min* levels, respectively. At every Max node, the move will be made to a successor with greatest value, and at Min nodes a successor with smallest value will be chosen. Once all the successors of a node have been evaluated, we can evaluate that node (and *back up* the value to its parent). This is called the *Minimax* algorithm.

The Minimax algorithm is summarized in Figure 2.2, which makes use of (1) a `terminal` function that returns true if a given state is terminal, (2) an `evaluate` function that returns the heuristic evaluation of a state, (3) a `numSuccessors` function that returns the number of successors a state has, and (4) a `successor` function that returns a new state after a move has been applied. `is_max_node` is a flag that is toggled between recursive calls to indicate if the node is Max or Min.

Consider the example in Figure 2.3 to see how Minimax works, with a 3-ply search. We start at the root. The root is not a terminal node (in fact, it is a Max node), so we will need to consider each of the root's successors (each of which is a Min node). We look at the first successor, A. It is not a terminal node, so we recurse and look at its successors. Since they are not terminal, we go another step deeper. The recursion stops at this point in this example, since we are only interested in looking ahead three steps. Both of these leftmost leaves (J and K) are scored using the evaluation function, and the parent, D (a Max node) returns a value of 5, which is the largest of the two leaf values. The next Max

```
int Minimax(Board board, int depth, int is_max_node) {
   /* Leaf node check */
   if(terminal(board) || depth == 0) return (evaluate(board));

   N = numSuccessors(board);
   if(is_max_node) {
      /* Maximize */
      score = -INFINITY;
      for(i = 1; i <= N; i++) {
         v = Minimax(successor(board,i), depth-1, !is_max_node);
         if(v > score) score = v;
      }
   } else {
      /* Minimize */
      score = INFINITY;
      for(i = 1; i <= N; i++) {
         v = Minimax(successor(board,i), depth-1, !is_max_node);
         if(v < score) score = v;
      }
   }

   /* Back up value */
   return (score);
}
```

Figure 2.2: The Minimax algorithm

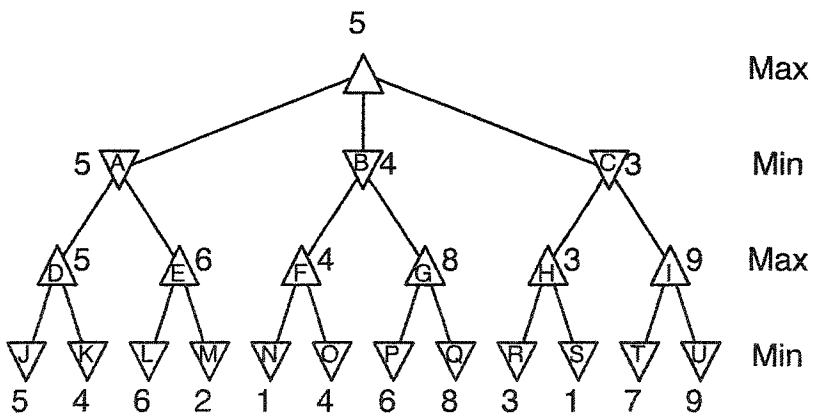

Figure 2.3: A Minimax tree

node to its left (E) is scored in the same way, and returns a value of 6. Now the value of A can be determined, by choosing the smallest value of its two successors. Since 5 is less than 6, this Min node returns a value of 5 to the root. Each of the root's successors is scored the same way. When all of the root's successors have been expanded, we can score the root. Since the root is a Max node, we choose the successor who leads to the greatest value. The move corresponding to reaching that successor is thus the move chosen. In our example, we choose the move that leads to the first of the root's successors, since the first successor has a higher value (5) than the other two successors (3 and 4).

2.4 Alpha-Beta

Minimax is the brute-force equivalent to exhaustive search in two-player domains, because Minimax will search every single possible node. But searching every node in the state space is not always necessary. Consider the previous example in Figure 2.3. By the time we start searching A's second successor E, we can use some assumptions to save us searching E in its entirety. Once we know the value of the first of A's successors, we obtain an *upper bound* on the score this Min node will have. In this case, we know A will have a value no greater than 5. We obtain this bound from the previous assumption that our opponent will always try and lead us toward the worst possible outcome. When we search L, and get a value of 6 for that leaf, we know that its parent E (a Max node) can have a value no less than 6. But since the opponent can already keep us to a score of 5, we know they will not give us an opportunity to get 6 or more. So we need not bother searching the second leaf (M) since the value for E can only get *better*. We can then "cutoff" search at E, since further search below the Max node is guaranteed not to change the outcome of the search.

This opportunity to obtain mathematically proven bounds on Max nodes and Min nodes allows us to have cutoffs, and therefore to search less nodes to get the same answer, leads us to the Alpha-Beta algorithm[15], summarized in Figure 2.4. At each step in Alpha-Beta we include lower and upper bound values (called alpha and beta, respectively) when we expand a node. This information is updated when a node's successors return values that allow us to change those bounds and adjust the *window* of search. In the case of a Max node, we can increase the alpha value as we search children with better scores. Min nodes will similarly adjust the beta value of the window when children with worse scores are seen. Whenever the two bounds of the window meet, we can stop search knowing that the node is not better than another node in the tree at the same level that we have already seen (it could be equally good, but in that case we can just safely go with the other node and stop searching this node). We saw before how the leaf with value 2 does not need to be searched. Figure 2.5 shows the result of Alpha-Beta applied to the tree from Figure 2.3. 5 of the 12 nodes are proved irrelevant to the search, which means we only searched about half the

```
int AlphaBeta(Board board, int alpha, int beta,
              int depth, int is_max_node) {
   if(terminal(board) || depth == 0) return (evaluate(board));

   N = numSuccessors(board);
   if(is_max_node) {
      score = -INFINITY;
      for(i = 1; i <= N; i++) {
         v = AlphaBeta(successor(board,i), alpha, beta,
                       depth-1, !is_max_node);
         if(v > score) score = v;
         if(score > alpha) alpha = score;
         if(alpha >= beta) return (score);
      }
   } else {
      score = INFINITY;
      for(i = 1; i <= N; i++) {
         v = AlphaBeta(successor(board,i), alpha, beta,
                       depth-1, !is_max_node);
         if(v < score) score = v;
         if(score < beta) beta = score;
         if(beta <= alpha) return (score);
      }
   }

   return (score);
}
```

Figure 2.4: The Alpha-Beta algorithm

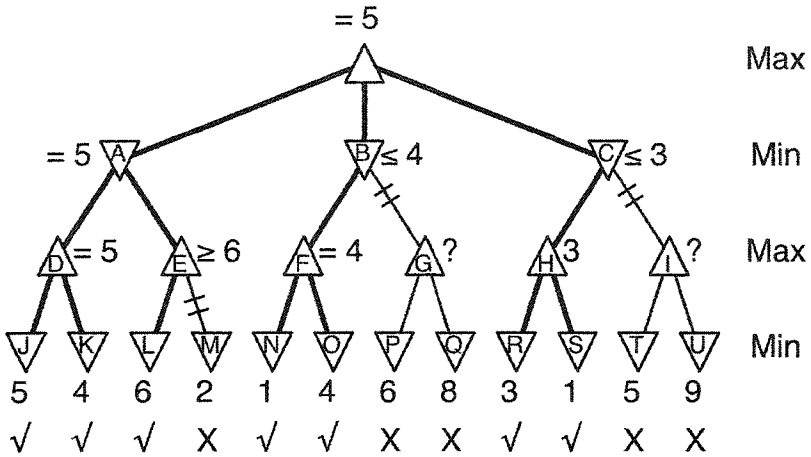

Figure 2.5: An Alpha-Beta tree

nodes as Minimax would have, but we got the same result.

The best case node expansions (a count of the number of nodes seen) for Alpha-Beta is $B^{\frac{D}{2}}$, where B is the branching factor and D the search depth. Alpha-Beta's worst case time requirement is B^D, which is the same as Minimax's requirement. Not all nodes in the tree are capable of cutoffs, however. An alternative way to view a Minimax tree is to view nodes as *All* nodes or *Cut* nodes. An All node requires that all of its children be searched, but a Cut node can generate cutoffs, sometimes as quick as after searching its first child. If every Cut node of a game tree generates a cutoff after searching its first successor, then the tree is reduced to a size of $B^{\frac{D}{2}}$, which is how the best case is derived. This reduction (the square root of the size of the original tree) is a substantial improvement over the worst case, which means we can search twice as deep as Minimax with the same number of node expansions. Counting node expansions is the most common metric for evaluating search algorithms, since node expansions is directly correlated to search time.

2.5 Search Enhancements

Alpha-Beta's success is not just related to its ability to generate cutoffs, but also comes from the many enhancements built onto it over the years. Speeding up search is usually done by reducing the number of nodes searched. We can accomplish this by being smarter about what order we search successors in, and we can also save ourselves time by storing information about completed searches in case we need to search the same node twice. Three important search enhancements are *move ordering*, *memory-assisted search* and *iterative deepening*.

2.5.1 Move Ordering

We can try to reduce the number of nodes we search by making a better evaluation function, but we can also reduce nodes by lowering the effective branching factor of the tree, since Alpha-Beta is dependent on searching the best move of a node first. One way to do that is to order the successors of a node by most- to least-favourable; by doing so we hope to raise our chances of getting a cutoff by getting higher values for a node sooner rather than later. This technique is often referred to as move ordering.[24] Move ordering is often done for every node that generates moves. Moves can be ordered using the usual evaluation function, but sometimes a different heuristic may be used for speed reasons.

Move ordering helps us get closer to the best case node expansions for Alpha-Beta, since we are trying to always search the best child of a node first. Even if move ordering does not generate immediate cutoffs at cut nodes, we will usually see a significant reduction in node expansions.

2.5.2 Memory-Assisted Search

Alpha-Beta can spend much of its time re-expanding nodes that it has already seen. As it is often the case with search trees, they are not trees at all, but directed acyclic graphs (DAGs), because of transpositions of states within a single search (this happens when different sequences of moves result in the same state). We would like our heuristic search algorithms to "remember" as much as possible about previous searches, to make future searches faster, by eliminating the need to search where one has searched before. This technique is referred to as *Memory-Assisted Search.*

One of the most common kinds of Memory-Assisted Search is the use of a *transposition table* (TT).[12] A TT is normally implemented as a hash table, because we want operations on it to be as fast as possible to minimize the overhead associated with reading and writing to the table. TT entries typically store an encoded state along with its last known heuristic value and the depth of search at which that value was obtained. TT entries also often contain the alpha and beta cutoff values (when used in an Alpha-Beta setting), as a method of "tightening" search windows and allowing for quicker cutoffs, and thus less search. The information stored in a TT is usually compressed to be as small as possible, to maximize memory usage. The TT is used before a state is expanded (TT lookup) to check to see if information about the state exists. If a state exists in the TT and its TT entry was searched to the same or greater depth than the current depth, we can use the TT value to either eliminate further search at this state, or at least narrow the search window for the state. Whenever we are about to return a value in a search, we store the state and its value in the TT. Sometimes two states will map to the same location in the hash table (a *collision*), at which point we need to decide whether or not to overwrite a value in the TT. There are

several techniques for dealing with this, but usually we only overwrite if the new entry is for a deeper search.

Transposition tables can store more than just searched values. We can also store the best move searched for a state in its TT entry. Then if we meet this state again, we can score it the highest of all successors when we perform move ordering. In this way we hope that the best move from a shallower search is the best move for a deeper search, and get a quick cutoff.

2.5.3 Iterative Deepening

Iterative search performs successive searches, starting each time from the same root, incrementing a depth parameter each time it starts over. Each repetition is called an *iteration.* Iterative deepening is a form of iterative search, where the nominal search depth starts at some small value and then is incremented after each search terminates. Iterative deepening is usually a component added to increase the effectiveness of memory-assisted search, as well as to improve move ordering.

Move ordering is especially effective when done at all nodes in conjunction with iterative deepening; before an iteration, the root's children can be re-ordered to reflect the scoring from the previous iteration. Since the order of children between searches is usually highly correlated, large portions of the tree can be potentially cut off quickly from the root.

2.6 Importance of Deep Search

In the multi-player setting, the deeper a program can search, the more likely it is to beat a program that searches shallower. In fact, a difference of only a single ply between two programs can have a substantial difference in performance[35].

Deep search also helps programs deal with the so-called *Horizon Effect*[28], where a program foresees a bad situation (good for the opponent), but makes (bad) moves in order to forestall the inevitable; in other words, the bad situation is "pushed over the horizon", where it cannot be seen anymore. The problem is that all searches will suffer from this effect unless they are able to reach a terminal state, or extend searches over the horizon to a "stable" state (which is done in *quiescence search*). The deeper a program can search, the better its chances to avoid the negative effects of the horizon.

Another reason why deep search is so important is that it can often overcome limitations in inherent human knowledge that is used by the program. While programming such knowledge to direct search (or choosing moves altogether) may be beneficial, it is in fact a dangerous thing to use. The explanation is that any knowledge one can put in a program about a game will ultimately be incomplete, since no one can really know everything about an "interesting" game. Added knowledge may also slow down a program because of

increased calculations. Deeper search may also dampen the effect of errors in a heuristic evaluation function.

Chapter 3

Search in Stochastic Domains

3.1 Games with Chance

So far, the discussion of search has centred around *deterministic* games, where an action is guaranteed to result in a specific state. Most games that have been extensively studied – such as checkers, chess, tic-tac-toe – fall into this category. However, games as simple as snakes and ladders[1] do not fall into this category, because they include an element of chance. The element of chance is often based on the roll of a die (or dice). Chance events are also present in card games like Poker, where one cannot be certain what card one will draw. Poker is also an *imperfect information* game, since opponents will *hide* their cards, and therefore hide information from the player. This thesis will only concern itself with *perfect information* games, where the entire state is known to both players, although the states that follow may not be deterministic.

With the addition of chance events, we need to add a new kind of node to our game tree: a chance node. A chance node will have successor states like Min or Max nodes, but each successor is associated with a probability of that state being reached. For example, a chance node in a game involving a single die would have six successor nodes below it, each representing the state of the game after one of the possible rolls of the die, and each reachable with the same probability of $\frac{1}{6}$.

The element of chance completely changes the landscape that search algorithms work on. In games of chance, we cannot say for certain what set of legal moves the opponent will have available on their turn, so we cannot be certain to avoid certain outcomes. One method used to simulate stochasticity, but remove it from the game tree so it can be searched using standard methods, is called *statistical sampling*, also referred to as *rollouts*. The method involves repeated trials where the outcome of the chance events are randomly decided before search begins, and then search is run normally on the resulting tree. Since each chance event only has one successor, they just become intermediary nodes in the tree. For games that

[1] Or chutes and ladders, if you are British.

involve dice, the chance events (the rolls, hence the term rollout) can be determined in advance or on-demand when a chance node is met in the tree. In order to get a good statistical sample, the number of trials must be high enough (for backgammon, this is often in the tens or hundreds of thousands of trials) in order to approximate the true distribution. While rollouts will remain a popular method used in search with games with chance, this thesis will concentrate on full search methods.

Previously, we saw how Minimax search worked in deterministic domains, and how Alpha-Beta improved on Minimax. The introduction of chance nodes means that we can no longer directly apply either algorithm to games of chance, since we cannot use Alpha-Beta windows with chance nodes as we would with Max or Min nodes, because Alpha-Beta cutoffs are based on the assumption that the game is deterministic. Chance nodes act as intermediaries, by specifying the state the game will take before a choice of actions becomes available. Before we can search trees with chance nodes, we have to figure out how to handle them.

3.2 Expectimax

The baseline algorithm for trees with chance nodes analogous to Minimax is the Expectimax algorithm[18]. Just like Minimax, Expectimax is a full-width, brute-force algorithm. Expectimax behaves exactly like Minimax except it adds an extra component for dealing with chance nodes (in addition to Min or Max nodes). At chance nodes, the heuristic value of the node (or Expectimax value) is equal to the weighted sum of the heuristic values of its successors. For a state s, its Expectimax value is calculated with the function

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

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

Figure 3.1 summarizes the Expectimax algorithm, which makes use of three new functions: (1) a `numChanceEvents` function to specify how many different values the chance event can take, (2) an `applyChanceEvent` function to apply the chance event to the state, (3) an `eventProb` function to determine the probability of the chance event taking that value, and (4) a `search` function that calls the appropriate function depending on the type of node that follows the chance node. Since most regular games have the same chance events at every chance node, `numChanceEvents` can be hard-coded as an integer, and eventProb can be replaced by a static lookup table. For games where chance nodes alternate with player turns, we can use the same Minimax algorithm from Figure 2.2, with the modification that Minimax's recursive call uses Expectimax instead of itself. We also use floating

```
float Expectimax(Board board, int depth, int is_max_node) {
   if(terminal(board) || depth == 0) return (evaluate(board));

   N = numChanceEvents(board);
   sum = 0;
   for(i = 1; i <= N; i++) {
      succ = applyChanceEvent(board,i);
      sum += eventProb(board,i) *
             search(succ, depth-1, is_max_node);
   }

   return (sum);
}
```

Figure 3.1: The Expectimax algorithm

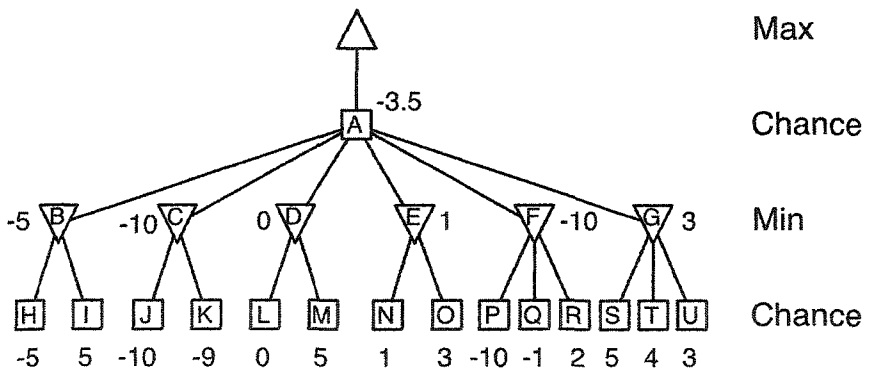

Figure 3.2: An Expectimax tree

point numbers instead of integers now for return values, since probabilities are real numbers and the sum may have a fractional component.

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

3.3 *-Minimax

As we have seen, Minimax is a sound algorithm, but its worst-case run time is far too slow for any interesting problem. We would like to obtain cutoffs in trees with chance nodes just like Alpha-Beta enables cutoffs in Minimax trees. We will need a new technique for finding valid cutoffs at chance nodes. At Min and Max nodes, we can use the same methods for cutoffs as Alpha-Beta uses, since we have not changed the definition of Min or Max nodes, we have just added chance nodes.

Bruce Ballard was the first to develop a technique, called *-Minimax, for enabling chance node cutoffs[5]. He proposed two versions of his algorithm, called Star1 and Star2. He also further refined the second to handle more general cases and have parameters to control functionality, and called the new version Star2.5. All the experiments that Ballard performed were in a rather abstract domain. He did not use a real domain, such as backgammon, to validate his results. In fact, Ballard's work seems to have been almost forgotten in the AI community in the last twenty years, which is truly unfortunate.

3.4 Obtaining Cutoffs

The basic idea of Expectimax is sound, but slow. Just as we can derive a strategy for obtaining cutoffs in Minimax to obtain Alpha-Beta, so too can we derive a strategy for obtaining cutoffs in Expectimax. Since there are three different types of nodes in a game tree for games with chance, there are three cases we need to consider for cutoffs. Since Max and Min nodes work the same way in trees with chance nodes as they do in trees without chance nodes, we get the cutoff strategies for those nodes for "free". All we need to concern ourselves with are chance nodes. If we pass alpha and beta values to chance nodes as we do Min and Max nodes, and we pass alpha and beta values from chance nodes to Min and

Max nodes, all that is left to consider is exactly what values we can pass, and how they will be used.

In the first case, chance nodes can have a search window just like Min and Max nodes, using alpha and beta values to determine if further search below the node is relevant. However, these alpha and beta values cannot be used just like they are used in Min or Max nodes, because the child of a chance node cannot be chosen deterministically (unless there is only one child, but that is an atypical case). We can obtain a cutoff, however, if the Expectimax value of a chance node falls outside the alpha-beta window. The problem is that we cannot know the exact Expectimax value of a chance node before we search all of its children.

However, if we know the lower and upper bounds of the range of values leaf nodes can take (called L and U respectively), we can determine bounds on the value of a chance node based on the worst-case conditions for both the alpha and beta values.

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

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

and the upper bound is equal to

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

So now we can figure out in what range the Expectimax value for a chance node must lie. We can use this range to help us generate cutoffs. Recall that the chance node itself was passed alpha and beta values. We can cut off our search if the lower bound of the Expectimax range for the chance node ever exceeds or equals beta,

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

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

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

where $(V_1 + \ldots + V_{i-1})$ are the accurate values for the first $i-1$ children of a node, V_i is the value for current node being searched, and $(U \times (N - i))$ and $(L \times (N - i))$ represent

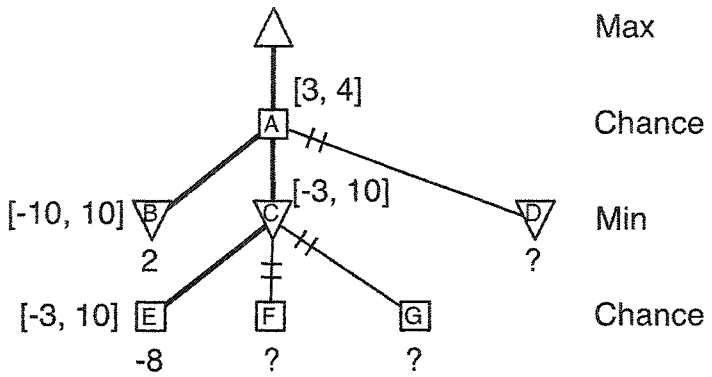

Figure 3.3: Fragment of a *-Minimax tree

the worst-case assumptions for the values of the remaining nodes. In either equation, we can solve for V_i, and use the value as either an alpha or a beta value for the next child.

Take the following example shown in Figure 3.3, where heuristic values range from $L = -10$ to $U = 10$, inclusive. The top-most chance node, A, is entered with a window of alpha=3 and beta=4 (we will write this as [3, 4]). Because we have not searched any of its children yet, we know its value lies in the range [-10, 10], and the alpha and beta values for the first child are equal to $\frac{1}{3}(3 \times L) = L$ and $\frac{1}{3}(3 \times U) = U$, which is also [-10, 10]. Assume that the first child (B) is searched and a value of 2 is returned. We now know the Expectimax range for the chance node is between $\frac{1}{3}(2 + 2 \times L) = \frac{1}{3}(-18) = -6$ and $\frac{1}{3}(2 + 2 \times U) = \frac{1}{3}(22) = 7\frac{1}{3}$. Since -6 is not greater than 4 and $7\frac{1}{3}$ is not less than 3, this child did not create a cutoff. Before we search the next child, we need to recalculate the alpha and beta values we want to pass down to it:

$$\frac{1}{3}(2 + V_i + (1) \times L) \geq beta \Rightarrow V_i \geq 20$$

$$\frac{1}{3}(2 + V_i + (1) \times U) \leq alpha \Rightarrow V_i \leq -3$$

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

Since the upper bound on a leaf node is 10, we will pass a window of [-3, 10] to the next child, C. Assume the next node searched at the bottom, E, has a value of -8. This will trigger a cutoff at C, because -8 lies outside the lower bound of the window (which is -3). The cutoff at C will also trigger a cutoff at the topmost chance node A. In fact, this could also trigger further cutoffs along this branch all the way up to the root; the possibility for two or more cutoffs to occur without intervening leaf searches is unique to trees with chance nodes, and not found in typical Minimax trees[5].

3.5 Star1

When we translate the ability to obtain chance node cutoffs into a procedural representation, we end up with Star1, Ballard's first version of the *-Minimax algorithm. Recall if L represents the lowest value a state can be given, and U the largest value a state can have, then we end up with a cutoff if

$$\frac{(V_1 + \ldots + V_{i-1}) + V_i + U \times (N - i)}{N} \leq alpha \tag{3.1}$$

or

$$\frac{(V_1 + \ldots + V_{i-1}) + V_i + L \times (N - i)}{N} \geq beta \tag{3.2}$$

Rearranging these equations, we determine the alpha value for the ith successor, A_i with

$$A_i = N \times alpha - (V_1 + \ldots + V_{i-1}) - U \times (N - i) \tag{3.3}$$

and the beta value for the ith successor, B_i with

$$B_i = N \times beta - (V_1 + \ldots + V_{i-1}) - L \times (N - i) \tag{3.4}$$

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

$$A_1 = N \times (alpha - U) + U$$

$$B_1 = N \times (beta - L) + L$$

and updating them with

$$A_{i+1} = A_i + U - V_i$$

$$B_{i+1} = B_i + L - V_i$$

where $i = 2 \ldots N$.

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

Figure 3.4 shows the resulting Star1 algorithm. The algorithm makes use of (1) a `terminal` function that returns true if a given state is terminal, (2) an `evaluate` function that returns the heuristic evaluation of a state (the evaluation function), (3) a `numSuccessors` function that returns the number of successors a state has, (4) a `successor` function that returns a new state, and (5) a `search` function which calls the appropriate function, either

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

Figure 3.4: The Star1 algorithm, adapted from [5]

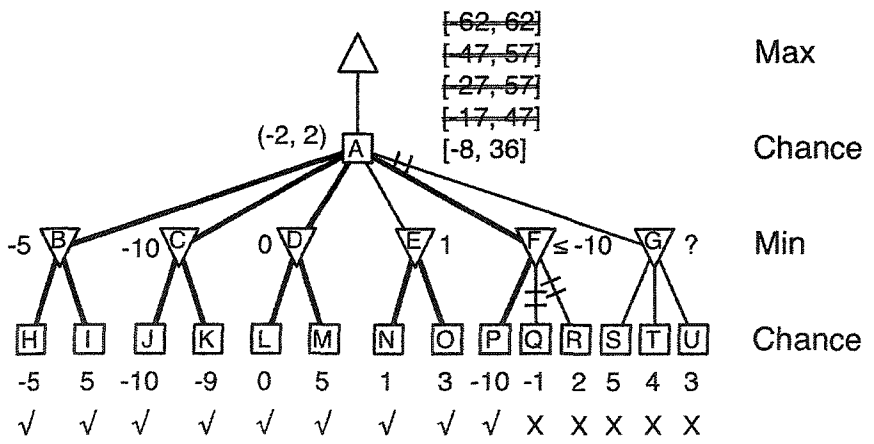

Figure 3.5: A Star1 tree

Star1 for a chance node or Alpha-Beta for a Min or Max node. This implementation assumes all values for the chance event have equal probability.

An example of Star1 cutoffs is shown in Figure 3.5. The uppermost chance node is initially passed bounds of [-2, 2]. The initial value for A is equal to $N \times (alpha - U) + U = 6 \times (-2 - 10) + 10 = -72 + 10 = -62$ and B is equal to $N \times (beta - L) + L = 6 \times (2 + 10) - 10 = 72 - 10 = 62$. After searching the root's first successor, the A and B values are adjusted for the second successor (C), where A becomes $-62 + 10 + 5 = -47$ and B becomes $62 - 10 + 5 = 57$. As we continue to search the children of the root sequentially, we can see that the root node's (A, B) window is equal to [-8, 36] by the time it reaches its fifth child F, who gets an Alpha-Beta window of [-8, 10]. After searching P, which has a value of -10, F gets an immediate cutoff and returns this value to its parent A, the uppermost chance node, which triggers another cutoff because -10 falls outside its lower bound of -8. The other children of F, as well as the sixth successor G, do not need to be searched, as we can prove that the Expectimax value of A must be less than -2 (it is in fact $-3\frac{1}{2}$, which we can read from Figure 3.2).

3.6 Star2

While Star1 results in an algorithm which returns the same result as Expectimax, and uses fewer node expansions to obtain the same result, its results are generally not very impressive. One reason is that Star1 is agnostic about its successors; it has no idea what kind of node (Min, Max or Chance) will follow it, but even if it did, it would not be able to take advantage of that knowledge. However, game domains are fairly regular; for example, in a standard Minimax tree, Min nodes and Max nodes are on levels that strictly alternate. Min always follows Max, and Max always follows Min. In games like backgammon, where each player rolls the dice, then moves, we end up with a tree like a Minimax tree, except we insert a chance node immediately after any non-terminal Min or Max node. In other words, we add a layer of chance nodes between each layer of nodes in a standard Minimax tree. Ballard refers to trees with this structure as *regular *-Minimax trees*, an example of which is shown in Figure 3.6, where +, - and * refer to Max, Min and Chance nodes, respectively.

Another drawback to Star1 is due to its pessimistic nature. We may potentially search nearly all the children of a chance node before a cutoff is obtained, because we assume that all unseen children have a worst-case evaluation. However, children of a successor of a chance node will tend to have values which are highly correlated. Instead of searching each child of a chance node fully and sequentially, and give a value of L to any children we haven't seen yet, we can get a more accurate picture just by searching a single successor of each child. This value we get for the child then becomes a bound on the true value for the child (a lower bound if the child is a Max node, and an upper bound if the child is a

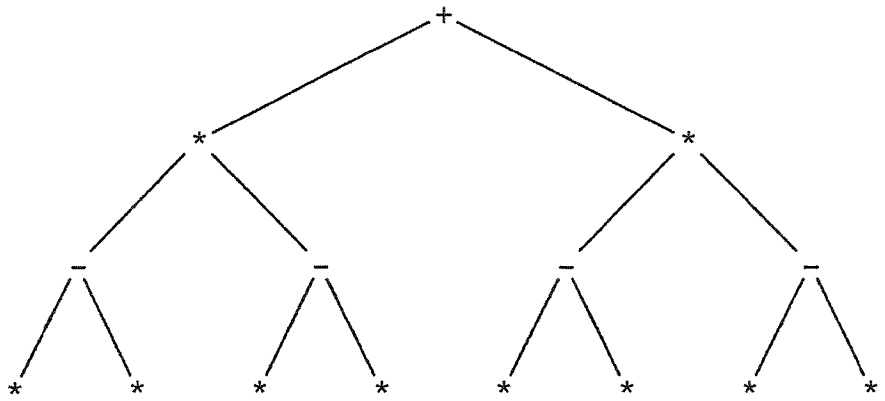

Figure 3.6: A regular *-Minimax tree

Min node). It is likely that the bound will be much better than L, especially if we chose the child well. We will therefore introduce this phase of speculative search (which we will call the *probing phase*) before sequentially searching each child, in order to obtain a quicker cutoff.

We need to modify the equations used to generate A and B in Star1 to reflect the new use of a probing phase in Star2. For Star2's probing phase, we derive the bounds for A and B just like we do in Star1's search phase, except we do not have alpha cutoffs at chance nodes followed by Min nodes (since we can only get an upper bound on those children), and we do not have beta cutoffs at chance nodes followed by Max nodes (since we can only get a lower bound on those children). Cutoffs generated at the probing phase are called *probe cutoffs*, and tell us how successful Star2 is at probing, or how good its *probing efficiency* is.

We obtain a cutoff in Star2's search phase if

$$\frac{(V_1 + \ldots + V_{i-1}) + V_i + (W_{i+1} + \ldots + W_N)}{N} \leq alpha \tag{3.5}$$

or

$$\frac{(V_1 + \ldots + V_{i-1}) + V_i + (W_{i+1} + \ldots + W_N)}{N} \geq beta \tag{3.6}$$

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

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

$$A_i = N \times alpha - (V_1 + \ldots + V_{i-1}) - (W_{i+1} + \ldots + W_N) \tag{3.7}$$

and the beta value for the ith successor, B_i with

$$B_i = N \times beta - (V_1 + \ldots + V_{i-1}) - (W_{i+1} + \ldots + W_N) \tag{3.8}$$

Like with Star1, these equations can be rewritten:

$$A_1 = N \times alpha - (W_2 + \ldots + W_N)$$

$$B_1 = N \times beta - (W_2 + \ldots + W_N)$$

and updated by

$$A_{i+1} = A_i + W_{i+1} - V_i$$

$$B_{i+1} = B_i + W_{i+1} - V_i$$

where $i = 2 \ldots N$.

Figure 3.7 shows the resulting Star2 algorithm for chance nodes followed by Min nodes (we will need a similar procedure for chance nodes followed by Max nodes). To get values for the probing phase, we need a procedure similar to Alpha-Beta since successors are Min or Max nodes. Figure 3.8 shows the Probe algorithm. Figure 3.9 shows the PickSuccessor algorithm used by `Probe`, which is explained in more detail below.

Consider the tree in Figure 3.10, to see Star2's strength. It is the same tree used in the previous example with Star1. For the probing phase, the alpha value changes just like with Star1, but the beta value does not. In this case, we only need to search five leaves: H, J, L, N and P, because by the time we reach child F, we give it a window of [-8, 10]. Since P has a value of -10, this causes a cutoff at F. It also causes a cutoff at A since F returns a value of -10, which is less than or equal to A. In this example our Probe function did a good job and we always chose the best child for probing (fortuitously), so we obtained a cutoff after searching about half the nodes Star1 searches.

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

In his paper, Ballard did not specify how `Probe` should choose a successor besides to say it could be done "at random or by appeal to a static evaluation function"[5]. Since the domain he used was limited to a depth=3 tree, all the probes done in his experiments were on leaf nodes. His domains also only had chance nodes at depth=1 (the nodes at depth=3 are technically chance nodes, but since they are leaves, they are just statically evaluated), so probing was always relatively inexpensive.

For Star2 to be successful, `Probe` must search a "good" child. We can abstract the selection process away from `Probe` and create another function, which we will call `PickSuccessor`.

```
float Star2_Min(Board board, float alpha, float beta, int depth) {
   if(terminal(board) || depth == 0) return (evaluate(board));
   N = numSuccessors(board);
   /* Range initialization */
   A = N*(alpha-U);
   B = N*(beta-L);
   BX = min(B, U);
   /* Probing phase */
   for(i = 1; i <= N; i++) {
      A += U;
      AX = max(A, L);
      w[i] = Probe_Min(successor(board,i), AX, BX, depth-1);
      if(w[i] <= A) return (alpha);
      A -= w[i];
   }
   /* Search phase */
   vsum = 0;
   for(i = 1; i <= N; i++) {
      A += w[i];
      B += L;
      AX = max(A, L);
      BX = min(B, U);
      v = search(successor(board,i), AX, BX, depth-1);
      if(v <= A) return (alpha);
      if(v >= B) return (beta);
      vsum += v;
      A -= v;
      B -= v;
   }
   return (vsum/N);
}
```

Figure 3.7: The Star2 algorithm, adapted from [5]

```
float Probe_Min(Board board, float alpha, float beta, int depth) {
   if(terminal(board) || depth == 0) return (evaluate(board));
   choice = PickSuccessor(board);
   return (Star2_Max(successor(board,choice), alpha, beta, depth-1));
}
```

Figure 3.8: The Probe algorithm

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

   return (choice);
}
```

Figure 3.9: The PickSuccessor algorithm, with quick two-quality check

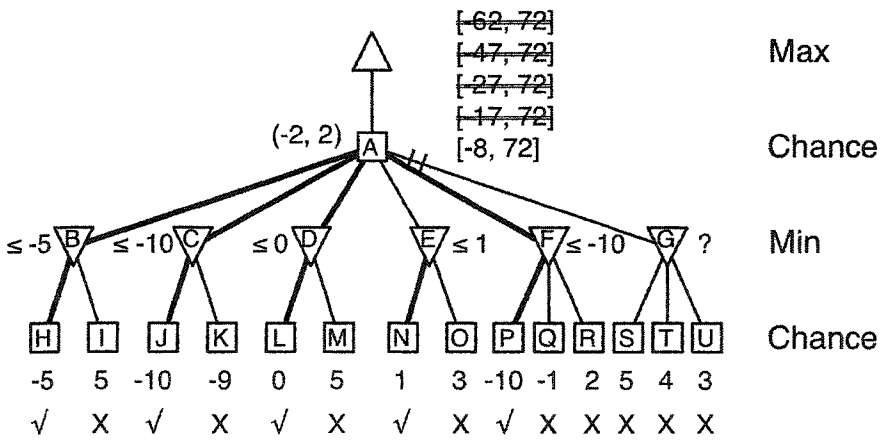

Figure 3.10: A Star2 tree, with good probing

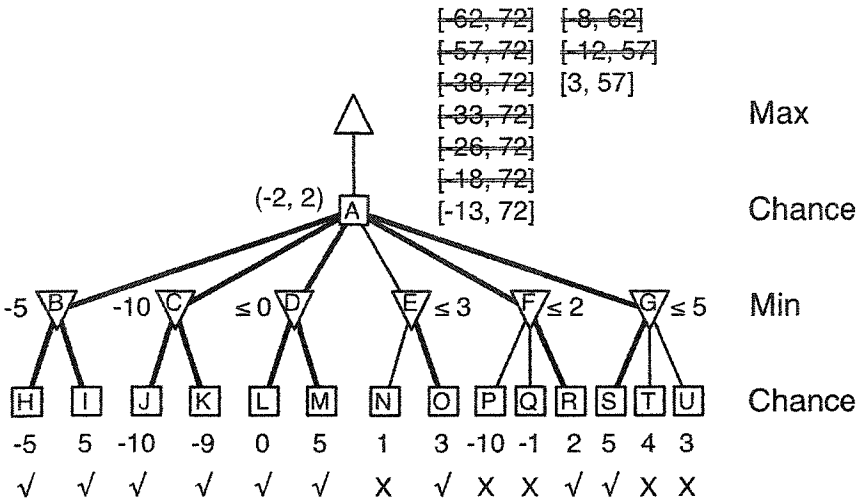

Figure 3.11: A Star2 tree, with bad probing

`PickSuccessor`, shown in Figure 3.9, will take a set of nodes and return the node it thinks is the "best". We want this selection process to be relatively fast and not use much overhead, so `PickSuccessor` may not want to use the evaluation function used for leaf evaluations, but instead use domain-specific knowledge to heuristically select a child. For example, in backgammon we may first select moves that result in hitting the opponent's blots, moves that form primes, or moves that form points. As soon as we see a successor that meets the best quality, we can simply exit with that successor as the choice. Failing that, we can keep track of a successor that has the next best quality. If no successors have either quality, then the first can just be chosen.

Even if we do not obtain a quick cutoff during the probing phase, we will have a tighter window for the search phase, which in itself will lead to quicker cutoffs, because we have better estimates of the values of the children. Reconsider once more the tree we have been using, but this time we will see what happens if Probe does a bad job. Figure 3.11 represents this situation. Now the probing phase will finish before we have obtained a cutoff, and so we will end up searching almost half of the leaves already. However, before the searching phase begins, notice that the window has been almost halved, because we have better upper bounds for the childrens' values; instead of starting with a window of [-62, 62] as Star1 would, we start searching sequentially with a window of [-8, 62]. Now, by the time we start to search the third successor D, we have passed it a window of $(3, 10)$. If we assume that the leaf node L is searched first, then we get a cutoff at D (because 0 is less than 3) as well as at A. We end up searching six leaves in the probing phase, and an additional three leaves in the search phase, for a total of nine leaves. In this particular situation, even the worst-case probing resulted in the same number of leaves expanded as Star1.

Chapter 4

Dice

This chapter will explore the game of Dice, a game invented to test the *-Minimax algorithms. Implementation issues of various search enhancements are discussed, and the performance of the algorithms in terms of node expansions and execution time is analysed with various branching factors and search depths. Tournaments of matches between programs with various depth settings are also run and evaluated.

4.1 The Game of Dice

Before testing the *-Minimax algorithms on backgammon, a smaller and simpler game domain was sought in order to obtain some preliminary results. Since many simple games that involve a small chance element don't involve strategy (such as snakes and ladders, or the card game War), a new game called Dice was developed. Dice is an N-in-a-row game played on a grid of squares (like tic-tac-toe, Connect-4 or Gomoku), except before players can move they must first roll an N-sided die. The value of the die will determine either which row (if X) or which column (if O) the player can move into. The first player to link N squares in a row wins the match.

4.2 Implementation Issues

The game definition itself is rather simple and thus mostly trivial to implement in code. An array of integers can represent the game state, pseudo random numbers can be easily generated to provide dice rolls, and moves are simply comprised of the row and column chosen by a player. Designing a Dice computer opponent is a little more tricky.

4.2.1 Evaluation Function

The first step is to design a decent evaluation function. In this case, 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. This is relatively fast and gives a decent strategic guess (since pairs of squares can quickly turn into triplets). If the game is more than 3-in-a-row, some tactical moves may be lost, since the evaluation function may favour obtaining a new pair instead of extending an existing pair to form a triplet. The evaluation function is also changed to reflect the player-to-move at that state, rather than scoring based on whether or not the player at the state is the same as the player at the root. The result is the score of the state, so that states that create new opportunities to join together squares, or block squares for the opponent, are favoured.

4.2.2 Transposition Table

A transposition table (TT) was used to speed up the search. The TT for Dice was a simple hash table of 128 MB (more or less space could be used, depending on the amount of main memory available). Each entry was 16 bytes large, containing the value for the stored state, a flag to indicate if the entry was in use, an indicator for the depth searched, two flags to determine what kind of value for the state is stored (a lower bound, upper bound, or an exact value), the best move chosen at that state, and the hash key for that state. A Zobrist hashing scheme[37] was used.

4.2.3 History Heuristic

In addition to the TT, another form of memory-assisted search was used: a *history heuristic* (HH). The HH is simply an array which keeps track of how often a move was selected as the "best" move at a node. This information can then be used during move sorting to favour moves chosen by previous searches, thereby likely improving cutoff performance.

4.2.4 Move Ordering and `Probe`

The move ordering and probe selection are almost nearly identical. As mentioned before near the end of Section 3.6, `Probe` didn't necessarily have to use the same evaluation function used for leaf nodes. In the case of Dice, the evaluation function was reasonably heavy, meaning it consumed most of the execution time during a search. The move ordering scheme worked as follows. Each move was applied to the game state and evaluated. If it was a winning position, then it was placed at the head of the moves list, and the function returned immediately. Otherwise, the number of pairs for each player were counted and the difference taken. Moves were then sorted based on this scoring scheme. `Probe` uses the same idea for choosing a successor.

4.3 Experimental Design

4.3.1 Hardware and Software

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

4.3.2 Environmental Conditions

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

4.4 Performance

Judging performance of a search algorithm usually means pitting it against other algorithms in the same class, and seeing how it compares in terms of total node expansions and execution time (counting all overhead). In order to obtain data, Dice positions were randomly generated and then searched by Expectimax, Star1 and Star2, each having all the same search enhancements. Each position was non-trivial: the player-to-move would have to have at least two moves to choose from, and none of the moves could lead to immediate wins. Results were obtained for search depths much greater than were possible for Ballard.

Boards of size 5×5, 7×7, 9×9, 11×11, 13×13, 15×15 and 17×17 were used for testing. 500 randomly generated positions were used for all board sizes. All boards were searched to at least depth=7 by Expectimax, Star1 and Star2.

There is a strong correlation between node expansions and time used in all experiments, mainly because most of the nodes in the tree will be leaves and they are all statically evaluated. In terms of CPU usage, the evaluation function for Dice consumed more than 90% of total cycles, with the rest of the time taken up by terminal position checks, TT calls and the search functions themselves. Move generation was nearly instantaneous because of the relative simplicity of the game.

When the branching factor is low, as is the case with 5×5 boards, there is only a reduction of about half in terms of average node expansions from Expectimax to Star2. Figure 4.1

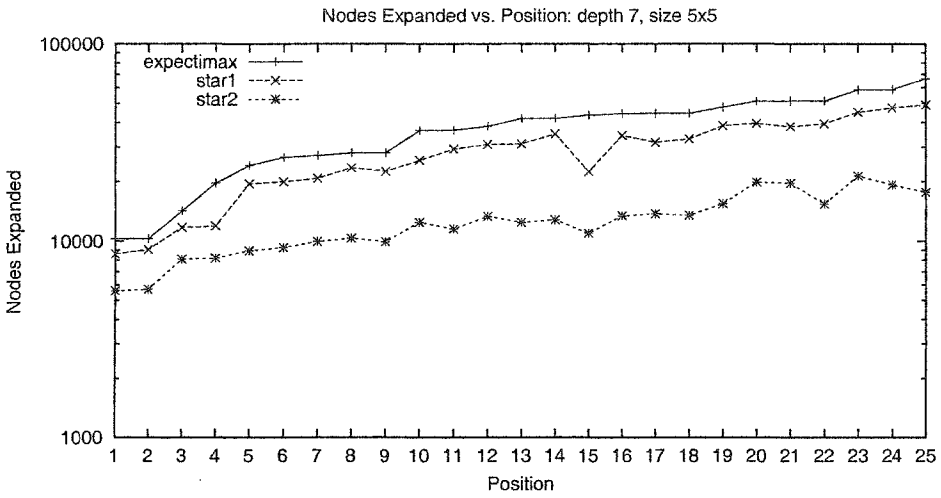

Figure 4.1: Node expansions at d=7 for board size of 5x5, for 25 positions

shows total node expansions at depth=7 for 25 positions, sorted by the Expectimax search tree size, on a logarithmic scale. Average node expansions and CPU time are shown in Figures 4.2 and 4.3, also on a logarithmic scale. One reason the reduction is so modest is because the branching factor is quite low. Not only is the board size relatively small, but as moves are made into empty squares, the branching factor gets closer and closer to 1. As the branching factor shrinks, so do the number of leaves in the tree, and therefore opportunities for chance node and move node cutoffs decrease. Since most of the nodes in the tree are leaves, and many of them are not pruned, the resulting reduction in node expansions is small. It is especially marginal for Star1, since most cutoffs in Star1 will occur at the last successors to a chance node. Star2 is not spectacular because with such a small branching factor, probing children becomes less effective; the Expectimax range of a chance node will not shrink fast enough for quick cutoffs to occur.

At a board size of 11×11, the difference in node expansions becomes significantly greater, as shown in Figure 4.4, where 25 different positions are looked at individually. Star2 is doing about 15% of the work of Expectimax at depth=7, compared to 50% before at depth=5. The difference in average time over all positions between the three algorithms for 11×11 is shown in Figure 4.6. The gap has certainly widened compared to 5×5 as the braching factor has increased, so probing becomes more fruitful.

4.4.1 Probe Efficiency

Ultimately, Star2's performance is directly related to probing success. Probing offers us a chance to get quicker cutoffs without needing to search all of the children of the successors to chance nodes. Getting a lower bound for the value of a successor by only looking at one of its children may be good enough to produce a cutoff at the chance node. The better the

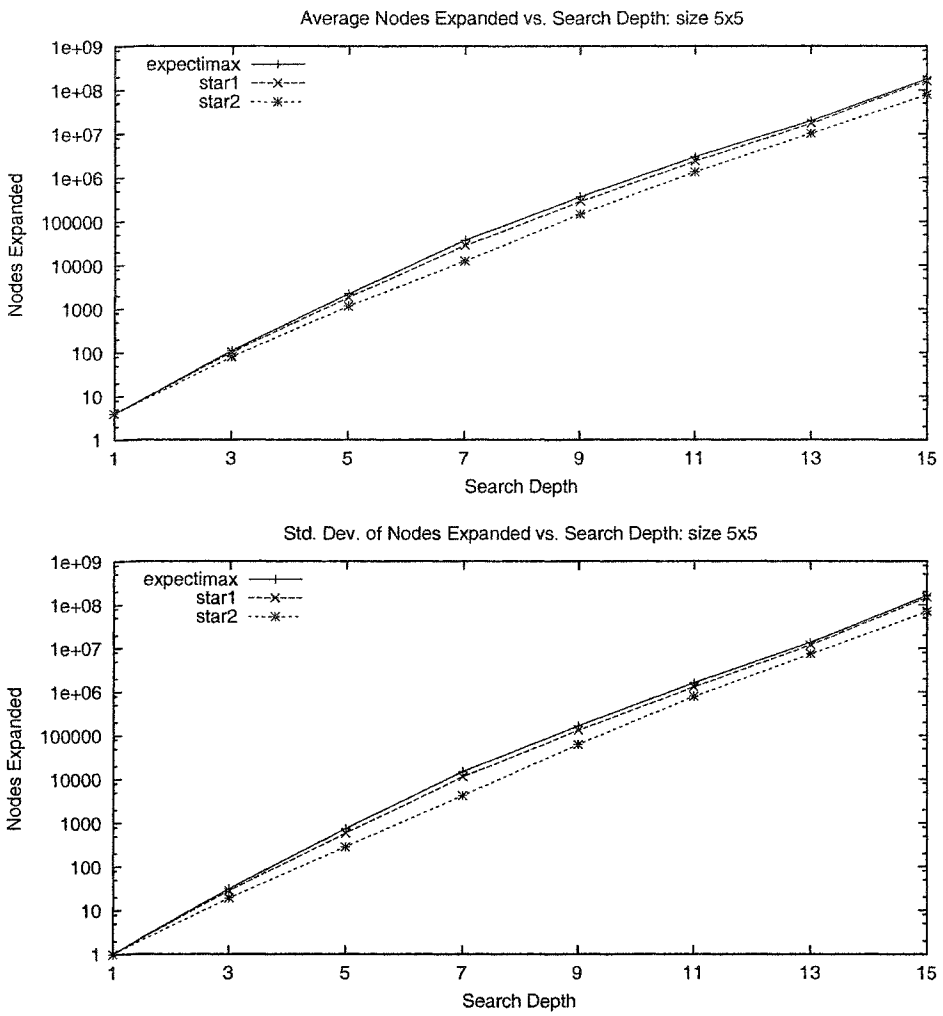

Figure 4.2: Average and standard deviation of node expansions for board size of 5x5, over 500 positions

child we select for each successor, the higher the lower bound we get for the successor, and the more likely we are to produce a quick cutoff.

Figure 4.7 and Table B.2 show the average and standard deviation of probe efficiency in Dice at various depths. In Dice, as the depth of the search increases, so does the effectiveness of probing. There is an odd dip in the average graph at depth=5, followed by a jump back up at the next depth, and then a plateau. At the same time, the standard deviation looks the opposite. For depth=3 trees, there is only one layer of chance nodes (the leaves at the bottom are chance nodes, but they just get statically evaluated). Star2's probing does not have a chance to recursively call itself, so its probing should be relatively good, since `Probe` will be returning an accurate value for children. At depth=5, probing will recursively call itself right before the horizon. At that point, probing will fail often if the Alpha-Beta window is not narrow enough. As long as the TT caches these leaf node evaluations, there

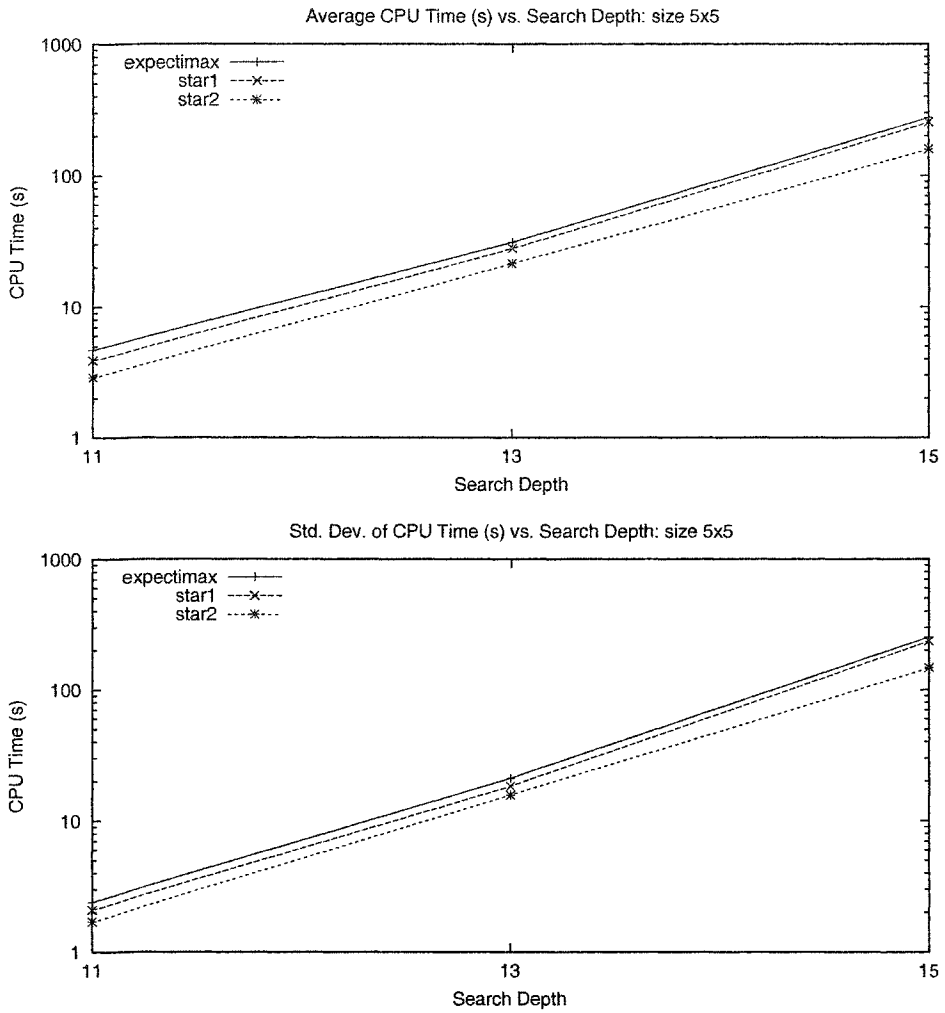

Figure 4.3: Average and standard deviation of time (s) for board size of 5x5, over 500 positions

won't be an impact of overall performance as Star2 switches from the probe phase to the search phase, but it will impact the probe performance. As the depth increases to 7, probe efficiency goes back up a bit, since there are two probes now that happen before the horizon. However, since most `Probe` calls happen just before the horizon, overall performance is not as good as at depth=3. The performance then mostly levels off but continues to decrease as relatively more and more probing calls happen just before the horizon.

Boards with higher branching factors result in better probing performance, as cutoffs at chance nodes become more likely, because there is sufficient chance to narrow the Expectimax range during the probing phase.

In Ballard's original paper, his Star2 probing effectiveness was somewhere between 33% at a branching factor of $N = 4$ and 45% at a branching factor of $N = 40$, compared to a range of 66% at $N = 5$ to 96% at $N = 17$ in Dice. This difference is mainly from picking

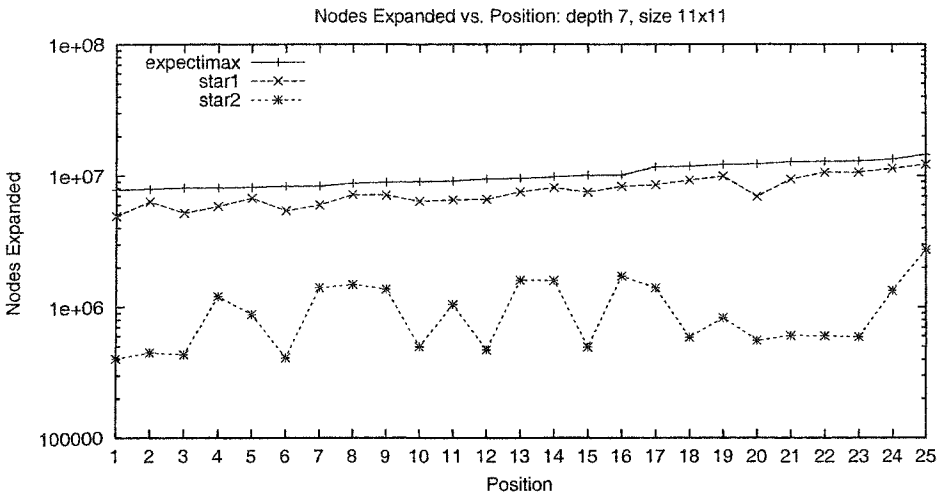

Figure 4.4: Node expansions at d=7 for board size of 11x11, for 25 positions

the right successor to probe.

4.4.2 Move Ordering

Ballard used random Move ordering in his original experiments. As he put it, "...all $N!$ permutations of successor arcs were assumed to be equally likely."[5]. One of his reasons for doing so was to give a conservative picture of what to expect in practice[5]. However, this assumption is not necessarily true. Move ordering is an important part of any search algorithm, and as such, even the simplest of routines can perform better than random in practice.

Ballard randomized successor orders for both chance nodes and Min or Max nodes. The implementation of Dice only included move ordering for Min and Max nodes. Chance successors (the die rolls) were pre-ordered in code to go from smallest to largest.

For ordering move nodes, four different schemes were implemented: no move ordering; random move ordering; static evaluation function move ordering; and "quick" move ordering.

Figure 4.8 shows the average node expansions for searches performed with various move ordering schemes at varying search depths on a 11 × 11 board using Star2. 500 positions in total were used for each scheme. Data is presented in Table B.17. No move ordering is usually beaten by the other schemes, although the differences are fractional. Random move ordering is sometimes better than no move ordering, and sometimes worse. These results would seem to indicate that random move ordering may not, in fact, be a conservative estimate of performance, since it requires some overhead, and may be worse than no move ordering at all. As such, the random scheme should probably not be used at all. Move ordering using the static evaluation function or the "quick" heuristic is quite good, but not overall stellar compared to no move ordering.

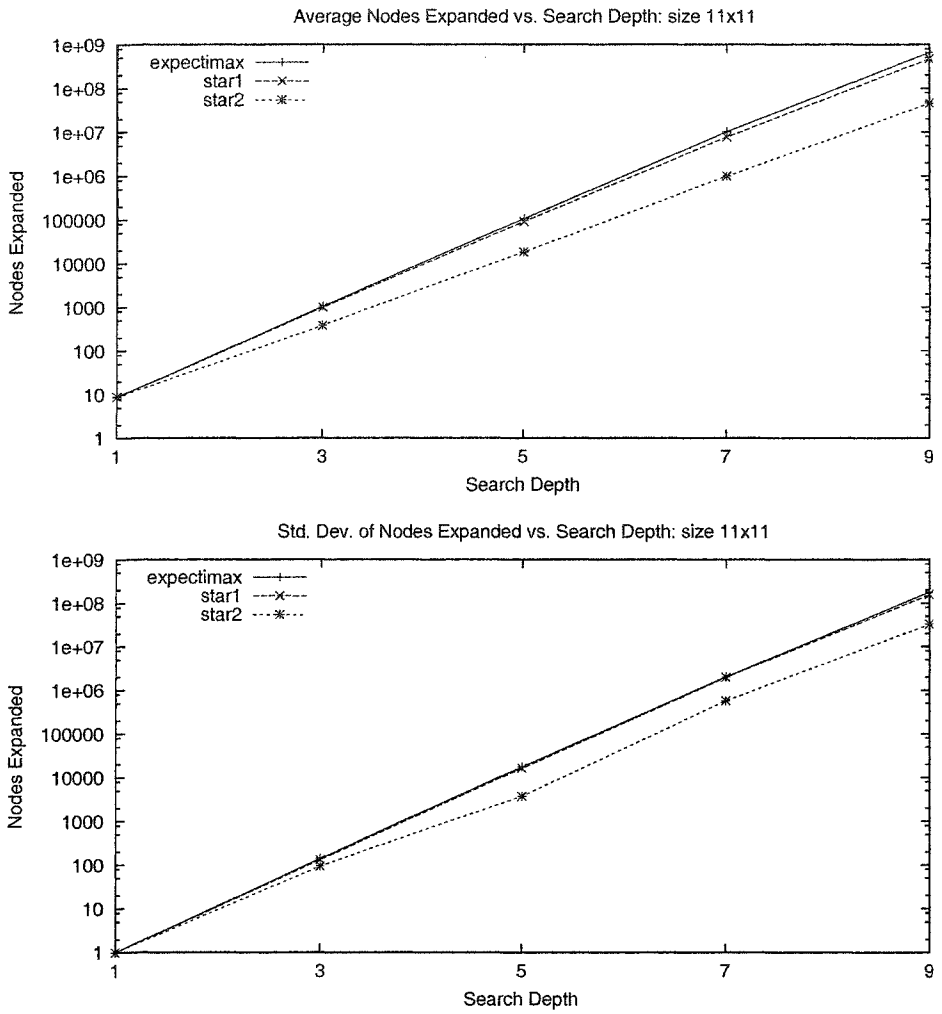

Figure 4.5: Average and standard deviation of node expansions at d=7 for board size of 11x11, over 500 positions

While a reduction in node expansions usually is directly correlated with a reduction in execution time, Figure 4.9 and Table B.18 show that this is not always the case. Three of the four schemes take about the same amount of execution time – only quick ordering is better! The reason, of course, goes directly back to why we do not want to use a heavy static evaluation function for probe successor selection. With static move ordering, all the time we save in reduced node expansions is eaten up by the fact that we are still applying the evaluation function to every node we generate. Since most nodes will be leaves, this means we really don't end up with many savings at all.

Finally, Figure 4.10 and Table B.19 show the scheme's effect on probe efficiency. The most informed scheme, using the static evaluation function, is best, but the quick scheme is not far behind. Obviously, our evaluation function does a good job at pre-ordering successors for the "quick" successor selection scheme, where the first move that meets the "best" quality

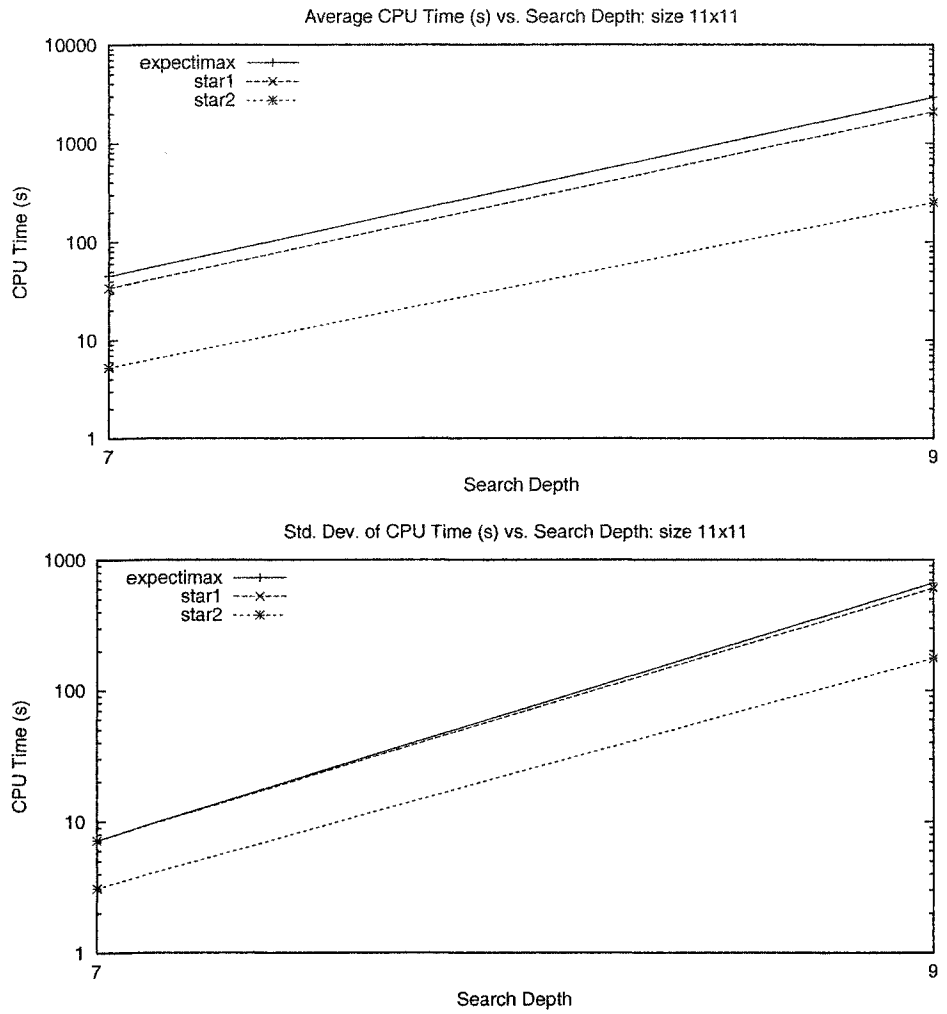

Figure 4.6: Average and standard deviation of time (s) for board size of 11x11, over 500 positions

will be taken. Failing that, the first move that meets the "good" quality is taken. Both random and no move ordering still maintain a probe efficiency greater than 50%, because the "quick" successor selection scheme works fairly well without any help.

4.5 Tournaments

Another way to measure an algorithm's performance is to pit it against itself in a tournament, where each player is searching to a different depth. The question to answer, then, is if deeper search increases real performance in the game. Tournaments were therefore run between combinations of players searching to depths of 1, 3, 5, 7 and 9. Each tournament used a file containing a sequence of seed values, such that they all would then have the same sequence of dice rolls across each tournament. The starting roll for each game was pre-set by a testing script, and went through the values from $1..N$ sequentially, in order to reduce

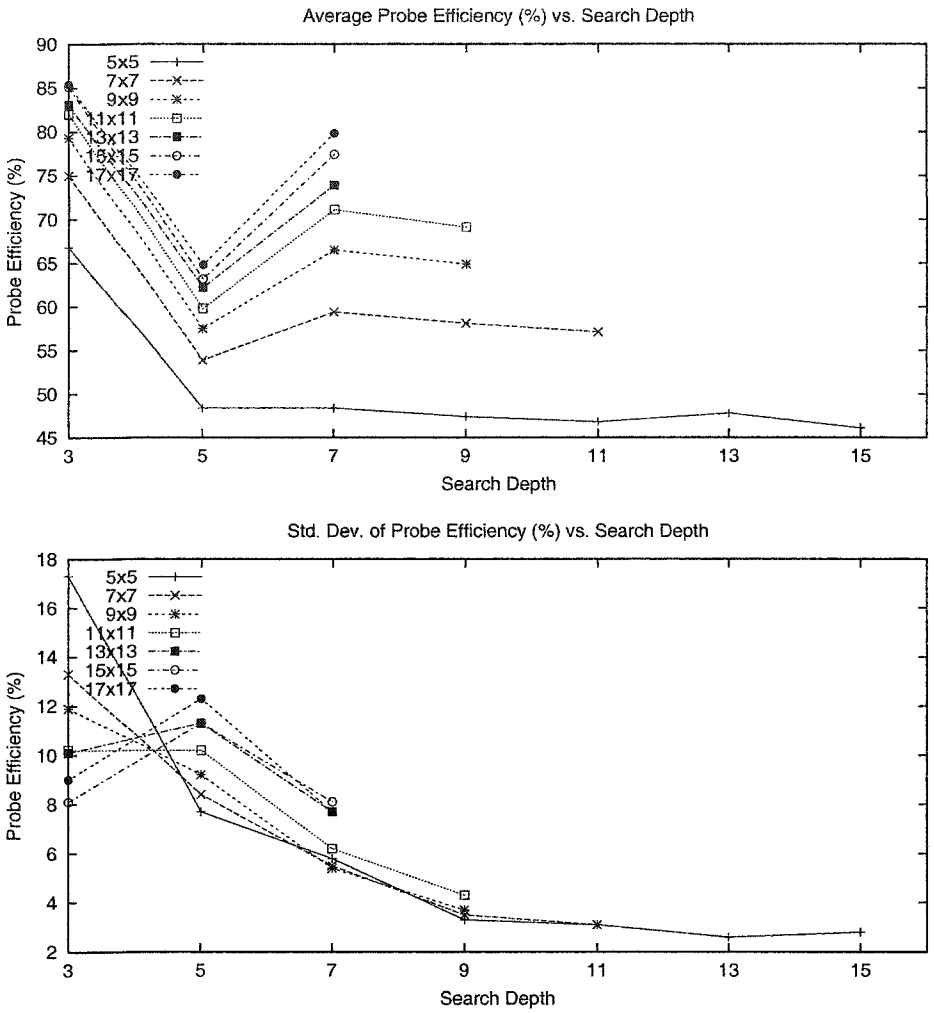

Figure 4.7: Average and standard deviation of probe efficiency for Dice

variance. Furthermore, for each opening roll and sequence of dice rolls, both players were given an opportunity to be the starting player. A file containing 9000 seeds was used, and therefore a total of 18,000 games per tournament were played.

The results of the tournament are shown in Figure 4.11 and Table B.1. On each graph, lines represent different players searching from depth=1 to depth=9. The line is then compared against an opponent on the x-axis, and the winning percentage for the player represented by that line is shown on the y-axis. A player searching to the same depth as its opponent has the same performance and so crosses the 50% winning percentage line where the two players' settings are identical.

There are some important points to make from the figure. The player at depth=1 never fares better than 50%. Deeper search results in a greater winning percentage. What is interesting is how the performance of the depth=3 player closely follows the depth=1 player, followed by a jump of around 10% for the next three players. While searching to at

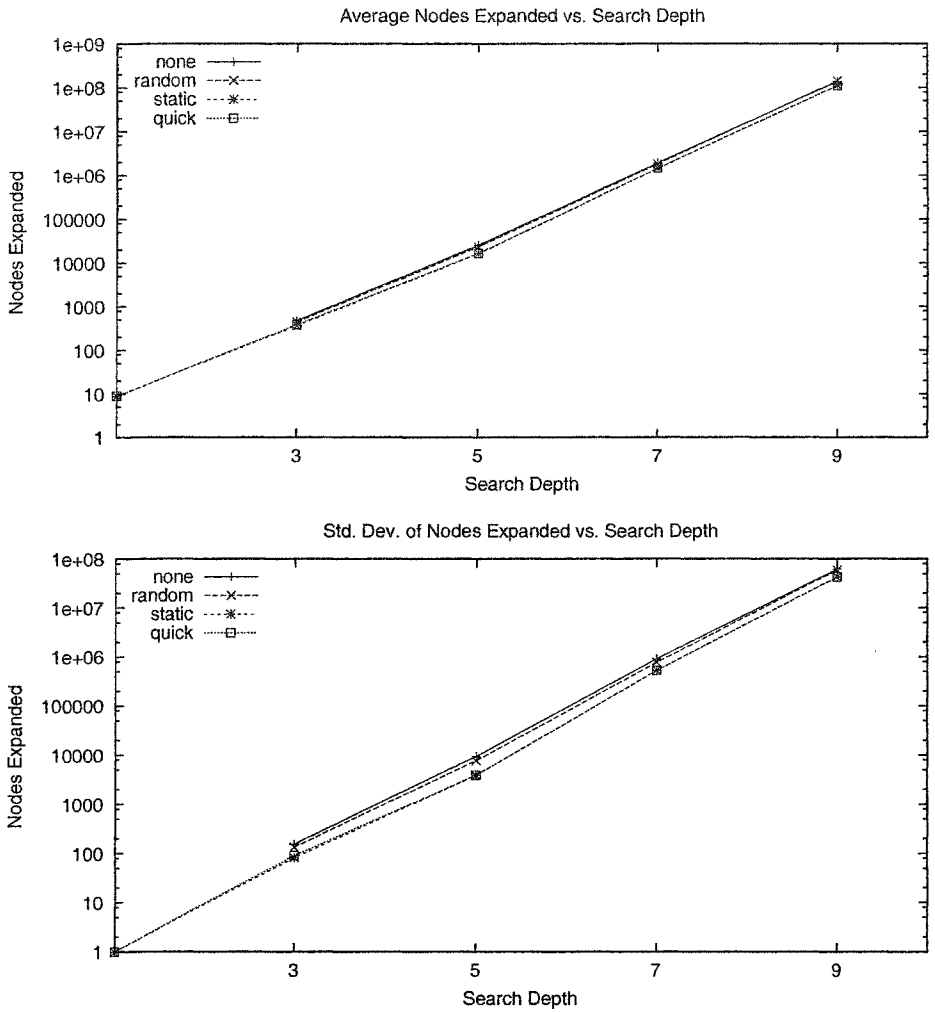

Figure 4.8: Average and standard deviation of node expansions for different move orderings for board size of 11x11, over 500 positions

least depth=5 results in decent tournament performance compared to shallower searches, the benefits level off very quickly. This jump probably comes from the nature of using a 4-in-a-row win. The die rolls inject enough randomness into the game to weaken the benefit of deep search. For example, being able to set up a future move by seeing 3-ply deep will be just as good as seeing 5 plies into the future, because the die roll will wash away tactical plays. The game also favours offensive play, because setting up groups of squares is better than trying to surround the opponent; if a player is nearing a win, then deep search beyond a certain level doesn't help. Consider a situation where the player to move has a 50% chance of winning on the next roll, and if they don't win, their opponent has a 50% chance of winning on their roll. The odds that the game will end in the next two moves is relatively high, and so in this case, deep search doesn't really help, because the stochastic nature of the game eliminates deep planning. On the other hand, seeing a single ply ahead,

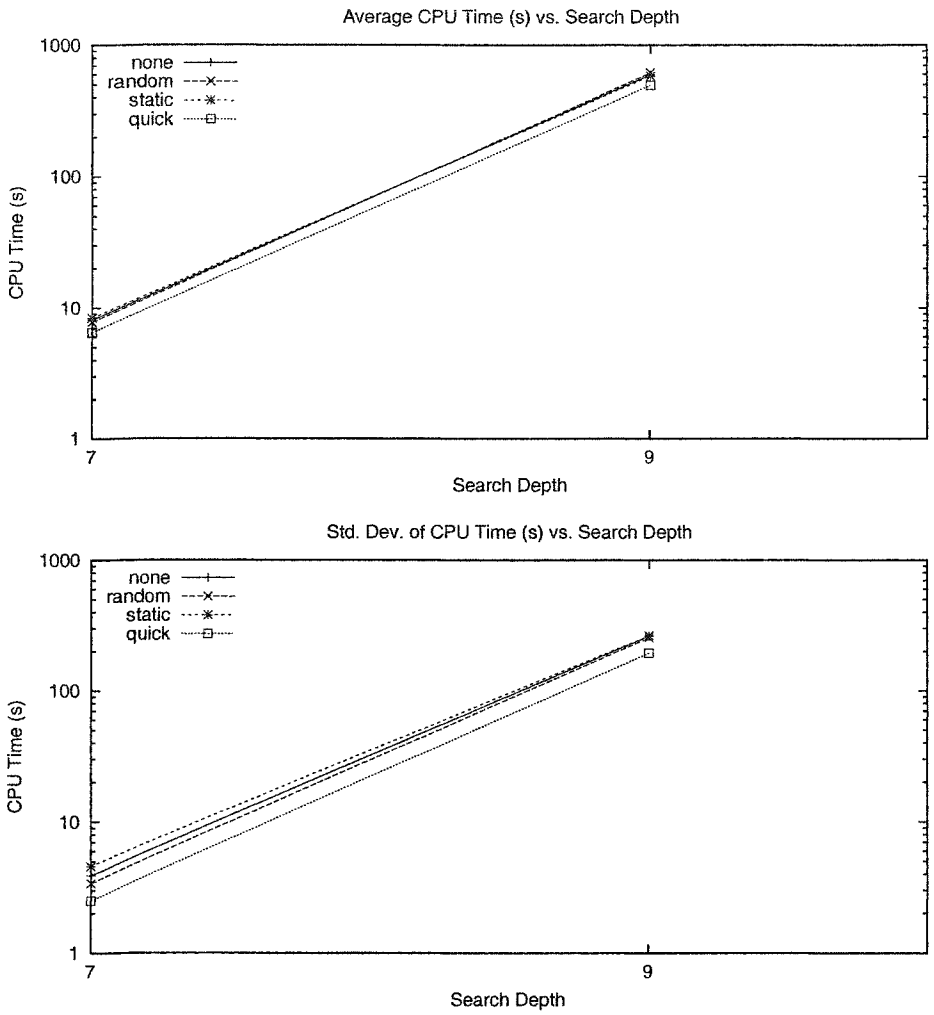

Figure 4.9: Average and standard deviation of time (s) for different move orderings for board size of 11x11, over 500 positions

compared to three or more moves ahead, makes a significant impact. Again, the nature of this game suggests that a player should endeavour to be aggressive and set up as many winning positions as possible. Seeing at least two of "our" moves ahead means the player is able to set up more winning positions. Defensive play doesn't seem to matter very much; you can block an opponent in one spot, but it is often impossible to block them everywhere. Furthermore, if they get a good roll on the next move, the game could be over.

Star2 allows for faster search in tournaments, which means players can search deeper in a given amount of time, which means players will have better overall performance.

4.6 Conclusions

For single searches on Dice positions, Star2 outperforms Expectimax in all cases. As the branching factor for the game increases, the performance gap between the two widens. For

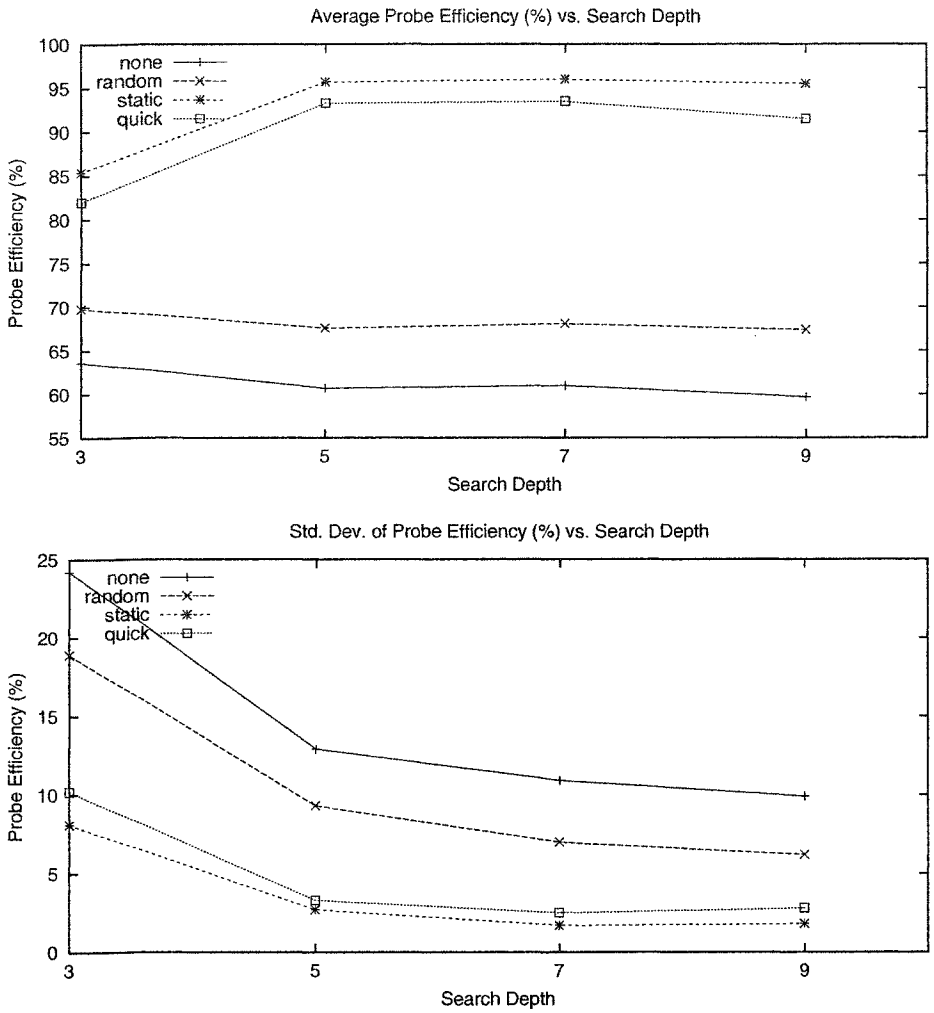

Figure 4.10: Average and standard deviation of probe efficiency for different move orderings for board size of 11x11, over 500 positions

games with large branching factors, doing exploratory probes results in greatly reduced node expansions and CPU time.

Good probing efficiency is critical to Star2's performance. Trees with small branching factors do not allow many probing cutoffs because there may not be sufficient opportunity to narrow the Expectimax window before all successors are probed. Trees with large branching factors allow for more cutoffs because there is usually time to narrow the window before all successors are searched.

Move ordering schemes can have a small but important impact on performance. Randomly ordering successors may not be the best baseline policy to use; sometimes no ordering does better. Move ordering using the static evaluation function results in reduced node expansions, due to a more informed ordering, but nearly the same time, because all nodes that are generated get evaluated. A move ordering scheme based on heuristic, domain-specific

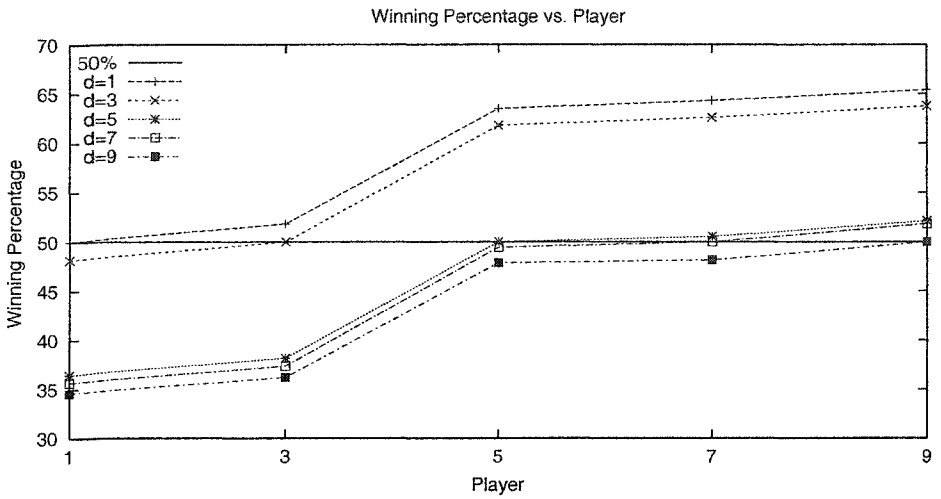

Figure 4.11: Tournament results for 4-in-a-row on 7x7 board, 18000 games per matchup

knowledge can reduce both node expansions and time, and need not be complicated to implement.

Good tournament performance in 7×7, 4-in-a-row Dice is dependent on seeing at least three plies deep, when using an evaluation function that just counts pairs of tokens. Deeper search results in only incremental improvement due to the stochastic nature of the game offsetting deep tactical play with quick, unavoidable wins.

4.7 Future Work

The Dice game can and should be extended to a bigger board, with a longer winning combination length. The evaluation function could also be improved a great deal instead of just counting pairs. A version of Gomoku or Connect-4 could be developed with dice rolls to determine legal moves. The advantage to using those games is that there are known good strategies for the deterministic versions of both games, meaning a powerful evaluation function should be easier to build. Another change would be to how chance events affect the game; instead of having a die roll specify a row or column, perhaps the player could choose a square from a random selection of empty squares. This change would increase the strategic aspect of the game.

Chapter 5

Backgammon

This chapter will introduce the game of backgammon, describe some of the history of computer backgammon programs, and describe one program (GNU Backgammon) in detail. Implementation issues of *-Minimax with the game of backgammon are discussed, including search enhancements. Experimental results are also presented.

5.1 The Game of Backgammon

Backgammon is an ancient game, considered to be maybe the oldest game still being played today. There are dozens of variants played in countries around the world, but the name backgammon is reserved for the most common type. Backgammon is a race game played on a board with 24 columns or *points* on which *checkers* (also called pieces, stones, or blots) are placed (see Figure 5.1). The goal of the game is to be the first to remove all your checkers from the board. It is normally played by two players who each have fifteen checkers. A point with no checkers on it is called *empty*. A single checker on a point is usually called a *blot*. Two or more checkers on a column are also referred to as a *point*. Each player moves their pieces in a direction opposite to the other on alternating turns. A pair of dice is thrown at the beginning of the turn to determine legal moves, and checkers can be moved anywhere except where the opponent has claimed a point. A roll of 3-1 means that a player can move one checker 3 points and another checker 1 point, or a single checker 4 points (as long as the opponent hasn't blocked the intermediate columns). When a checker is moved to a column containing a single opponent blot, this is called a *hit*. The opponent's single checker is then moved to the *bar* and the player's checker takes its place. Whenever a player has a checker on the bar at the beginning of their turn, they are required to move the checker off the bar back into the opponent's *home board* (the opponent's last 6 points) before they can move any other checker (the bar can be considered to be the 25th point on the board). Points formed in the opponent's home board are called *anchors*, because they lock in a column that can be used for safely moving off the bar. All the standard rules for moving checkers

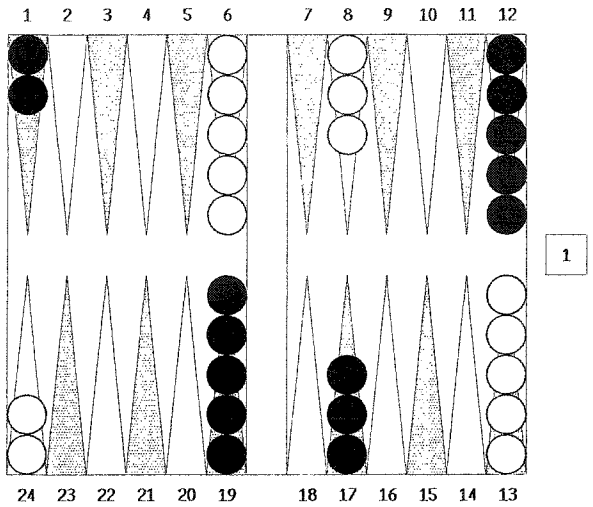

Figure 5.1: The initial starting position for backgammon. White moves counterclockwise toward 1, while black moves clockwise toward 24.

on the board apply to moving checkers off the bar.

If a player has at least one checker on the bar on their turn, but neither of the dice can be used to move the checker back on the board (because the opponent has *blocked* those columns with points), this is called *dancing* on the bar. Figure 5.2 shows an example, where white cannot move back onto the board from the bar because black has blocked both the 24 and 20 points. If the opponent is lucky enough to have all six inner points blocked (also called a *prime*), then the player cannot possibly move on his or her turn.

A player rolling a *double*, when both dice have the same value, are allowed *four* moves equal to that value on their turn. Since double rolls only account for $\frac{1}{6}$ of all possible rolls, they are relatively infrequent, but they often cause significant swings in the outlook of a game.

If ever possible, a player *must* use both of the dice rolls available to them, or in the case of doubles, all four moves. For example, Figure 5.3 shows white being forced to move off the bar onto the 22 point, and then needing to move another checker with the 4 roll.

If all moves cannot be used (perhaps due to an opponent blocking movement), then as many as possible must be used. In the case that a player can use one die roll or the other die roll, but not both die rolls, then the die roll with the greater value must be used, such as in the position in Figure 5.4, where white's final checker will be forced to move to the 21 point.

As stated before, the goal of backgammon is to be the first to remove or *bear off* all of

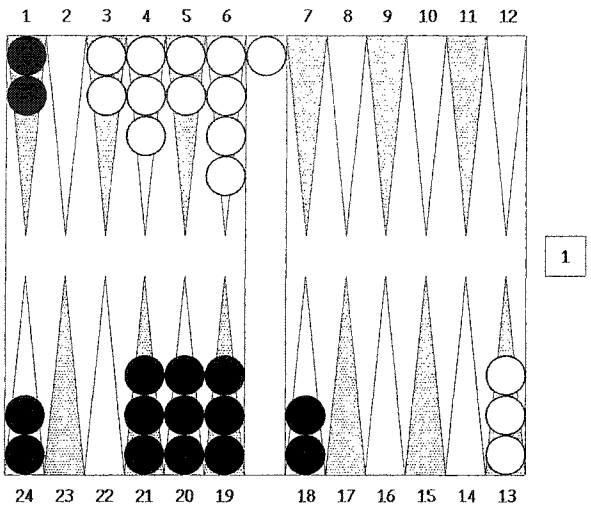

Figure 5.2: White to play 5-1: dancing on the bar

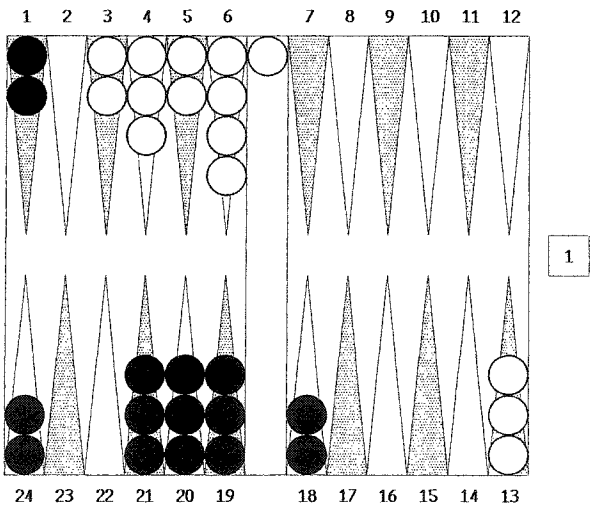

Figure 5.3: White to play 4-3: a forced move off the bar

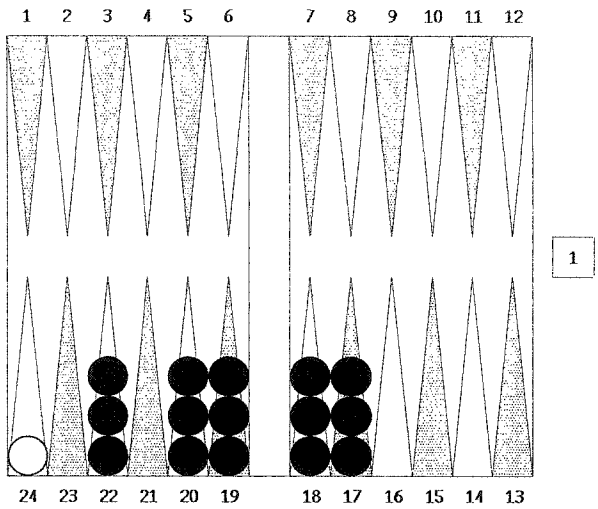

Figure 5.4: White to play 3-1: only one roll can be used

one's pieces from the board. This is accomplished by first moving all of one's checkers into one's home board, and then removing checkers one by one based on the dice rolls. A checker can be removed if it is on a point equal to one of the dice rolls. However, if there are no checkers on a point matching a dice roll, but there are also no checkers behind that point, then the roll can be used to move off the checker farthest behind. In Figure 5.5 white rolls 5-4, and is able to remove the checker on the 5 point; since there are no checkers on or after the 4 point, white can use the second half of the roll to bear off the blot on the 3 point. Without this rule, endgames would be rather tedious, and very heavily based on luck.

Usually people play matches consisting of multiple games, where each game is worth a single point. In this case, a *doubling cube* is also used to speed up the game, as well as add an element of gambling to backgammon. The doubling cube (sometimes also just referred to as *the cube*) was introduced in the 1940s. Initially, the cube is placed in the middle of the table (it is not *owned* by anyone) and its value is considered to be equal to 1. During the game, if a player particularly likes his chances of winning, they may *offer a double* to their opponent, which is an offer to double the stakes of the game. If the opponent *declines* the offer, this is a resignation of the game for a single point. If the opponent *accepts* the double, the cube is *turned* to show twice its current score, and the opponent now *owns* the cube, and the game is played at twice the stakes. From that point on, the opponent will have the sole option of being able to offer the next double. If accepted, the cube would be turned again, and ownership passed to the other player. Backgammon players often talk about the

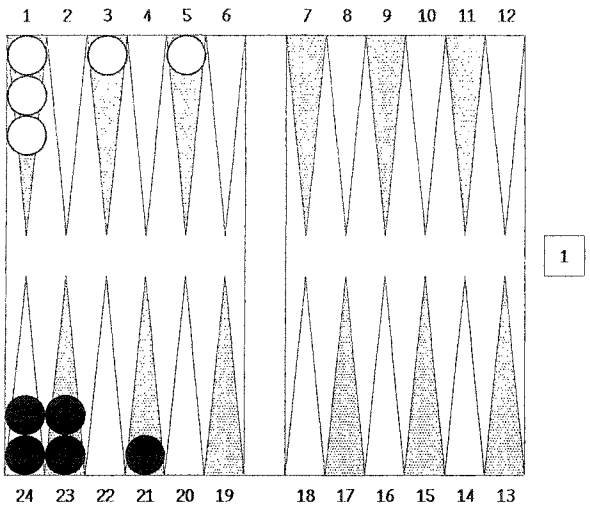

Figure 5.5: White to play 5-4: bearing off

complexity of *cube handling* in match play: weighing the odds one can win at higher stakes versus the increased price of failure. Even at the world championship level, matches are routinely won and lost by poor doubling decisions. Even so, the doubling cube remains the most popular addition to the game, and is now considered standard in a backgammon set.

There are two special types of wins that result in more points for the winner: the *gammon* and the *backgammon.* A gammon win happens when the opponent has not removed any of their checkers off the board, and makes the game worth double (further multiplying the cube value). A backgammon win happens when the conditions for a gammon win are met, plus the opponent has at least one checker on the bar or in the player's home board. Backgammons make games worth three times the amount of the cube, but are generally very rare, since most players can at least escape their opponent's home before the game ends.

Besides typical "match" play, there is also another type of play in backgammon called a *money game.* All the rules are the same, except for the addition of the *Jacoby rule* which states that gammons and backgammons do not count unless the cube has been turned at least once. The purpose behind this rule is to speed up the game, so people are less likely to play for gammons with a centred cube.

5.2 Backgammon Programs: Past and Present

With such a gigantic state space (estimated to be bigger than 10^{20} states[32]) and such an imposing branching factor (there are 21 unique dice rolls, and about 20 moves per roll, on average), it's not surprising that most of the early computer backgammon programs were *knowledge-based.* Knowledge-based systems do not rely much on search (or at all), but rather attempt to choose moves based on *expert knowledge* about the domain, usually programmed into the system by a human expert. These are usually called *ad hoc* methods.

The first real success in computer backgammon was BKG, developed by Hans Berliner. In 1979, BKG played the world champion at the time, Luigi Villa, and managed to defeat the human 7-1 in a five point match.[6] While many people were shocked, even Berliner himself would concede weeks after the match that BKG had been lucky with rolls and made several technical blunders. However, Villa had not been able to capitalize on those mistakes – such is the life with dice.

The second milestone in computer backgammon was Neurogammon[31], the work of IBM researcher Gerald Tesauro. Neurogammon used a *neural network* (a computer model loosely based on a biological brain) for evaluating backgammon positions. Neurogammon was trained with *supervised learning*; it was fed examples labeled by a human expert, and told what the answer should be. The program quickly became the best in computer backgammon, but still only played at the level of a strong human amateur player.

Tesauro went back to the drawing board, with a desire to improve his creation even more. One of the first things he changed was the data the program was training on. Instead of using hand-labeled positions, he decided he would rely solely on *self-play* to generate training data – the program would simply play against itself. This has advantages over the previous method since a human expert may label positions incorrectly, or tire quickly (Neurogammon only used selected positions from about 3000 games[31] to train checker play, culled from games where Tesauro had played both sides), but self-play also may lead a program into a *local* area of play. For example, a program can learn how to play well against itself, but not against another opponent. This local minima problem in backgammon is partially overcome due to the fact that the environment is stochastic – dice insert a certain level of randomness – so a program is forced to explore different areas of the state space.

The other thing Tesauro changed was the training method itself. Instead of using a supervised learning approach that adjusted the network after each move (which he could do before because each training example was labeled), Tesauro decided on adapting *temporal-difference learning* for use with his neural network[30][32]. TD learning is based on the idea that an evaluation for a state should depend on the state that follows it. In a game sense, the computer keeps track of each position from start to finish, and then works backward. It trains itself on the last position, with the target score being the outcome of the game.

Then it trains itself on the second last position, trying to more accurately predict the score it got for the last position (*not* the final score). The last position is the only position which is given a *reward signal*, or absolute value; all other positions are only trained to better predict the position that followed it. In games, the reward signal is related to the outcome of the game. If the program had lost, the reward signal would be low (to act sort of like a punishment). If the program won, the reward signal would be high. Since backgammon cannot end in a draw, the reward signal could never be zero.

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

TD-Gammon's use of temporal difference learning and a neural network evaluation function has lead to several copy-cat ventures, including the commercial programs Jellyfish[3] and Snowie[4], as well as the open-source GNU Backgammon[2] (also known as Gnubg). Several versions of GNU Backgammon have sprung up on the Internet, and it has quickly become one of the most popular codebases for developers.

5.3 Some Failings of Backgammon Programs

While so much time and effort has been put into creating backgammon programs with increasingly stronger evaluation functions, almost no thought has been put into improving the search used in the programs. Both TD-Gammon and Gnubg use a *forward-pruning* based approach to search, where some possible moves are eliminated before they are searched in order to reduce the branching factor of the game. Depending on the approach, using forward pruning can be a bit of a gamble, since the program is risking never seeing a good line of play, and therefore never having the chance to take it.

There are two important reasons why improvements in search have not been developed in backgammon. The first is that the current crop of neural network-based evaluation functions are pretty accurate, but take far too long in processing terms. For example, a complete 3-ply search of an arbitrary position in backgammon can take several minutes to complete. This

is clearly undesirable from a performance perspective. The second reason has to do with the game itself. Since there are 21 distinct rolls in backgammon (with varying probability), and often up to 20 moves per roll, the effective branching factor becomes so large that, especially for a slow heuristic, searching anything deeper than a ply or two becomes impractical. It is pretty clear due to these reasons why efforts have concentrated on developing an evaluation function that is as perfect as possible (as close to an *oracle* as one can get), instead of trying to grapple with the explosive branching factor inherent in the game.

But search *is* still important. Deeper search allows for the inaccuracies of a heuristic to be reduced, and as mentioned before, the deeper a program can search, the better that program can play. Backgammon is no exception, even with a trained neural network acting as a near-oracle. Still, it is interesting to note that improving search in backgammon programs has not been a priority, to the point where some of the GNU backgammon team are unfamiliar with the concept of Alpha-Beta search. Tesauro thinks that improvements in search will come as a result of faster processors and Moore's Law[33], and has not yet considered using a new algorithm.

Backgammon can be considered the Drosophila of perfect-information chance games. It has been explored heavily in the past few decades, but nearly all the research has centred on producing good evaluation functions for estimating the utility of a state (or board position). It will be the primary test domain used in this thesis.

5.4 Overview of GNU Backgammon

GNU Backgammon is an open-source backgammon program developed through the GNU Project. Development began in 1997 by Gary Wong, and has continued up to this time with contributions from dozens of people. The other five primary members today are Joseph Heled, Øystein Johansen, David Montgomery, Jim Segrave and Jørn Thyssen. The current version of Gnubg, 0.14, boasts an impressive list of features, including TD-trained neural network evaluation functions, detailed analysis of matches (including rollouts), a tutor mode, bearoff (endgame) databases, variable computer skill levels and a graphical user interface. Gnubg is also free, and since its exposure to the backgammon community was heightened, it is one of the most popular and strongest backgammon programs available.

5.4.1 The Evaluation Function

Gnubg has three different neural networks it uses for evaluating a backgammon position, depending on the classification of that position: either contact (at least one checker of a player is behind a checker of the other player), crashed (same as contact but with the added restriction that the player has 6 or less checkers left on the board, not including any checkers on the opponent's 1 or 2 points) or race (the opposite of a contact position). Since each of

the three types of positions are quite different from the others, using three different neural networks improves the quality of the evaluation.

Each neural network is first trained using temporal difference learning, using self-play, similar to TD-Gammon. The input and output representations of the neural networks are also similar to TD-Gammon. The input neurons are comprised of both a raw board representation (with 4 neurons per point per player) as well as several hand-crafted features, such as the position of back anchors, mobility, as well as probabilities for hitting blots.

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

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

A new network is trained against this database so its depth=1 evaluations more closely

resemble a depth=5 search, and after the new network is fully trained, it then provides new entries for each position in the database. Gnubg was able to obtain a rating of about 1930 at a depth=1 setting on the First Internet Backgammon Server (FIBS), which put it roughly at an expert level on the server.

5.4.2 The Search Algorithm

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

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

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

For depth=1 searches, Gnubg simply performs a static evaluation of all root move candidates (a candidate being a move that has not been pruned by the move filter), and the move with the highest score is chosen. At chance nodes in the search tree, all rolls in the roll set (the set is usually all 21 rolls but it can be reduced for speed) are investigated, and the best move for each roll (chosen by simple static evaluation) is applied and expanded, until the depth cutoff is reached. As we saw before with Expectimax, the size of the tree is $O(B^{\frac{D}{2}}N^{\frac{D}{2}})$, where B is the branching factor at move nodes, N is the branching factor at chance nodes, and D is the search depth. By only doing a static evaluation of children at move nodes and then choosing only one for further expansion, the size of a Gnubg search tree is $O(P^{\frac{D}{2}}N^{\frac{D}{2}} + B^{\frac{D}{2}} \times B)$ (where P is the pruned branching factor at move nodes), and in the best case is generally asymptotically similar to $B^{\frac{D}{2}}$, since the variable branching factor at move nodes is usually about the same as the fixed branching factor (21) at chance nodes.

In other words, this pruning technique allows the search tree to be exponentially smaller than the full tree in depth (with savings about $O((\frac{P}{B})^D)$, but error is also introduced.

5.4.3 Best in the World?

On September 5th, 2003, Michael Howard posted the results of a duel between Gnubg and Jellyfish to the `rec.games.backgammon` group on UseNet.[14] Howard had the two programs play 5,000 money games, each using their "optimal" settings. Gnubg came out the winner by an average of 0.12 points per game. One of the Gnubg developers, Jørn Thyssen, commented that the results were within his estimated 95% confidence interval of +/- 0.1 points per game to show that Gnubg was clearly a stronger program.[36] While not completely shocking, it is a strong statement on the strength of the Gnubg program to be able to outmatch an expensive, "professional" backgammon program like Jellyfish.

5.5 Implementation Issues

Unlike Dice, backgammon is not a trivial game to implement. While the board itself can be fairly easily represented by a two-dimensional array of integers, generating moves is rather complicated to not only do correctly, but also efficiently.

5.5.1 Move Generation

There are three different stages of the game that change what moves are legal: (1) when a player is on the bar, (2) when the player has no checkers on the bar but has checkers outside their home board, and (3) when a player has no checkers on the bar and all checkers inside their home board. The third stage also has different rules about when checkers can be borne off, if the player doesn't have a checker on a point equal to one of the rolls. There is also the situation where a player can use one but not both of their rolls, in which case the higher roll can be used. This can be handled by always making the largest roll the first one examined, and then slicing off part of the array of moves after all moves are generated. Avoiding duplicating moves is also an important consideration because of the explosive branching factor for some situations (like a doubles roll for a player with checkers on several different points). Move generation is done recursively, where a single "partial move" is done at a time (moving a single blot), and continuing until all such partial moves are completed, and then the move is stored.

While checking for duplicate moves after they are generated is rather inefficient, a simpler way to avoid most duplicate rolls is to just limit the next recursive call to applying partial moves on or after the point from which the last partial move was made. While some double roll moves may still be duplicated, the transposition table can take care of the majority of those.

5.5.2 Evaluation Function

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

While the Dice code used an evaluation function that just returned a difference of pairs (in other words, an integer), the Gnubg evaluation function returns a floating point number (the value representing the equity of the player who just moved). Whenever search programs use floating point numbers, there is always the risk of floating point operations having rounding errors; even comparing two (seemingly) identical values may not result in the expected truth value.

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

Using floating point numbers instead of integers also meant a performance hit, because floating point operations can be much more costly than integer operations.

5.5.3 Transposition Table

The same code used for the Dice TT was used with backgammon, with one minor changes to the TT entries: they stored double-precision floating point numbers instead of integers for values.

5.5.4 History Heuristic

Since the HH is usually represented by an array of moves (each entry representing the number of times a move was chosen as best), in backgammon this representation is near-impossible – a rough bound on the number of different moves is $(25 \times 6)^4$. The number of possible moves in backgammon is so large than even if the HH could fit into memory (by encoding one partial move in a byte, and using a four byte primitive), it surely would not fit into cache. Since the HH is supposed to be a small portion of memory to modestly help with search, using a half a gigabyte of memory is not reasonable, and we'd end up playing havoc with the CPU's cache. For this reason, the HH was not implemented for backgammon.

5.5.5 Move Ordering and Probe

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

5.5.6 Non-uniform Chance Event Probabilities

While only a single die was used in Dice and so each chance event had a uniform probability of occurrence ($\frac{1}{6}$), two dice are used in backgammon, and all combinations do not have the same likelihood: the 1-1, 2-2, 3-3, 4-4, 5-5 and 6-6 double rolls all have the same probability ($\frac{1}{36}$), but all other combinations have a probability of $\frac{2}{36}$. For this reason, the formulas used to derive the equations for A and B need to be modified. Ballard talks about the modification process in [5] but does not go into much detail. Note that this process doesn't affect Expectimax, just Star1 and Star2.

Recall the inequality for obtaining A_i:

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

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

$$(P_1 \times V_1 + \ldots + P_{i-1} \times V_{i-1}) + P_i \times V_i + U \times (1 - P_1 - \ldots - P_i) \leq alpha \tag{5.1}$$

or

$$A_i = \frac{alpha - U \times (1 - P_1 - \ldots - P_i) - (P_1 \times V_1 + \ldots + P_{i-1} \times V_{i-1})}{P_i} \tag{5.2}$$

where P_i is the probability that the ith chance occurs, for A. B can be found similarly with

$$B_i = \frac{beta - L \times (1 - P_1 - \ldots - P_i) - (P_1 \times V_1 + \ldots + P_{i-1} \times V_{i-1})}{P_i} \tag{5.3}$$

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

$$A_i = \frac{(alpha - U \times Y_i - X_i)}{P_i} \tag{5.4}$$

and

$$B_i = \frac{(beta - L \times Y_i - X_i)}{P_i} \tag{5.5}$$

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

We can use these equations to determine A and B for Star1 as well as Star2's probing phase. When calculating A and B values in Star2's search phase, we can still use Equation 5.4 to get A, but for B we will need to modify Equation 3.8, and get

$$B_i = \frac{(beta - W_i - X_i)}{P_i} \tag{5.6}$$

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

5.6 Experimental Design

The same laboratory conditions used for Dice were used for backgammon experiments. The only change is the usage of the Gnubg codebase for the evaluation function. Only the needed object files were included, and they all were compiled with `-O3` optimization.

5.7 Performance

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

500 randomly selected contact positions were used for testing. Each was searched to depths of 1, 3 and 5 by Expectimax, Star1 and Star2.

Just like with Dice, there is a direct relation between time and node expansions, as the Gnubg evaluation function is very heavy in terms of CPU usage (over 90%, much like Dice).

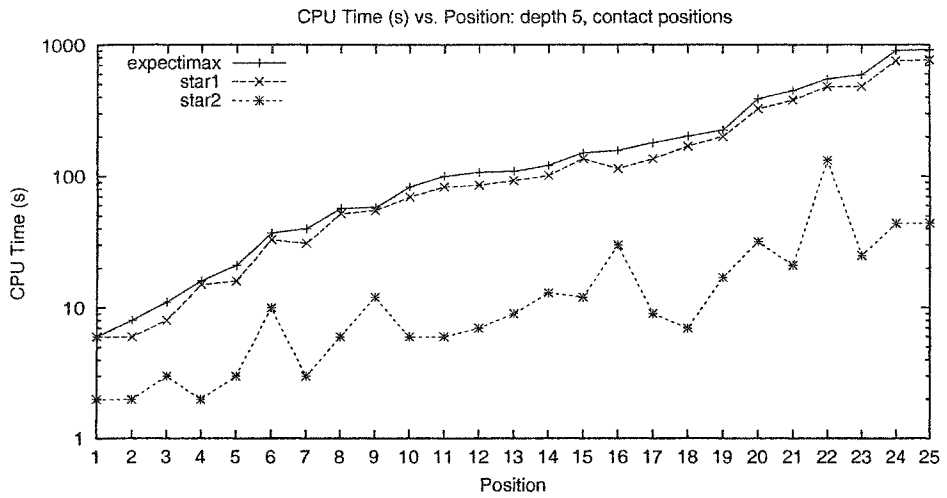

Figure 5.6: Time used (s) at d=5 for 25 contact positions

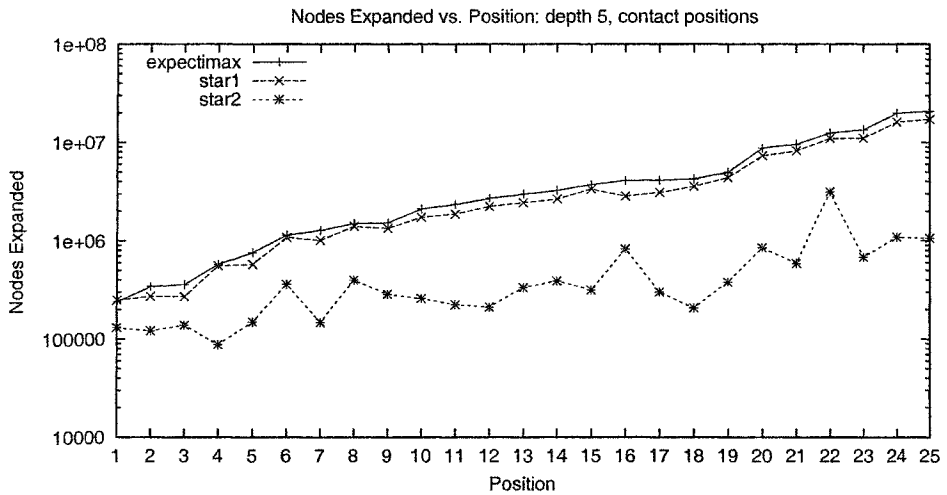

Figure 5.7: Node expansions at d=5 for 25 contact positions

In Figures 5.6 (CPU time) and 5.7 (node expansions) graphed on a logarithmic scale, we can see some variation in the amount of effort Star2 requires to complete a search at depth=5, which reflects the variety of backgammon positions during a match. Each of the 25 positions shown were selected at random from the Gnubg contact position database, searched by all three algorithms, and then sorted in order of Expectimax time. The variation in savings for Star2 for the 25 positions goes from about 75% to about 95%. Expectimax and Star1 closely follow each other, where Star1 has only a slight decrease in overall costs.

Table 5.1 summarizes the time usage over 500 positions. Star2 is clearly the most efficient of the algorithms by over a factor of 10, but even at 21 seconds per search, this would probably still be too slow for tournament play. Figure 5.8 shows average and standard deviation of node expansions over 500 positions, graphed on a logarithmic scale.

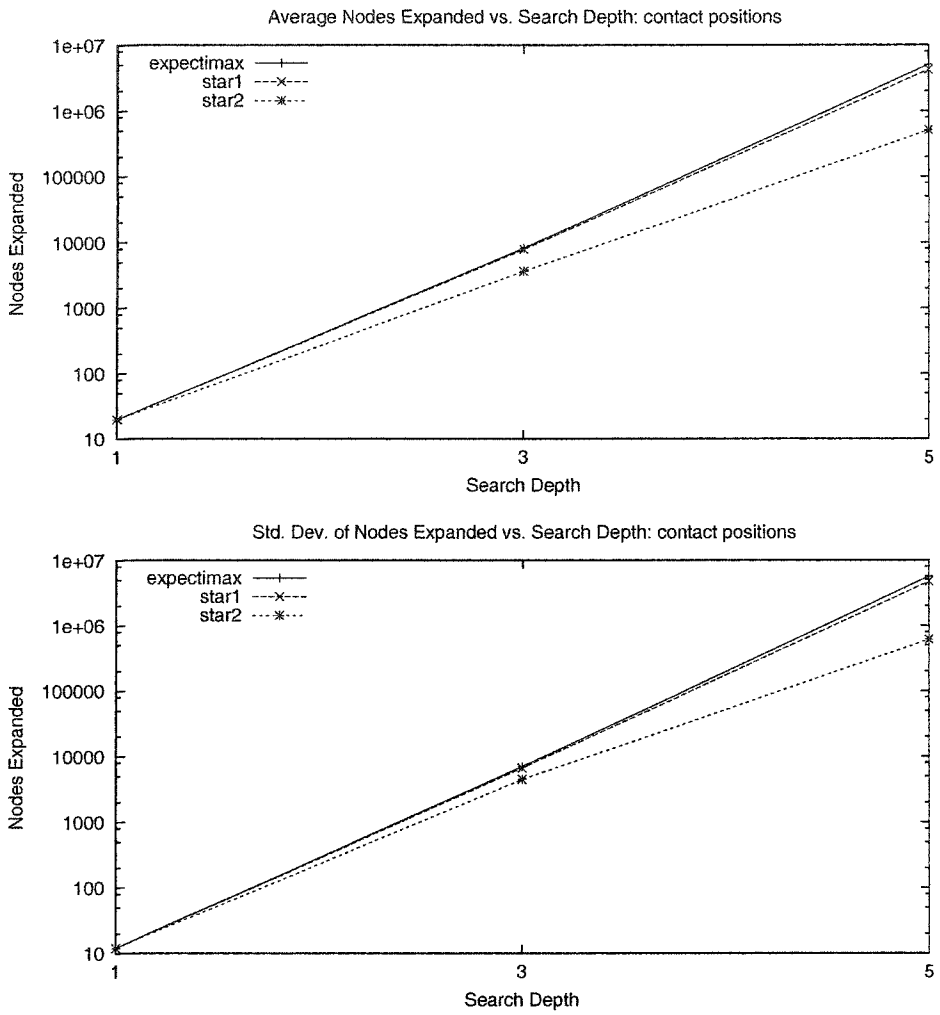

Figure 5.8: Average and standard deviation of node expansions over 500 contact positions

5.7.1 Probe Efficiency

Table 5.2 shows the resulting probe efficiency for using Star2 in backgammon. The results are modest compared to some of the values seen for Dice, but backgammon has a much larger branching factor, so deep searches are not possible. The "quick" successor selection scheme for backgammon is also relatively weaker than Dice's, because backgammon is a much more complicated game. Not only it is harder to quickly find good qualities, but it's also harder to define good qualities for backgammon positions. Still, these results are better than Ballard's, whose probing was never successful more than about 45% of the time. The improvement here is probably due to better move ordering.

Table 5.1: Average time (s) over 500 contact positions

	Expectimax		Star1			Star2		
	μ	σ	μ	σ	%	μ	σ	%
d=3	1.1	0.7	1.1	0.6	100	1.0	0.1	91
d=5	315.0	566.8	258.6	472.6	82	21.0	36.9	7

Table 5.2: Probe efficiency for Backgammon

	d=3		d=5	
	μ	σ	μ	σ
contact	68.9%	29.5%	64.2%	22.6%

5.7.2 Odd-Even Effect

Many 2-player game-playing computer programs suffer from what is called the *odd-even* effect, where alternating levels of move nodes will give scores that are either optimistic (for odd-ply searches) or pessimistic (for even plies). For example, a depth=1 search in any game will tend to be optimistic, since we are only investigating the moves currently available to us. The odd-even effect comes from the way in which an evaluation function is created, which generally tries to score the position for the player-to-move.

Tables 5.3 and 5.4 show the results of two different trials of 3200 backgammon positions. The positions were generated as a continuous sequence of cubeless money games, with the computer playing for both sides. This generated a decent set of "real-world" moves for backgammon. The table shows the average difference, absolute average difference, and absolute standard deviation in the root node value when comparing searches of the same positions to different depths.

Table 5.3: Root value differences, over 3200 moves (A)

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

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

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

These results show a tangible odd-even effect with the Gnubg evaluation function. Even if searches to different depths produce different values for the root, the move chosen at

Table 5.4: Root value differences, over 3200 moves (B)

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

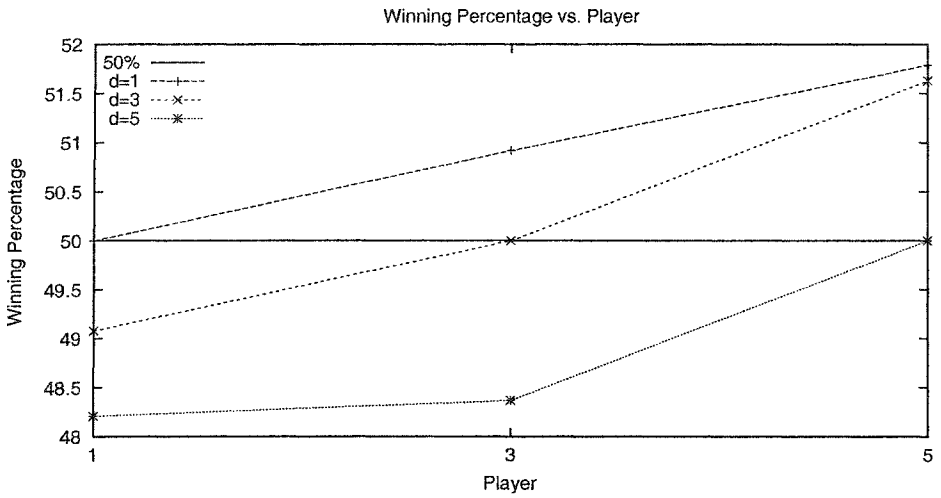

Figure 5.9: Tournament results for Gnubg with no noise versus Gnubg with no noise, 18000 games per matchup

the root usually is the same across searches of different depths. This means the evaluation function itself is very consistent between depths. These results also show that a depth=1 search value is a reasonable predictor for a depth=5 search value for the same position.

5.8 Tournaments

Tournaments were set up just like with Dice to investigate how deep search impacts game performance. Tournaments were set up between the Gnubg search function and itself, and Star2 against itself. Since Gnubg also has a facility for adding deterministic noise to an evaluation, different noise settings were also investigated.

While we saw that deep search was beneficial for tournament performance in Dice, this was not evident in backgammon. Figure 5.9 shows the results of Gnubg playing against itself at different depth settings. These graphs are the same type from the previous chapter. On each graph, lines represent depth settings from depth=1 to depth=5. The line is then compared against an opponent on the x-axis, and the winning percentage for the player represented by that line is shown on the y-axis. A player searching to the same depth as its opponent has the same performance and so crosses the 50% winning percentage line where the two players' settings are identical.

We can see from the graph that a depth=5 barely shows any significant improvement over shallower searches. In fact, the three depth settings are nearly identical. This suggests

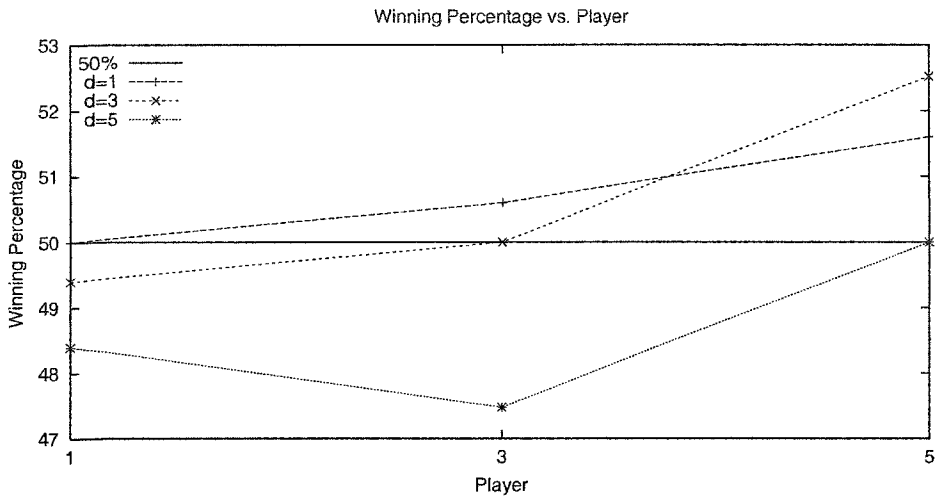

Figure 5.10: Tournament results for Star2 with no noise versus Star2 with no noise, 4000 games per matchup

that deeper searches are only finding better moves a small fraction of time, which suggests that the three searches are choosing the same move just about every time. That means the Gnubg evaluation function must be extremely consistent between depth levels.

Figure 5.10 shows Star2 performance when playing against itself, in the same manner as Figure 5.9. Deep search is still pretty much irrelevant using the Gnubg evaluation function as-is.

Since the evaluation function is so consistent, results were also desired for a less consistent setting. Instead of developing a new evaluation function, noise can just be added to the evaluation function. Gnubg has a built-in noise generator already, which can add either deterministic or non-deterministic noise to each evaluation. Since it is highly desirable that the evaluation for a state be always deterministic, especially when transpositions are possible, another tournament using deterministic noise was added. Only a modest amount of noise was added, consistent with an "Intermediate" level of play for Gnubg. Figure 5.11 shows tournament results in an identical manner to the previous two graphs. Now, deeper search is paying off to a significant degree; a depth=1 search now loses to a depth=5 search 65% of the time. Depth=1 fares slightly better against depth=3 at about 42% winning percentage. Depth=5 wins slightly less than 55% of the time against depth=3, but it is still a tangible amount.

Deep search helps to mitigate bad evaluation functions by adding more foresight to the move decision process. Adding deterministic noise to the Gnubg evaluation function shows that deep search becomes important again in backgammon.

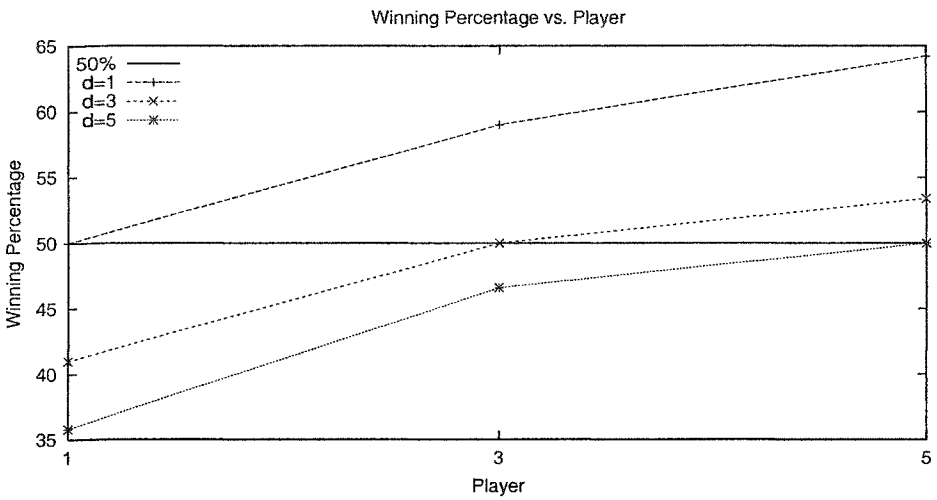

Figure 5.11: Tournament results for Star2 with n=0.0300 versus Star2 with n=0.0300, 1000 games per matchup

5.9 Conclusions

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

Unlike Dice, strong cubeless money game tournament performance is not linked with deep search. The Gnubg evaluation function is sufficiently well-trained and consistent that searches to increasing depths almost always choose the same move at the root. When the searches do not agree on the best move it is usually because they are searching a tactical position. But even the occurrence of tactical positions is relatively infrequent, and the benefits of deep search in these situations is usually washed away by the randomness of the dice rolls.

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

5.10 Future Work

With an excellent evaluation function such as Gnubg's set of neural networks, checker play is virtually perfect, even with shallow search. However, since backgammon matches are

generally played with a doubling cube, and cube decisions are usually the most important part of the game, this work should be extended to cubeful games and cube decisions. Being able to see even a couple of ply deeper in these situations can make or break a player's chances of winning.

New evaluation functions should also be considered. The Gnubg evaluation function is much heavier (about 21,000 evals per second on an Athlon 1800) relative to evaluation functions for many other games. Faster methods may prove useful combined with deep search, like, for example, using a GLEM-based evaluation approach[10]. If the amount of evaluations per second could be increased into the range of a million, full 3-ply (depth=7) searches should be possible.

Since Star2 is so reliant on successful probing, a more powerful `Probe` function would also increase performance. Right now `PickSuccessor` uses some ad-hoc rules about good backgammon play for quickly choosing a child, but there are perhaps better techniques for making this decision.

Chapter 6

Conclusions

This thesis involved the application of the *-Minimax algorithm, which allows for backward pruning in game trees with chance nodes, to two games: Dice and backgammon. Both Star1 and Star2 were applied versus Expectimax, the baseline algorithm for game trees with chance nodes. Single searches of positions as well as tournaments were run, in order to investigate the effectiveness of the algorithms in reducing search costs in either domain, as well as to see if deep search makes a positive impact on performance.

6.1 Dice

Overall, we saw that Star1 does outperform Expectimax, but generally by not more than a modest linear factor. Star2 outperforms both Expectimax and Star1, usually by a large factor. In Dice, we saw that a small branching factor resulted in relatively weak performance from Star2, but as the branching factor increased, so did the effectiveness of Star2, from a 50% reduction on a 5×5 board up to more than 90% reduction in costs on a 17×17 board. We also saw a strong correlation between node expansions and time used, stemming mainly from the fact that when an evaluation function is sufficiently heavy, nearly all CPU time will be spent evaluating nodes. Tournament performance in 7×7, 4-in-a-row Dice was strongly reliant on seeing at least 3 plies deep (depth=5), but deeper search did not really matter. We saw that Star2 is reliant on proper probe selection to ensure high efficency, and that move ordering is an important consideration for reducing node expansions and time, and raising probe efficiency.

6.2 Backgammon

In backgammon, we used a strong, open-source codebase in GNU Backgammon to develop a backgammon program capable of playing cubeless money games. We saw how even at a small depth, the large branching factor of backgammon created gigantic search trees for Expectimax. We also saw that Star2 reduced the search costs by about 90% at depth=5, which

resulted in significant savings. Curiously, we also saw that, using GNU Backgammon's evaluation functions as-is, there was no great improvement in tournament performance when doing a depth=1 search versus a depth=5 search, whether using Star2 or GNU Backgammon's own forward pruning-based search routine. However, with the introduction of even a small amount of deterministic noise into the evaluation, deep search had a positive impact on performance once again.

6.3 Future Work

Extending the Dice game to different board sizes and winning combination lengths should be investigated, as well as creating a better evaluation function. A game based on Gomoku or Connect-4 would provide an interesting counterpoint to those two games, which are already solved.

Checker play in backgammon seems relatively trivial, since heavy analysis of even the most tactical positions often becomes meaningless thanks to lucky dice rolls for either side. Still, cubeful play remains an important area to be investigated, because of the great impact good cube handling has on overall performance in match play. Combined with appropriate use of forward-pruning in backgammon, *-Minimax may also provide some incremental improvements in play from deeper search.

There are other perfect-information stochastic games which could benefit greatly from the use of *-Minimax search. One excellent domain would be the German tile-laying game Carcassonne. While there is only one real computer version of the game, produced by KOCH Media (`http://www.carcassonne-online.de/`), newer programs could definitely provide some competition and renewed interest. One good reason why *-Minimax may work well in this domain is because of the endgame play in Carcassonne. Often games are very close leading up to the final few tiles to be played, and games often slow down significantly when players do not see immediate "good" moves to play. In this case, being able to see a line of play from even five or six tiles out could result in expert play. Because computers can also keep track of which tiles have been played better than humans, a computer player could also avoid many of the pitfalls which plague humans. However, since the branching factor at chance nodes after the root starts at 40 (when using the most common expansion tileset, Inns & Cathedrals), some form of statistical sampling may be required to jumpstart the computer player. Other games like Paris-Paris (where the branching factor at move nodes is at most 3 for a 2-player game, but the branching factor at chance nodes can be as high as 35 choose 3) would use *-Minimax in the same way: primarily for mid- to end-game play, after the board begins to take form.

The *-Minimax algorithms seem to also be applicable to MDPs, especially in the area of multi-agent MDPs. While solving MDPs usually involves an Expectimax-type evaluation

of states one step away during value iteration, perhaps that component could be changed to a depth-N search of states, where the action at any given state would be determined by the current policy at that iteration. This may produce quicker convergence, or in the case of multi-agent MDPs, a better method for choosing actions that lead to higher rewards.

A general approach to solving games that combine elements of skill and chance will remain an open research problem for a while to come, but they provide some of the most interesting domains as they often have elements at which computers excel but humans don't (optimization, uncertainty calculation), and vice-versa (long-term planning, opponent modeling). Games that combine skill, chance, imperfect information and opponent interaction are the most difficult domains for computers, so cross-disciplinary approaches involving combining elements of heuristic search, machine learning, agent theory, game theory and even psychology may prove the most fruitful in the years to come.

Bibliography

[1] Victor Allis. A Knowledge-based Approach of Connect-Four. Master's thesis, Vrije Universiteit, October 1988.

[2] GNU Backgammon (backgammon software). http://www.gnubg.org/.

[3] Jellyfish (backgammon software). http://jelly.effect.no/.

[4] Snowie (backgammon software). http://www.bgsnowie.com/.

[5] Bruce W. Ballard. The *-Minimax Search Procedure for Trees Containing Chance Nodes. *Artificial Intelligence*, 21(3):327–350, 1983.

[6] H. Berliner. Backgammon Computer Program Beats World Champion. *Artificial Intelligence*, 14:205–220, 1980.

[7] Darse Billings, Aaron Davidson, Jonathan Schaeffer, and Duane Szafron. The Challenge of Poker. *Artificial Intelligence Journal*, 134(1-2):201–240, 2002.

[8] Michael Buro. ProbCut: An Effective Selective Extension of the Alpha-Beta Algorithm. *ICCA Journal*, 18(2):71–76, 1995.

[9] Michael Buro. The Othello Match of the Year: Takeshi Murakami vs. Logistello. *ICCA Journal*, 20(3):189–193, 1997.

[10] Michael Buro. From Simple Features to Sophisticated Evaluation Functions. In H. J. van den Herik and H. Iida, editors, *Proceedings of the First International Conference on Computers and Games (CG-98)*, volume 1558, pages 126–145, Tsukuba, Japan, 1998. Springer-Verlag.

[11] Matthew L. Ginsberg. GIB: Imperfect Information in a Computationally Challenging Game. *Journal of Artificial Intelligence Research*, 14:303–358, 2001.

[12] R.D. Greenblatt, D.E. Eastlake, and S.D. Crocker. The Greenblatt Chess Program. In *Fall Joint Computer Conference 31*, pages 801–810, 1967.

[13] P. E. Hart, N. J. Nillson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[14] Michael Howard. The Duel, August 5, 2003. [Online] rec.games.backgammon.

[15] D. Knuth and R. Moore. An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[16] R. E. Korf. Iterative-deepening A*: An optimal admissible tree search. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 1034–1036, 1985.

[17] Allis L.V., van den Herik H.J., and Huntjens M.P.H. Go-Moku Solved by New Search Techniques. In *Proceedings of the 1993 AAAI Fall Symposium on Games: Planning and Learning*, pages 1–9, 1993.

[18] D. Michie. Game-playing and game-learning automata. In L. Fox, editor, *Advances in Programming and Non-Numerical Computation*, pages 183–200. Pergamon, New York, 1966.

[19] Martin Mueller. Computer Go. *Artificial Intelligence*, 134(1-2):145–179, 2002.

[20] M. Newborn. *Kasparov versus Deep Blue: Computer Chess Comes of Age.* Springer-Verlag, 1997.

[21] John W. Romein and Henri E. Bal. Solving the Game of Awari using Parallel Retrograde Analysis. *IEEE Computer*, 36(10):26–33, October 2003.

[22] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 1995.

[23] Arthur L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–219, July 1959.

[24] Jonathan Schaeffer. The History Heuristic and the Performance of Alpha-Beta Enhancements. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(11):1203–1212, 1989.

[25] Jonathan Schaeffer. *One Jump Ahead: Challenging Human Supremacy in Checkers.* Springer-Verlag, 1997.

[26] Jonathan Schaeffer, Yngvi Bjornsson, Neil Burch, Rob Lake, Paul Lu, and Steve Sutphen. Building the Checkers 10-Piece Endgame Databases. *10th Advances in Computer Games (ACG)*, November 2003. To appear.

[27] Brian Sheppard. *Towards Perfect Play of Scrabble.* PhD thesis, Universiteit Maastricht, July 2002.

[28] D. Slate and L. Atkin. Chess 4.5: The Northwestern University Chess Program. In P. Rey, editor, *Chess Skill in Man and Machine*, pages 82–118. Springer Verlag, New York, 1977.

[29] Michael Strato. The history of backgammon. http://www.gammonvillage.com/.

[30] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, 1998.

[31] Gerald Tesauro. Neurogammon: A neural-network backgammon learning program. In *Heuristic Programming in Artificial Intelligence*, pages 78–80, 1989.

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

[33] Gerald Tesauro. Programming Backgammon Using Self-Teaching Neural Nets. *Artificial Intelligence*, 134(1-2):181–199, 2002.

[34] R. Thomas and K Hammond. Java Settlers: A Research Environment for Studying Multi-Agent Negotiation. In *Proceedings of Intelligent User Interfaces*, pages 240–240, 2002.

[35] Ken Thompson. *Advances in Computer Chess 3*, chapter Computer Chess Strength, pages 55–56. Pergamon Press, 1982. edit. M.R.B. Clarke.

[36] Jørn Thyssen, 2003. Personal Communication.

[37] A. L. Zobrist. A new hashing method with applications for game playing. Technical report, Department of Computer Science, University of Wisconsin, Madison, 1970.

Appendix A

Negamax Formulation of Search Functions

A.1 Negamax Formulation of Alpha-Beta

Most game programs do not use the regular form of Alpha-Beta (described back in Figure 2.4), because enforcing the evaluation function to always report a value in terms of the root player is non-intuitive. There really isn't any difference between the code for Min nodes and the code for Max nodes, except they are doing things oppositely, because the evaluation function gives scores relative to the root player. However, if we change our evaluation function to always score a node based on the player-to-move, then every node in the search becomes a Max node; either the first player is maximizing, or the second player is maximizing. If the layers of Max and Min nodes strictly alternate, then a backed up value simply needs to be negated by the parent node, since what is good for the opponent is bad for that player, and vice-versa. The last thing we need to do is think about the alpha and beta values for the window. Since the original implementation adjusts only the alpha values for Max nodes, and since the next level down is for the opposite player, we will therefore swap and negate the values of alpha and beta when we pass them down to the child. By negating every returned value, eliminating the `is_max_node` branch, and swapping the alpha and beta window bounds, we end up with the Negamax formulation of Alpha-Beta as described in [15] in Figure A.1.

A.2 Negamax Formulation of Expectimax, Star1 and Star2

Now that we have adjusted the formulation for Alpha-Beta, we need to adjust the algorithms used at chance nodes. First, we notice that since the children of a chance node in a regular *-Minimax tree will be for the same player (the chance node may just represent the player throwing dice, but they still haven't moved), so we don't need to adjust the return values

```
float nAlphaBeta(Board board, float alpha, float beta, int depth) {
   if(terminal(board) || depth == 0) return (evaluate(board));

   N = numSuccessors(board);
   score = -INFINITY;
   for(i = 1; i <= N; i++) {
      v = -nAlphaBeta(successor(board,i), -beta, -alpha, depth-1);
      if(v > score) score = v;
      if(score > alpha) alpha = score;
      if(alpha >= beta) return (score);
   }

   return (score);
}
```

Figure A.1: Negamax formulation of the Alpha-Beta algorithm

from the Min or Max nodes at all. We also don't want to change the order of the alpha or beta values since, again, the player hasn't changed yet. For Expectimax, nothing needs to change; we can use the same code. For Star1, nothing needs to change either, since Star1 is already "agnostic" about what types of nodes proceed it. However, Star2 needs to be changed. Since we are always trying to maximize at each step, we simply need a single version of Star2 for chance nodes followed by Max nodes, the implementation of which is shown in Figure A.2. This implementation assumes search in regular *-Minimax trees, so the call to `search` can be replaced with a call to `nAlphaBeta_MM`, a negamax version of Alpha-Beta which will call `nStar2` instead of itself. A new version of `Probe` is also needed, `nProbe`, which will just be `Probe_Max`, except it will call `nAlphaBeta_MM`. Most of this code should already be present in the form of the version of Star2 for chance nodes followed by Max nodes, `Star2_Max`.

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

Figure A.2: Negamax formulation of the Star2 algorithm

Appendix B

Dice

Table B.1: Tournament results for 4-in-a-row on a 7x7 board, 18,000 games per matchup

	d=1	d=3	d=5	d=7	d=9
1	*	51.83%	63.52%	64.32%	65.40%
3	48.17%	*	61.82%	62.63%	63.78%
5	36.48%	38.18%	*	50.54%	52.12%
7	35.68%	37.37%	49.46%	*	51.83%
9	34.60%	36.22%	47.88%	48.17%	*

Table B.2: Probe efficiency for Dice

	d=3		d=5		d=7		d=9	
	μ	σ	μ	σ	μ	σ	μ	σ
5x5	66.8%	17.3%	48.4%	7.7%	48.4%	5.8%	47.4%	3.3%
7x7	75.0%	13.3%	53.9%	8.4%	59.4%	5.5%	58.1%	3.5%
9x9	79.3%	11.9%	57.5%	9.2%	66.5%	5.4%	64.9%	3.7%
11x11	82.0%	10.2%	59.8%	10.2%	71.1%	6.2%	69.1%	4.3%
13x13	83.1%	10.1%	62.2%	11.3%	73.9%	7.7%		
15x15	85.1%	8.1%	63.2%	11.3%	77.4%	8.1%		
17x17	85.4%	9.0%	64.8%	12.3%	79.8%	7.7%		

	d=11		d=13		d=15	
	μ	σ	μ	σ	μ	σ
5x5	46.8%	3.1%	47.8%	2.6%	46.1%	2.8%
7x7	57.1%	3.1%				
9x9						
11x11						
13x13						
15x15						
17x17						

Table B.3: Average time (s) for board size of 5x5, over 500 positions

	Expectimax		Star1			Star2		
	μ	σ	μ	σ	%	μ	σ	%
d=11	5.5	2.7	4.5	2.2	82	3.2	1.5	58
d=13	38.1	24.4	34.3	22.2	90	25.4	17.0	67
d=15	365.6	335.4	333.7	298.7	91	194.9	170.8	53

Table B.4: Average node expansions for board size of 5x5, over 500 positions

	Expectimax		Star1			Star2		
	μ	σ	μ	σ	%	μ	σ	%
d=1	4	1	4	1	100	4	1	100
d=3	118	24	109	22	92	85	17	72
d=5	2397	634	2024	528	84	1239	317	52
d=7	42073	13910	32154	10706	76	14445	4626	34
d=9	412521	165125	327932	134161	79	169668	62070	41
d=11	3485102	1692682	2796416	1372936	80	1498603	694405	43
d=13	23860669	15444789	21437750	13979997	90	12175740	7850960	51
d=15	230937374	213184303	210644039	190750225	91	95343501	81808390	41

Table B.5: Average time (s) for board size of 7x7, over 500 positions

	Expectimax		Star1			Star2		
	μ	σ	μ	σ	%	μ	σ	%
d=7	1.1	0.3	1.0	0.1	91	1.0	0.0	91
d=9	20.9	6.2	15.8	5.3	76	5.8	2.5	28
d=11	483.0	195.9	356.9	157.3	74	97.6	45.9	20

Table B.6: Average node expansions for board size of 7x7, over 500 positions

	Expectimax		Star1			Star2		
	μ	σ	μ	σ	%	μ	σ	%
d=1	6	1	6	1	100	6	1	100
d=3	299	54	276	49	92	161	38	54
d=5	12114	2648	10196	2327	84	3871	965	32
d=7	448736	117863	339354	98614	76	85446	31527	19
d=9	9109391	2904555	6940886	2464782	76	1878648	770055	21
d=11	208791749	91060266	155518292	72987326	74	31680234	14846272	15

Table B.7: Average time (s) for board size of 9x9, over 500 positions

	Expectimax		Star1			Star2		
	μ	σ	μ	σ	%	μ	σ	%
d=7	8.6	1.8	6.4	1.7	74	1.5	1.1	17
d=9	314.7	80.5	230.9	74.0	73	45.4	26.5	14

Table B.8: Average node expansions for board size of 9x9, over 500 positions

	Expectimax		Star1			Star2		
	μ	σ	μ	σ	%	μ	σ	%
d=1	8	1	8	1	100	8	1	100
d=3	605	93	563	86	93	260	62	43
d=5	40632	7488	34223	6856	84	9103	2004	22
d=7	2556033	560494	1908067	511270	75	307370	138409	12
d=9	91955795	25388492	68103347	23112399	74	10166029	5389658	11

Table B.9: Average time (s) for board size of 11x11, over 500 positions

	Expectimax		Star1			Star2		
	μ	σ	μ	σ	%	μ	σ	%
d=7	45.7	7.6	34.3	7.2	75	5.2	2.8	11
d=9	2964.8	661.3	2088.3	597.7	70	240.5	160.5	8

Table B.10: Average node expansions for board size of 11x11, over 500 positions

	Expectimax		Star1			Star2		
	μ	σ	μ	σ	%	μ	σ	%
d=1	9	1	9	1	100	9	1	100
d=3	1073	154	1008	142	94	386	90	36
d=5	108631	18168	92663	17124	85	18791	4076	17
d=7	10440509	2008884	7901976	1879951	76	931700	505976	9
d=9	674004977	172284142	478594692	152279828	71	43650330	28791026	6

Table B.11: Average time (s) for board size of 13x13, over 500 positions

	Expectimax		Star1			Star2		
	μ	σ	μ	σ	%	μ	σ	%
d=5	1.4	0.5	1.2	0.4	86	1.0	0.0	71
d=7	190.3	27.2	143.8	27.0	76	15.4	8.7	8

Table B.12: Average node expansions for board size of 13x13, over 500 positions

	Expectimax		Star1			Star2		
	μ	σ	μ	σ	%	μ	σ	%
d=1	11	1	11	1	100	11	1	100
d=3	1742	224	1643	211	94	555	162	32
d=5	247808	36386	212209	34752	86	35024	8590	14
d=7	33770133	5639624	25635916	5446810	76	2251628	1329830	7

Table B.13: Average time (s) for board size of 15x15, over 500 positions

	Expectimax		Star1			Star2		
	μ	σ	μ	σ	%	μ	σ	%
d=5	3.6	0.6	3.1	0.6	86	1.0	0.0	28
d=7	663.5	88.0	499.0	91.8	75	39.7	26.9	6

Table B.14: Average node expansions for board size of 15x15, over 500 positions

	Expectimax		Star1			Star2		
	μ	σ	μ	σ	%	μ	σ	%
d=1	13	1	13	1	100	13	1	100
d=3	2662	301	2525	287	95	722	174	27
d=5	504294	65459	432390	65444	86	58704	13254	12
d=7	92105475	13688189	69644237	14109164	76	4685333	3303548	5

Table B.15: Average time (s) for board size of 17x17, over 500 positions

	Expectimax		Star1			Star2		
	μ	σ	μ	σ	%	μ	σ	%
d=5	8.9	1.2	7.6	1.1	85	1.1	0.5	12
d=7	2114.7	275.8	1598.4	282.5	76	102.0	75.1	5

Table B.16: Average node expansions for board size of 17x17, over 500 positions

	Expectimax		Star1			Star2		
	μ	σ	μ	σ	%	μ	σ	%
d=1	14	2	14	2	100	14	2	100
d=3	3795	429	3617	411	95	942	259	25
d=5	925415	115862	797480	114984	86	94260	23966	10
d=7	218749807	30405596	166155134	31485764	76	9113269	6810862	4

Table B.17: Average node expansions for different move orderings for board size of 11x11, over 500 positions

	d=3		d=5		d=7		d=9	
	μ	σ	μ	σ	μ	σ	μ	σ
none	469	153	25184	9322	1889848	898560	136621337	60848820
random	443	134	23224	7703	1794483	765833	140430139	58851717
static	367	82	16324	3863	1443771	525432	111554612	42970424
quick	386	90	16602	3886	1437847	520116	109955050	42504397

Table B.18: Average time (s) for different move orderings for board size of 11x11, over 500 positions

	d=7		d=9	
	μ	σ	μ	σ
none	8.1	3.9	595.8	262.4
random	7.8	3.4	616.3	254.9
static	8.4	4.6	582.1	263.7
quick	6.5	2.5	495.3	193.8

Table B.19: Probe efficiency for different move orderings for board size of 11x11, over 500 positions

	d=3		d=5		d=7		d=9	
	μ	σ	μ	σ	μ	σ	μ	σ
none	63.6%	24.2%	60.7%	12.9%	61.0%	10.9%	59.7%	9.9%
random	69.8%	18.9%	67.6%	9.3%	68.1%	7.0%	67.4%	6.2%
static	85.4%	8.1%	95.7%	2.7%	96.0%	1.7%	95.5%	1.8%
quick	82.0%	10.2%	93.3%	3.3%	93.5%	2.5%	91.5%	2.8%

Appendix C

Backgammon

Table C.1: Average node expansions over 500 contact positions

	Expectimax		Star1			Star2		
	μ	σ	μ	σ	%	μ	σ	%
d=1	33	51	33	51	100	33	51	100
d=3	12287	17020	11372	15602	93	3544	4668	29
d=5	6478981	11122146	5297752	9261568	82	526042	860694	8

Table C.2: Tournament results for Gnubg with no noise versus Gnubg with no noise, 18000 games per matchup

	1	3	5
1	*	50.92%	51.79%
3	49.08%	*	51.63%
5	48.21%	48.37%	*

Table C.3: Tournament results for Star2 with no noise versus Star2 with no noise, 2000 games per matchup

	1	3	5
1	*	50.60%	51.60%
3	49.40%	*	52.52%
5	48.40%	47.48%	*

Table C.4: Tournament results for Star2 with n=0.0150 versus Star2 with n=0.0150, 1000 games per matchup

	1	3	5
1	*	55.53%	54.67%
3	44.47%	*	51.57%
5	45.33%	48.43%	*

Table C.5: Tournament results for Star2 with n=0.0300 versus Star2 with n=0.0300, 200 games per matchup

	1	3	5
1	*	59.00%	64.20%
3	41.00%	*	53.40%
5	35.80%	46.60%	*