UNIVERSITY OF CALIFORNIA

Los Angeles

Multi-Player Games: Algorithms and Approaches

A dissertation submitted in partial satisfaction of the

requirements for the degree Doctor of Philosophy

in Computer Science

by

Nathan Reed Sturtevant

2003

The dissertation of Nathan Reed Sturtevant is approved.

Adnan Darwiche

Judea Pearl

William R. Zame

Richard E. Korf (Committee Chair)

University of California, Los Angeles

2003

Contents

V. Pruning Algorithms

VI. Additional Techniques for Improving Performance

VII. Experimental Results

VIII. Contributions, Conclusions, and Future Work

Index of Tables and Figures

VITA

June 17, 1974	Born, Ventura, California
1996	B.S., Electrical Engineering and Computer Science With Honors University of California, Berkeley Berkeley, CA
1998-99	Research Assistant Computer Science Department Univeristy of California, Los Angeles
1998-2001	Teaching Assistant (Spring Quarter) Computer Science Department University of California, Los Angeles
2000	M.S., Computer Science University of California, Los Angeles Los Angeles, CA
2000-03	Research Assistant Computer Science Department University of California, Los Angeles

PUBLICATIONS

Sturtevant, N., Tang, N., Zhang, L., The Information Discovery Graph: Towards a Scalable Multimedia Resource Directory, Proceedings of the IEEE Workshop on Internet Applications (WIAPP), July, 1999.

Sturtevant, N., Korf, R., On Pruning Techniques for Multi-Player Games, Proceedings, AAAI-2000, Austin, Tx, pp 201-207.

Sturtevant, N., A Comparison of Algorithms for Multi-Player Games, Proceedings of the 3rd International Conference on Computers and Games, 2002.

Sturtevant, N., Last-Branch and Speculative Pruning Algorithms for Max^n, Proceedings, IJCAI-2003, Acapulco, Mexico.

ABSTRACT OF THE DISSERTATION

Multi-Player Games:

Algorithms and Approaches

by

Nathan Reed Sturtevant

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2003

Professor Richard Korf, Chair

Historically, much work in Artificial Intelligence research has gone into designing computer programs to play two-player perfect-information games such as Chess, Checkers, Backgammon, and Othello. Comparatively little work, however, has gone into multi-player games such as Chinese Checkers, Abalone, Cribbage, Hearts, and Spades. As a result, we have highly optimized techniques for two-player games,

but very little knowledge of how they work in multi-player games.

In this thesis we extend many of the standard techniques from two-player games to multi-player games. We present two decision rules, max^n [Luckhardt and Irani, 1986] and paranoid, examining their theoretical properties. For max^n we also introduce several pruning techniques, including Alpha-Beta Branch-and-Bound pruning and Speculative pruning. Speculative pruning is the first multi-player pruning algorithm that can prune any constant-sum multi-player game, and provides an order of magnitude reduction in node expansions over previous search techniques in games like Chinese Checkers.

We also analyze the properties of common two-player game techniques, such as zero-window search and iterative deepening, showing how their properties change in multi-player games. Finally, we present results of all these techniques in a variety of multi-player domains, including Chinese Checkers, Abalone, Cribbage, Hearts and Spades. These methods have allowed us to write state-of-the-art programs for playing Hearts and a version of Chinese Checkers on a smaller board.

Chapter 1

Introduction

1.1 Motivation

Researchers in the field of Artificial Intelligence (AI) have always been interested in the question of how computers can be taught, can learn, or be programmed to do the things that humans do. Things that are exceptionally easy for computers, like mathematical computation and data storage, are very difficult for humans. But, things that humans do well and often take for granted, such as carrying on a conversation, are much more difficult for a computer.

One difficult task that AI researchers have been interested in since before the inception of the field is that of game playing. Shannon proposed a chess machine as early as 1950 [Shannon, 1950], Turing wrote a program to play chess in 1951 [Turing et al., 1953], but never ran it on a computer, and Samuel had a quality checkers player by the early 1960's [Samuel, 1959]. While some may dismiss these as merely "parlor" games, the video game industry currently makes more money each year than Hollywood [Baird, 2003]. This in itself does not justify game research, but it does mean that games are something that people have a fundamental interest in.

For instance, Electronic Arts CTO, Steve Anderson, recently claimed that Electronic Arts feels that developing good Artificial Intelligence systems for their games is one of their major challenges for the next decade [Anderson, 2003]. As the graphic capabilities of video game consoles have grown, it has become more and more obvious how poor the character movement and decision systems in current games actual are. In fact, many "artificial intelligence" systems in games are just large finite state machines. The artificial intelligence technology used for such systems is becoming more and more important in creating a realistic environments. Finally, the development of high-performance graphic chips is beginning to free up the CPU for other tasks such as improved AI.

Beyond the game industry, games have also played a large part in social structures throughout history. "Human fascination with game playing is long-standing and pervasive. Anthropologists have catalogued popular games in almost every culture....Games intrigue us because they address important cognitive functions." [Epstein, 1999] This suggests, and we agree, that games have value beyond pure entertainment, as they often form part of the social structures within a culture or family group.

While a great deal of research has gone into two-player games, many games and most real-world domains involve more than two parties, whether in cooperation or in competition. At first this may seem like a small difference, but as [Luce and Raiffa, 1957] point out, "it has long been recognized in sociology, and in practical affairs, that between two-person situations and those involving three or more per-

sons there is a qualitative difference which is not as simple as the difference between 2 and 3." Obviously in a two-player strictly competitive game, the strongest player will usually win. But, the introduction of a third player doesn't guarantee that the strongest player will win. In fact, many have informally observed this in the recent TV game show "The Weakest Link," where the smartest players are usually voted out of the game in the stages when they become a threat to the remaining players.

Besides a recent surge of work on agent-based systems, relatively little work has been done in what practical algorithms and techniques could actually be used to play games with more than two players.

In this thesis we lay the groundwork for extending research from two-player games to multi-player games. It is our hope and goal that in addition to providing a framework for work on multi-player games, this work would also be useful to the greater domains of decision making in contexts with multiple competing agents.

1.2 Problem Overview

If one wishes to write a high-quality two-player game playing program, there is generally a straightforward set of steps by which it can be done, starting with minimax and alpha-beta pruning, a customized evaluation function, and continuing through a list of well-researched topics. AI researchers know the basic steps, and new techniques are often just needed to fine-tune certain aspects of a game. There are notable exceptions to this, particularly Go and Shogi (Japanese Chess), but as we will discuss in later chapters, this formula has seen a fair amount of success.

Compared to this, there are relatively few answers when it comes to multi-player games. There is no standard algorithm that has been used for multi-player games, and it is unclear exactly which techniques are worthwhile to use in multi-player games.

There are three categories of multi-player games that are most widespread and that usually require computer opponents. The first category is the first-person shooter (FPS). These games usually require quick movement and reflexes, but movement and shots shouldn't be so precise as to make a human opponent always lose. Games in this category include Doom, Quake, and Half-Life. Another category is the real-time strategy (RTS) game. These games usually require you to coordinate armies of units across a battlefield for real-time combat. This category of games includes Starcraft, Warcraft, and Myth. Finally, there are board and card games in which play is less frenzied, but can often be just as intense.

Most people writing computer opponents for FPS and RTS games use fairly ad-hoc methods, but even so it is usually easy for a computer opponent to annihilate even the best human. Thus, the issue in these games is how to write a balanced opponent. In board and card games, however, it is much more difficult to develop a strong opponent, so this is where we focus our work.

1.3 Problem Approach

An important element of any game is the choice of the decision rule which is implemented by the algorithm used to search the game tree. The standard two-play-

er decision rule is minimax. In this thesis we consider two decision rules, paranoid [Sturtevant and Korf, 2000] and max^n [Luckhardt and Irani, 1985]. The paranoid decision rule results from the simplification of a multi-player game to a two-player game by assuming that all our opponents are collaborating against us, while max^n is a generalization of minimax to a *n*-player game.

The choice of decision rule may be made on the theoretical strength of the rule or its computational efficiency, but implicit in the decision rule is a model of our opponents. If we choose to use the paranoid algorithm, we are using a model in which we expect our opponents to collaborate, while the max^n decision rule assumes every player is out for themselves. Although we will not discuss opponent modeling in detail, it is important to realize that we cannot get away with the implicit assumptions we might have made in a two-player game. In a subtle way, this is an issue behind many of the difficulties that arise in multi-player games.

The minimax decision rule has dominated two-player game research because of the efficiency of alpha-beta pruning [Knuth and Moore, 1975], which allows minimax to be implemented much more efficiently than nearly any other algorithm proposed for two-player games. In practice, unfortunately, most simple analogs to alpha-beta pruning in multi-player games are either ineffective or not valid. The best previous technique, shallow pruning [Korf, 1991] has a reasonable best-case performance, but this cannot realistically be achieved in practice.

Therefore, we introduce a set of new pruning techniques which are much more effective in practice. These include branch-and-bound pruning, alpha-beta

branch-and-bound pruning [Sturtevant and Korf, 2000], last-branch pruning and speculative pruning [Sturtevant, 2003]. We also extend the previous analysis of immediate and shallow pruning, providing enhanced bounds on when they can be applied along with their effectiveness.

Besides pruning techniques, there have been many other methods used to calculate decision rules and search game trees more efficiently. These often involve extensive use of computer memory, while most decision rules require very little memory for computation. We consider some of these techniques, such as transposition tables and opening and closing books, discussing how they can be used in multi-player games, and some of the issues that we need to be aware of when using them in multi-player games.

Finally, we also implemented a range of practical games including Chinese Checkers, Hearts, Spades, Cribbage and Abalone. We use these games as a test bed to measure the efficiency and effectiveness of the techniques discussed here.

We will begin Chapter 2 with an overview of games in which computer programs play well, along with some of the techniques used in those programs. This is followed in Chapter 3 by a description of the multi-player games that we use in this thesis, along with a few other popular multi-player games.

In Chapter 4 we begin to present our new work as we cover the paranoid and $\max^n$ decision rules, followed by various pruning techniques for these algorithms in Chapter 5. Chapter 6 discusses some of the other techniques that have been used in two-player games and how they can be applied to multi-player games. Chapter 7

describes the software package we have written to play and test multi-player games, along with the results of these experiments. We conclude in Chapter 8 with a summary of our work and a description of different areas of open research.

Chapter 2

Expert-Level Two-Player Game Programs

There are many different domains in which computer programs have been written that exhibit expert-level play. The story of how this has occurred and how humans have responded to the challenge is quite interesting, and we suggest that anyone interested in the details of the story should see [Schaeffer, 2001].

In this chapter, we will discuss these domains more briefly, looking for what the key elements were that allowed these successes. This history will partly set the stage for our discussion of multi-player games, as it provides a list of techniques that have proven their worth in two-player games.

2.1 Two-Player games

Chess dominated early game research, with a truly "expert" program seeming to be about 10 years off [Newell, et al., 1958] for a long time. It wasn't until IBM's work on specialized hardware for Deep Blue in the late 1990's that the top human chess player was defeated in an exhibition match. Recent efforts on programs such as Deep Fritz and Deep Junior have begun to produce similar results on more commonly-available hardware. Expert chess programs rely on minimax

search with alpha-beta pruning, iterative deepening, opening books, end-game databases, transposition tables and quiescence (selectively deeper) search. The other important factor has been highly tuned and accurate evaluation functions.

Checkers is another game that received early consideration. Samuel's checkers program [Samuel 1959, 1967] used machine learning to come up with a good evaluation function. But, it wasn't until the late 1980's and early 1990's when computers offered serious challenges to humans. One of the primary reasons for this was the development of a massive end-game database, containing all positions with 8 or fewer pieces on the boards [Schaeffer, et al., 1992]. Chinook, the program written based on this research, was the first computer program to ever win a game against a human champion in formal match play, and it is the strongest Checkers player in the world. Chinook used minimax and alpha-beta as the basic core of its search engine.

As in Checkers, expert Othello programs are currently known to be better than all humans. The top Othello program, Logistello [Buro, 1997], learned an extremely sophisticated evaluation function for play. In fact, it actually learned 13 different evaluation functions that were used separately across different stages of the game. This, combined with traditional minimax and alpha-beta pruning has been quite successful, allowing Logistello to strongly sweep the world champion in an exhibition match in 1997.

Each of these first three games has used very similar techniques, and they also have very similar properties, all being two-player perfect information games.

The next three games we discuss are not perfect-information games.

Backgammon is a non-deterministic game, as it involves the random roll of dice. The best backgammon program, TD-Gammon [Tesauro and Sejnowski, 1989] is similar to Logistello in that it uses a learned evaluation function for play. In backgammon this is even more important, since the random dice rolls greatly restricts the amount of search possible in the game. TD-Gammon used temporal difference learning [Tesauro, 1995] to learn its evaluation function. TD-Gammon is currently playing at the level of the best backgammon players in the world.

Scrabble is a word game in which imperfect knowledge of the state of the game limits the possibility of deep search. In most of the game there are too many possible moves, much less opponent hands, to consider them all. Thus, the best Scrabble program, Maven [Sheppard, 2002], relies on sophisticated move-selection algorithms, as well as limited sampling of possible opponent responses. In an exhibition match against one of the best North American players in 1998, Maven won 9-5 [Schaeffer, 2001].

Bridge is one of the domains for which good two-player game programs have more recently been developed. GIB [Ginsberg, 2001] is generally considered to be the best Bridge program in the world, although it hasn't played any sanctioned matches against top world players, so it is difficult to know exactly where it stands. The biggest issue in Bridge is the (lack of) knowledge about your opponents' hands. GIB handles this using Monte-Carlo simulation, and more recently other imperfect information methods, which we will discuss later in this chapter. GIB also uses so-

phisticated transposition tables.

To summarize work in two-player games, all the games mentioned here use the minimax decision rule with alpha-beta pruning for efficient search. They also use a combination of transposition tables, opening/closing books, and hand-crafted evaluation functions. There are still significant challenges in two-player games which these techniques alone have not been able to solve. Specifically, games like Go and Shogi are too complicated for standard search techniques. There is presently a lot of research effort going into these games, which is not the case for multi-player games.

2.2 Extending Work Multi-Player Games

We can use some of the techniques from two-player games directly in multi-player games without much thought. Methods for learning a static evaluation function are likely to be useful in both two-player and multi-player games, as well as techniques like Monte-Carlo sampling.

Despite this, there isn't a single multi-player game for which expert-level programs have been written. Probably the most skilled program in a multi-player game is Loki [Billings, et al., 2002], which plays Texas Hold 'em Poker, although it has not yet played against humans for money, which will be a true test of its skill. Poker is one of the more widely played multi-player games, and we will discuss briefly in the next chapter.

Given the lack of work in multi-player games, there are many unanswered

question about how to play these games in practice. Prior to this work, there was limited research into multi-player game tree search. Specifically, papers addressed the max^n algorithm [Luckhardt and Irani, 1986], the introduction of shallow pruning and the incorrectness of deep pruning [Korf, 1991], along with some work on showing that max^n can suffer from game tree pathologies [Mutchler, 1993]. However, this in itself is hardly sufficient to provide a basis for writing an expert-level multi-player game. The later chapters of this thesis will extend many of the two-player techniques found in the previous section to multi-player games.

2.3 Solving Perfect and Imperfect Information Games

Most search algorithms are designed for perfect-information games such as Chinese Checkers. To play imperfect-information games such as Spades or Hearts, we must either modify the standard algorithms to allow for imperfect information, or use new algorithms designed for imperfect-information games.

If, in a card game, we could see our opponents' cards, we would be able to use standard search algorithms such as minimax to play the game. While in most games we don't know the exact cards our opponent holds, we do have an estimate of the probability of our opponent holding any particular hand or card. Thus, we can create a hand that should be similar to what our opponent holds, and use a perfect-information algorithm to play against it.

The full expansion of this idea results in Monte-Carlo sampling. Instead of generating just a single hand, we generate a set of hands that are representative of

the actual hand we expect our opponent to have. In Bridge, for instance, we would generate hands that are consistent with the bidding and play so far. We then solve each of these hands using the standard minimax algorithm. When we have completed the analysis of each hand, we combine and analyze the results from each hand to produce our next play. As play continues we update our models to reflect the plays made by our opponent.

Many of the imperfect information games we consider in this thesis are card games, and so they resemble Bridge in terms of their imperfect information. Thus, in much of our work we will discuss all games as if they are perfect information games, making the assumption that, as in Bridge, we will be able to use perfect-information methods to play imperfect information games.

Chapter 3

Multi-Player Games

There are a wide variety of multi-player games. We begin by covering many of these games here, paying particular attention to the games we have spent more time investigating. A secondary purpose is to catalogue some of the interesting multi-player games which have not been studied in depth.

It is important to note that many games such as Bridge are played with four players in two teams. While these games have multiple players, the players are in two coalitions, and so regular two-player game theory applies.

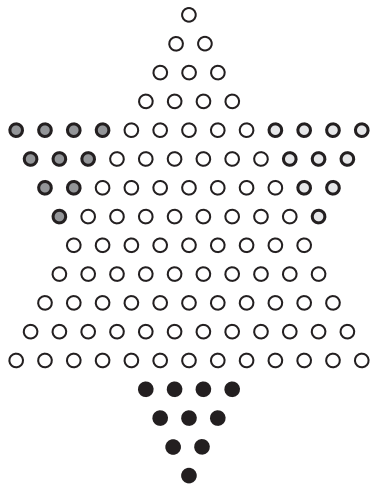

Figure 3.1: A Chinese Checkers board with three players.

3.1 Chinese Checkers

Chinese checkers is a perfect information game for 2-6 players. A Chinese Checkers board is shown in Figure 3.1. The goal of the game is to get 10 pegs or marbles from one's starting position to one's ending position as quickly as possible. The positions vary depending on how many players are in the game. Starting and ending positions are always directly across from each other on the board, and players are placed as symmetrically as possible around the board. In a two-player game, the players would start at the top and bottom of the board, while in a three-player game the players begin in alternating corners of the board.

Pegs are moved by stepping to an adjacent position on the board or by jumping over adjacent pegs. One can jump over any player's pegs, or chain together several jumps, but pegs are not removed from the board after a jump. We demonstrate this with a set of consecutive moves in Figure 3.2. The first move is a simple jump

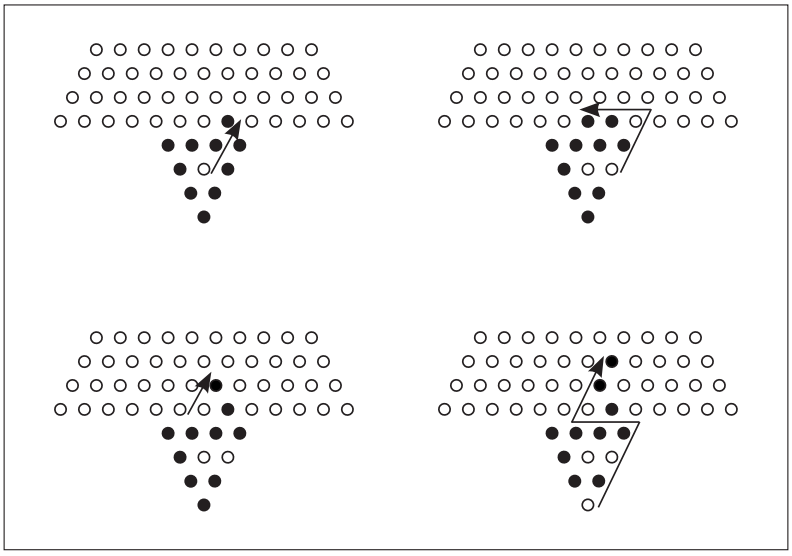

Figure 3.2: Move possibilities to start a game of Chinese Checkers.

over a single piece. The second move is a double jump. The third move is a simple move to an adjacent space, and the last move involves 4 hops. Players can chain together as many jumps as they wish over both their own pieces and their opponents pieces, as long as every segment of the jump is from an empty space to another empty space over a single peg. There are no forced moves in Chinese Checkers, as there are in Checkers.

3.2 Abalone

Abalone, like Chinese Checkers, is a perfect-information board game for 2-6 players. Abalone is a more recent game which became popular in the mid-1990's. An example of an abalone board is shown in Figure 3.3. The goal of the game is to push 6 of your opponents pieces off the edge of the board.

A player can move 1-3 pieces on his turn. There are two classes of moves allowed in an abalone game, which are illustrated in Figure 3.4. The first class of moves are simple moves of one's own pieces. This involves moving a row of one, two, or three linearly contiguous pieces together one space along any axis. In the

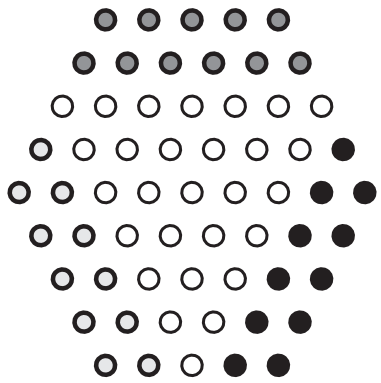

Figure 3.3: A three-player abalone game board.

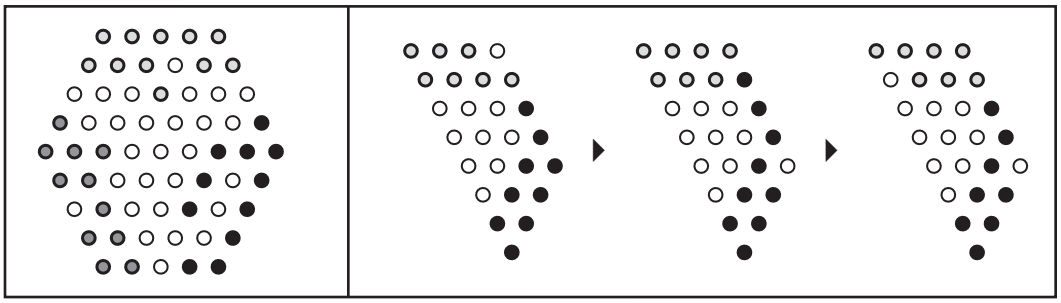

Figure 3.4: Example moves from the game of Abalone.

first example in Figure 3.4 each player has made a move of this class on the board. The player at the top has moved a single pieces down and to the left. The player on the bottom left has moved two pieces along the axis of those pieces one space upwards. The bottom right player has moved three pieces together up and to the left.

The second class of moves allowed in abalone are push moves. These are illustrated by the right half of Figure 3.4. The first move is made by the bottom right player, using three of his pieces to push one of the upper players pieces upwards. Then, the upper player responds by using three of his pieces to push his opponent's piece off the board. When making push moves, one must push pieces along the same axis of the pieces being moved, and you must always push less pieces than you are moving yourself. So, three pieces can push one or two pieces, and two pieces can push one piece, but you can never use four pieces to push three pieces.

3.3 Card Games

Card games have many similarities which we will consider first before discussing any particular games in detail. We break them into two classes here, trick-based games and non-trick-based games. Trick-based games include Hearts, Spades

and Pinochle, while Cribbage, Uno and others are not trick based.

In a trick-based game, cards are dealt out to each player face down before the game begins. The first player plays (*leads*) a card face-up on the table, and the other players follow in order, playing the same suit as lead if possible. A play out of suit is called a "slough." When all players have played, the player who played the highest card in the suit that was led "wins" or "takes" the trick. He then places the played cards facedown in his discard pile, and leads the next trick. This continues until all cards have been played. Some games contain "trump" suits. If a trick contains a trump card, the highest trump card played will win the trick, regardless of the suit lead. In trick games, points are assigned by either the number of tricks taken or the point value of each card taken in a trick.

Most trick games are divided into sets of hands. Each hand represents one deal of the cards, and the play of those cards according to the rules of the game. After a hand is played out, points are added to player's total scores, the cards are shuffled, dealt, and then the next hand begins. Players continue this process until at the end of a hand one player's score surpasses a predetermined limit, at which point the game is over.

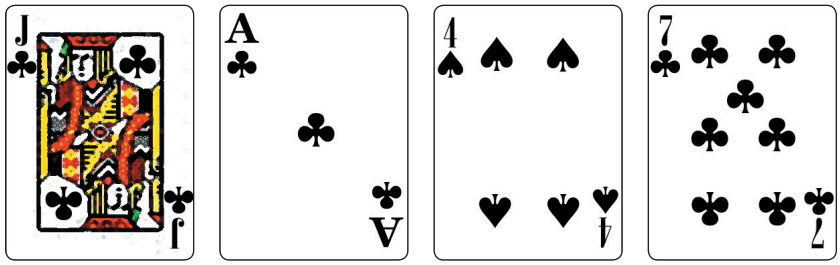

Figure 3.5: A sample card trick.

We demonstrate a sample trick in Figure 3.5. In this trick, the first player has lead the Jack of Clubs. The second player followed suit with the Ace, but the third player sloughed the Four of Spades, because he had no Clubs in his hand. If there is no trump or a suit other than spades is trump, the second player will win the trick with the Ace. But, if spades are trump, then the third player will win with the four.

We cannot cover every detail of every card game here. More details of these and many other games can be found from various sources. Currently http://www.pagat.com/ is probably the most comprehensive reference of card games available, covering many more games than found in so-called "dictionaries" of card games such as [Parlett, 1991], [Gibson, 1974] or [Hoyle, et al., 1991].

3.3.1 Hearts

Hearts is usually played with four players, although there are variations for two or more players. In Hearts there is no trump suit, and points are assigned by the cards taken in your tricks. Players get 1 point for every heart taken and 13 points for the Queen of Spades, for a total of 26 points, and the goal is to minimize the number of points taken.

In addition to these basic rules, there are several special rules in Hearts. At the beginning of each hand, cards are exchanged between the players on the table. Each player chooses 3 cards to pass, lays them down on the table, and then picks up the cards passed to them. Passing alternates from hand to hand, following a pass pattern such as "left, right, across, hold," where a hold means that no cards are passed.

In Hearts one can also “shoot the moon”, or take all the points available. When one player does this, instead of getting 26 points, the other players in the game all get 26 points each, and the player who shot the moon gets 0 points. This creates a tension between minimizing one’s points, but also assuring that no one player is able to take all the points.

Other variations that are sometimes played include getting -5 points for taking no tricks, getting -10 points for taking the Jack of Diamonds, having the player with the 2 of Clubs always lead. Some people allow a player who shoots the moon to get -26 points instead of giving other players 26 points. Hands are usually played iteratively until one player’s score reaches 100 points.

3.3.2 Spades (8-5-3 / Sergeant Major)

Spades is usually played as a four-player game in two teams. But, there is a three-player version in which players are not in teams. There are other games such as 8-5-3, also called Sergeant Major, that are very similar to Spades, and are always played by three players.

The basic goal of these games is to take as many points as possible, where each player gets one point for every trick taken. There is also a minimum number of tricks which must be taken to avoid some penalty.

In Spades there is a simplistic bidding procedure that occurs after the cards have been dealt, but before play begins. Starting with the dealer, and continuing clock-wise around the table, players announce how many tricks they plan to take in the game. Some rule variations require that the last player cannot bid such that all

players can make their bids exactly. In three-player Spades each player has 17 cards, so this would require that the bids do not sum exactly to 17.

If a player takes at least as many tricks as they bid, they get 10 points for every trick taken, plus one point for each additional over-trick. If, in the course of the game, they accumulate 10 over-tricks, they take a 100 point penalty. If a player does not make their bid, they lose 10 points for every trick bid. So, if you bid 5 tricks and take 8 you will get 53 points. If you bid 8 and take five, you get -80 points. As the name suggests, Spades is always trump.

In Sergeant Major or 8-5-3, each player must take a predetermined number of tricks (8, 5, or 3), and the person who needs to take the most number of tricks gets to declare trump. The number of tricks you need to take rotates each hand. If a player didn't make their minimum required score on the previous hand they must trade some of their cards to the player(s) who took the extra tricks. This is done by having the player who took the extra tricks give a single card to the losing player. This player then must pass the highest card in their hand from the same suit back to the winning player. Play can continue until a player reaches a fixed score, or until one player is able to take all the tricks.

3.3.3 Cribbage

Cribbage is the first of the card games we consider that isn't trick based, although cards are played in sequence by players in the game until no player can make a legal move. Like the other card games we've described here, cribbage has variations for 2-4 players, and the 3-player version isn't played in teams. We cover

the rules for the 3-player game here. Cribbage has three stages, the “discard”, the “play”, and the “show”. In the first stage players are dealt five cards, one of which they discard face-down into a “crib”. One card is also dealt into the crib so that, along with the discards, an additional four-card hand is formed. After all players have discarded, one more card, called the “start” card is selected randomly from the deck and placed face up on the top of the undealt cards. All players use this card as part of their hand in the “show”.

In the play, players play their cards out consecutively. Each time a card is played, a player says the sum of all cards played to that point. The sum of cards is not allowed to exceed 31. If a player has no cards left to play or cannot play without the sum exceeding 31 points he instead says “go”. When all players have said “go”, play continues with the sum reset to zero. During the play, points are awarded if consecutive plays form a run of three or more cards, if consecutive cards match, or if the sum of cards so far is 15 or 31. We demonstrate this in Figure 3.6.

In this play of cards the first player would play their card and then say “four.” The second player would follow with their card saying “ten.” The third player would smile and say “15 for two (points) and a run of three for five (total points).”

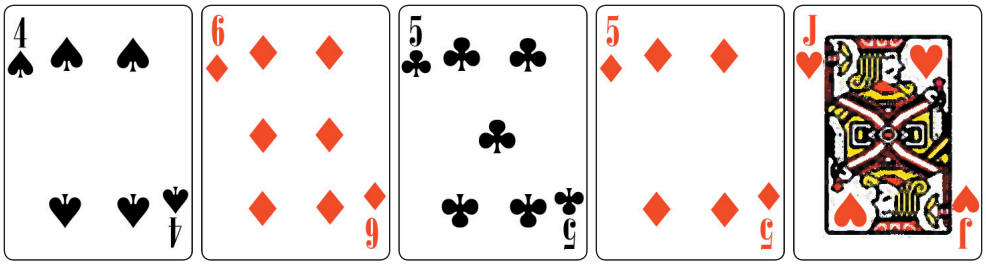

Figure 3.6: Sample play of cards from “the play” in cribbage.

Whenever a set of consecutively played cards can be sorted into a run of three or more cards, a player scores one point for every card in that run, in this case 3 points. In addition, since the sum reached 15, another two points are awarded. The first player would then continue by playing their five, saying "twenty for two," as the last two cards form a pair. Three-of-a-kind is worth 6 and four-of-a-kind is worth 12. The second player continues with the jack saying "thirty." If nobody has an ace, they will each say "go", and the second player will collect one point for playing the last card.

In the "show", the players sum up the points from the cards they originally had along with the start card, with the dealer also scoring the hand formed by the cards that were discarded. Points are awarded similarly to the play except that players can use all arrangements of their hand that will give them points. These include straights, combinations of cards that add to fifteen, a four-card flush in your hand or a five-card flush in your hand including the start card. Additionally, the jack of the same suit as the start card is worth one point. For instance, if your four-card hand plus the start card contained the same cards as Figure 3.6, it would be worth 16 points: two runs of three for 6, four ways to get fifteen for 8, and a pair for 2. The game ends as soon as any player reaches 121 points.

3.4 Other Games

Besides the games considered so far, there are other popular games we will not consider in detail in this thesis. We describe them briefly here, both for the sake

of completeness, but also so that we can reference them when we speak to the applicability of different pruning techniques later in this thesis.

3.4.1 Pinochle

Pinochle is a popular card game that has similarities to both Hearts and Spades. Like Spades, Pinochle has a bidding procedure, except in Pinochle the player winning the bid gets to set trump. In the main play of Pinochle, players are trying to maximize the number of points they take. But, more like Hearts, points are awarded based on the cards taken in each trick. Specifically, the Ace, Ten and King of each suit are worth one point, while all other cards are worth nothing.

3.4.2 Big-2 / Feudal Wars / Da Er

This class of games has many variations played all over the world. It is especially popular in China. It has a flavor of trick-based games, but also a flavor of poker. The goal of the game is to play out all your cards as quickly as possible. Plays are made by leading out a set of cards that match one of the well-known hands found in five-card stud poker. These include singles, pairs, straights, full-house, and so on.

The first player chooses their lead, and other players then follow in a manner that will always increase the value of the hand, matching a pair with a higher pair, a full house with a higher full house, etc. A player can pass at any time in a round and then join in later if the play returns to them. When all players pass, the last player to play gets to lead the next round. The game is called Big-2, because the two counts as the high card instead of low card.

There are variations on the game that allow for passing of cards similar to Sergeant Major and variations on what sets of cards can be played when. For instance, some rules allow a flush to beat a straight, while other rules require that once a straight is played all players must follow with a straight.

3.4.3 Uno / Crazy 8's

Uno and Crazy 8's are similar games in which players are simply trying to get rid of all the cards in their hand. Instead of following the lead in a trick, there is a single card showing at all times. When it is a player's turn they must play a card of the same suit or rank on top of the showing card. If they cannot play a card they must draw cards until they can play. Play continues until someone runs out of cards. In Uno you must say "uno" when you have one card left, and there are wild cards, and cards which force your opponents to draw. In Crazy 8's, the rules are similar except that when you play an eight you can call any suit as the next suit to be played.

3.4.4 Poker

Poker is probably the most popular card game used in gambling. There are many variations played, and it is one of the few multi-player games on which researchers have focused. There are many interesting issues in writing a high-performance Poker program, including handling imperfect information and opponent modeling. In fact, these issues dominate over the more traditional search techniques presented here. Because of this, we will not consider poker in this work. The current best Poker program is Loki, described in [Billings, et al., 1999] and [Billings, et al., 2002].

Chapter 4

Decision Rules

The first thing needed to play any game is a strategy. From a game-theoretic perspective, a strategy usually requires that a player explicitly enumerate all possible states of a game that they could face, along with the decision they would make at those points. While this may be feasible for small games, it is not feasible for most real-life games. However, a strategy is defined implicitly for a game by a *static evaluation function* and a *decision rule*. The evaluation function assigns a value or utility to each leaf node in the game tree, and the decision rule dictates how the values are propagated up the game tree. A static evaluation function estimates the utility of a state through human expert-level knowledge or through learned attributes. In chess, for instance, experts often speak of the material value of the pieces on the board, which is often used as part of an evaluation function. Each piece can be given a value, such as 1 for a pawn, 3 for a knight or bishop, 5 for a rook and 9 for a queen. A simple evaluation function would just return the difference in the material value between the two players. Thus, these two components are used together to dictate the strategy for play in a game.

The static evaluation function used in a game is very domain dependant, while decision rules are generic and can be applied across many domains. So, the most fundamental choice we must make when writing a program to play a game is what decision rule we will use to guide our moves in the game. Once this is done, we can use a variety of methods to craft the best evaluation function possible for any particular domain. We obviously would like to choose a decision rule with strong theoretical properties, although properties of a decision rule which are strong in theory may not be so in practice.

We begin this chapter with a review of the standard 2-player decision rule, minimax. We then discuss decision rules that can be used for games with more than two players. In this chapter we will just discuss decision rules and their theoretical properties. Other features, such as pruning techniques, will be discussed in Chapter 5 and following chapters.

4.1 Two-Player Minimax

The minimax algorithm is a simple but powerful decision rule for searching two-player game trees. It can be used for playing strictly competitive (also called zero-sum) two-player games. In a zero-sum game the players have symmetric utility functions for the leaf nodes in the tree. The minimax procedure has been studied in depth, and we will not cover all of its properties here. Many more details can be found in [Luce and Raiffa, 1957].

We illustrate the minimax procedure in Figure 4.1, with explicit values for

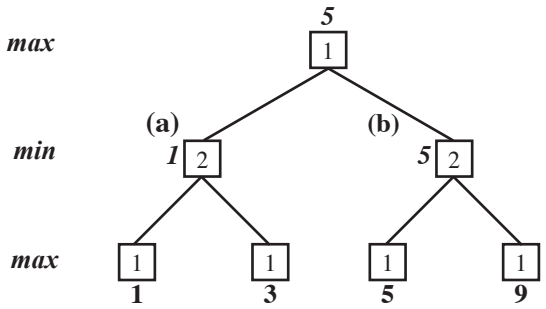

Figure 4.1. A 2-player minimax tree fragment.

each player. Each node is drawn as a box with the number of the player to move inside the box. Outside the box is the minimax value of each node.

In Figure 4.1, Player 1 is a max-player and Player 2 is a min-player. At the node marked (a), Player 2 is to move, so Player 2 will consider the minimax values of his children, 1 and 3, and, as a min-player, choose the value that is minimum, 1. Player 2 is also to move at node (b), and again he will select the minimum value of his children, 5. Similarly, the node at the root is a max node, with child values of 1 and 5. So, Player 1 will choose the maximum child value, 5, to be the minimax value of the game tree.

The static evaluation function used to get the minimax values in the tree is only applied at the leaves of the trees. All other minimax values in the tree, shown in italics, come from backing up minimax values from the leaves of the tree.

In the minimax algorithm, the competing players are commonly referred to min and max. At any node in the search tree, the minimax value of that node is defined by one of three cases, shown in the pseudo-code for the algorithm:

```
minimax(node)
{
       if node is terminal
              return static evaluation function

       if node is min-node
              currentBest = ∞
              FOREACH child of node
                     currentBest = min(currentBest, minimax(child))

       if node is max-node
              currentBest = -∞
              FOREACH child of node
                     currentBest = max(currentBest, minimax(child))

       return currentBest
}
```

4.1.1 Theoretical Properties of Minimax

Theoretically speaking, the minimax algorithm is calculating an equilibrium point [Luce and Raiffa, 1957] in the search tree. If the players in the game are using the same evaluation function and if they are searching the game tree in its entirety, this guarantees several properties. Most importantly minimax guarantees that regardless of the strategy of one's opponent, one will never do worse than the value returned by minimax. This can be equated in some sense with playing optimally, as it is the best strategy possible against an optimal opponent, but it might not take advantage of a suboptimal opponent.

[Schaeffer, et al, 1992] noted that their Checkers program, Chinook, might have a choice between two moves that both will lead to draw, given an optimal opponent. But, one move might lead to an obvious draw, while the other move would lead to a draw most humans would miss. In this case, with a computer playing against a sub-optimal opponent, there is a need to do more than calculate an

equilibrium point. There also needs to be a mechanism for forcing our opponents to play the most difficult line of play possible in order to beat us. As of yet there is no automatic way of doing this.

Another point which detracts from the theoretical guarantees of the equilibrium point is that in most games we aren't able to search a game tree in its entirety. This means that the values being backed up are not actual evaluations of the state of the game, but heuristic approximations, and our opponent is not guaranteed to be using the same evaluation function that we are.

Other objections against minimax as a basic decision can be found in [Russell & Norvig, 1995]. But, regardless of these arguments, the fact remains that minimax has been the decision rule used for most successful expert-level game implementations.

4.2 Paranoid Algorithm

The paranoid decision rule [Sturtevant and Korf, 2000] is the simplest multi-player algorithm, as it side-steps the issue of multiple players by reducing a game from an *n*-player game to a 2-player game. While this is not necessarily accurate, it is simple. Because of this reduction, paranoid is identical to minimax from a theoretical standpoint.

The paranoid algorithm reduces a *n*-player game to a 2-player game by assuming that a coalition of *n*-1 players have formed to play against the remaining player. We demonstrate this in Figure 4.2. In a multi-player game each player's

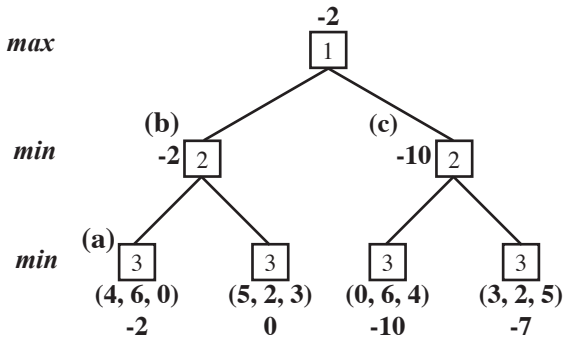

Figure 4.2. A 3-player paranoid tree fragment.

score is represented by an entry in a *n*-tuple, where the *i*th entry is the score for the *i*th player in the game. In this figure, the score for each player is found in the triple next to each node. To convert the game into a two-player zero-sum game, we change the evaluation triple to a single value, the difference between the score of the first player and the remaining players' scores from the. At node (a), for example, Player 1 has a score of 4, while Player 2 has 6 points and Player 3 has 0 points. Thus, the paranoid evaluation is 4-(6+0) = -2.

We calculate the paranoid value of the game tree exactly as we did for minimax. At node (b), Player 2 chooses the best value for the coalition of Players 2 and 3, -2. Similarly at node (c), Player 2 chooses the minimum of his children, -10. At the root of the tree, Player 1 can get a value of -2 from the left child and -10 from the right child, so he will choose to move towards the left branch, and the paranoid value of the game tree will be -2.

4.2.1 Paranoid Deficiencies

As one may suspect, there will be cases where the paranoid algorithm makes poor decisions based on its paranoid view of the world. We demonstrate this with a

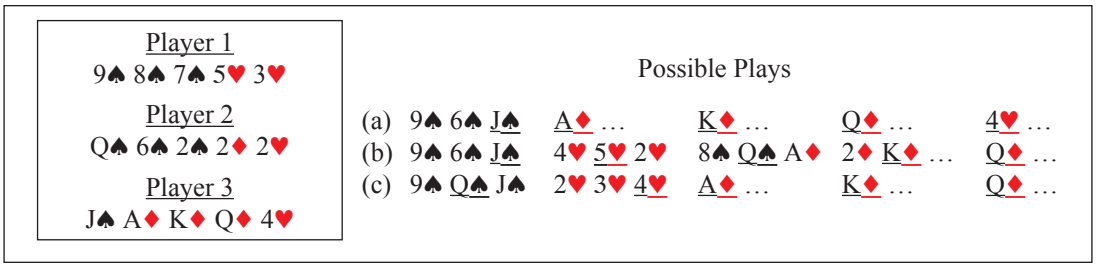

Figure 4.3: Paranoid worst-case analysis 1.

sample hand from the game of Hearts in Figure 4.3. In this example we show three possible ways that the tricks could play out, underlining the cards that will take tricks. We will do our analysis from Player 2's perspective, assuming that Player 1 has already led the 9 of spades. There are three relevant lines of analysis. The first line of play (a) is what we would expect to happen. In this line Player 2 ducks Player 1's lead with the 6, and Player 3 is forced to take the trick with the jack. At this point Players 1 and 2 can duck any card that Player 3 plays, so Player 3 should take the rest of the tricks and get all 17 points remaining in the game, four hearts and the Queen of Spades.

But, the paranoid algorithm puts Players 1 and 3 in a coalition, so the line of play in (a) is the worst possible for the coalition. Instead, the paranoid algorithm would come up with the line of play in (b) or (c), as Player 1 and 3 will have lower combined scores in these lines of play. In the case of (b) Player 2 will duck the spade trick once again. Then, instead of ducking a heart trick, Player 1 will take the trick with his 5. This allows him to lead back spades until Player 2 is forced to take the Queen of Spades. The final score will be 13 points for Player 2 and 4 points for the coalition of Player 1 and 3. In this analysis, Player 1 is taking 3 points with his 5 of

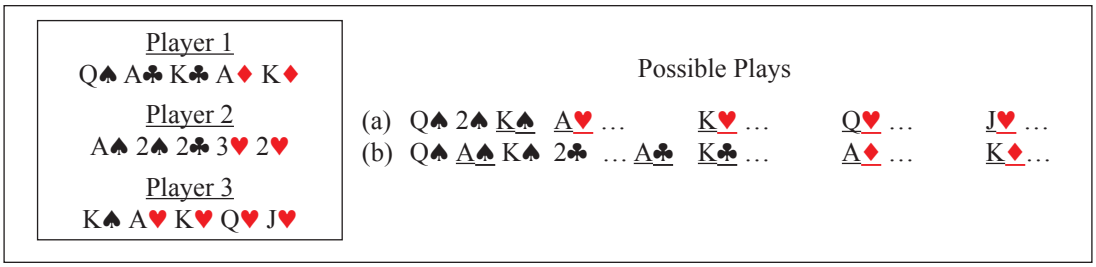

Figure 4.4: Paranoid worst-case analysis 2.

Hearts to save Player 3 from taking the Queen, which he usually won't be willing to do in practice. The line of play in (c) has the same result as line (b), except that Player 2 takes the Queen of Spades immediately instead of waiting to take it later.

When using the paranoid algorithm, Player 2 will not accept line (a) as a possible outcome of the game, and will have to break the tie between lines (b) and (c). Because both line (b) and line (a) start identically, it is possible that if Player 2 chooses to play line (b) that his opponents may end up playing out line (a) instead. (Particularly if they aren't really in a coalition.) Ideally a tie-breaking rule would always be able to tell us to choose line (b) over line (c), but it is possible for there to be more subtle interactions that won't always be easily determined with a tie-breaking rule.

Figure 4.4 is another even more extreme example of analysis by the paranoid algorithm. In this case we analyze the situation from the perspective of Player 1. In this example, the expected line of play is (a), where Player 3 is forced to take the Queen of Spades with the King, and then will take the rest of the tricks for a total of 19 points. But, a paranoid analysis will come up with the line of play in (b), where Player 2 will take the Queen of Spades in order to lead back low clubs for Player 1 to

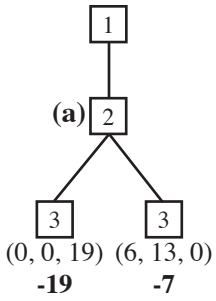

Figure 4.5: The paranoid game tree behind Figure 4.4.

take. In this case Player 1 will take 6 points, Player 2 will take 13 points, and Player 3 will take none.

The real difficulty comes when the paranoid algorithm compares this situation to another situation in which Player 1 is guaranteed to take 5 points no matter the opponent strategy. In such a case most human players would opt to risk taking 6 points than be guaranteed to take 5, but the paranoid algorithm will always choose to take 5 guaranteed points over a risk of taking 6.

A partial game tree behind this decision is in Figure 4.5. Using the paranoid decision rule at node (a) Player 2 (as maximizer) must choose between a score of -19 and -7, and will select the right branch with a score of -7. If Players 2 and 3 are truly in coalition this is the correct move. But, if not, Player 1's paranoid model of Player 2 has Player 2 paying a high penalty (13 points) for staying in the coalition.

Despite these issues, the paranoid algorithm still has some merits. Because it reduces a game to a two-player game, it will inherit all the properties of minimax, particularly the two-player notion of an equilibrium point. This can be used to provide a bound on one's score, which could be very useful at times. For instance, you

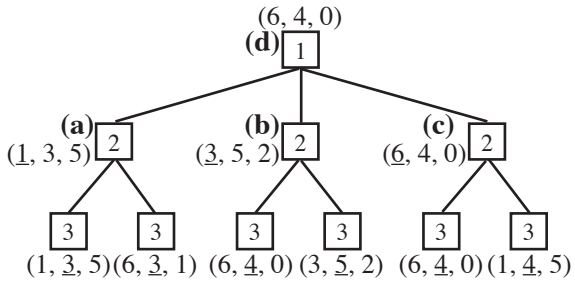

Figure 4.6: A 3-player max^n game tree

may just need to take less than 10 points to win the game, and the paranoid algorithm could provide a line of play with that guarantee. In addition, it has desirable pruning properties that we will consider in Section 5. Finally, in a game in which it is difficult for players to collude, the paranoid algorithm will end up approximating the max^n algorithm, which we consider in the next section. This may seem counter-intuitive, but in a game where there is no advantage for the players to collude, there is also no disadvantage in assuming one's opponents are in a coalition. So if, in such a game, assuming everyone is in a coalition allows us to prune more, we can do so with no loss to the model of how our opponents behave. Of course if our play is truly independent, then we aren't actually playing a competitive game.

4.3 Max^n

The max^n algorithm [Luckhardt and Irani, 1986] is the generalization of minimax to a game with *n* players. For two-player games, max^n reduces to minimax. In a max^n tree with *n* players, the leaves of the tree are *n*-tuples, where the *i*th element in the tuple is the *i*th player's score. At the interior nodes in the game tree, the max^n value of a node where player *i* is to move is the max^n value of the child of that node

for which the *i*th component is maximum. This can be seen in Figure 4.6.

In this tree fragment there are three players. The player to move is labeled inside each node. At node (a), Player 2 is to move. Player 2 can get a score of 3 by moving to the left or right. We break ties to the left, so Player 2 will choose the left branch, and the max^n value of node (a) is (1, 3, 5). At (b), Player 2 selects between 4 and 5, so the max^n value there is (3, 5, 2), and at (c) Player 2 breaks another tie to select (6, 4, 0). Finally, at the root, Player 1 will select the child which has the greatest first component, (c), with (6, 4, 0) as the max^n value of the entire tree. The pseudo-code for max^n is as follows:

```
Maxn(Node, Player)
{
       IF Node is terminal
              RETURN static value
       Best = Maxn(first Child, next Player)
       FOR each remaining Child of Node
              Curr = Maxn(Child, next Player)
              if (Curr[Player] > Best[Player])
                     Best = Curr
       RETURN Best
}
```

4.3.1 Equilibrium Points

The idea of equilibrium points exists in multi-player games as well as in two-player games. [Nash, 1951] introduced and proved the existence of an equilibrium point for any game, while [Jones, 1980] provided a method for calculating equilibrium points in multi-player games, which eventually resulted in the max^n algorithm [Luckhardt and Irani, 1986].

In a *n*-player game, an equilibrium point exists when, given a strategy that leads to a particular max^n value, “no player finds it is to his advantage to change to

a different strategy so long as he believes that the other players will not change." [Luce and Raiffa, 1957]. But, this ends up being much weaker than the same concept in a two-player game.

In a multi-player game there are multiple equilibrium points that may have completely different equilibrium values and strategies. Some discussion of the weakness that results from multiple equilibrium points can be found in [Luce and Raiffa, 1957]. We extend these ideas and make them concrete with a few points on the practical consequences of multiple equilibrium points.

We begin with an example of multiple equilibrium points in Figure 4.6 at node (c). At this node Player 2 can choose either of his children, as both will lead to the same score for Player 2. But, if Player 2 chooses (1, 4, 5) as the max^n value of node (c), Player 1 will choose the result from node (b), (3, 5, 2) to be the max^n value of the tree. Thus, the max^n values (6, 4, 0) and (3, 5, 2) are both the result of valid equilibrium strategies in the tree.

In section 4.1.1 we discussed that, in a two-player game, minimax is the best strategy we can use against an optimal opponent. In addition, no matter what strategy the minimizer uses, minimax will always provide a lower bound on the score of the maximizer. In multi-player game this is not the case. In fact, unless our model of our opponent is perfect, the max^n decision rule can make no guarantees about our score in the game.

Theorem 4.1. In a multi-player game, if a player incorrectly models their opponents' tie-breaking rule, we cannot bound the error between the calculated max^n

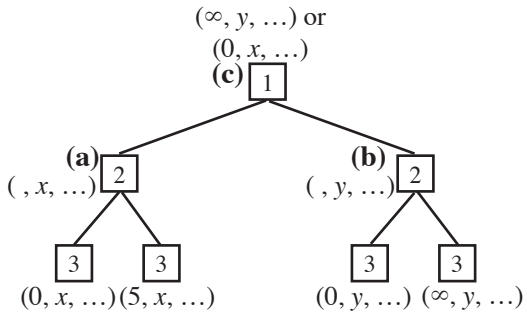

Figure 4.7: Generic tie-breaking in a max^n game tree

value and the actual max^n value of the tree.

Proof: Figure 4.7 contains a generic max^n tree. We have represented Player 2's possible scores by x and y. At node (a), Player 2 can decide whether Player 1 will get 0 points or 5 points on that branch. Similarly, at node (b), Player 2 can decide whether Player 1 will get 0 points or ∞ points. So, by adjusting his tie breaking, Player 2 can give Player 1 a score of 0, 5, or ∞. These values can be chosen arbitrarily, so the theorem holds. □

This result doesn't mean that max^n is a worthless algorithm. A game tree with no ties will have a single equilibrium point and a single max^n value. In addition, each possible max^n value that results from a particular tie-breaking rule will lead to

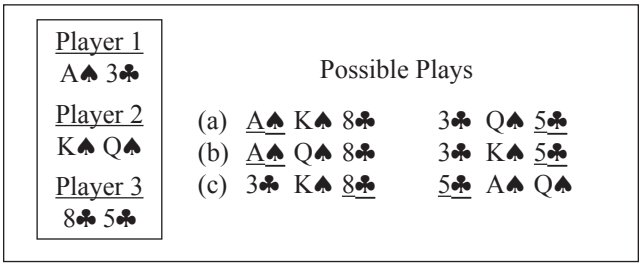

Figure 4.8: Tie breaking situation

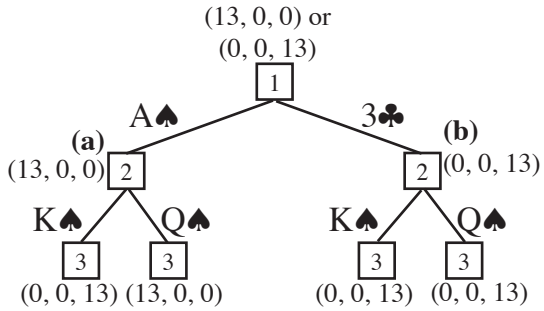

Figure 4.9: Max^n game tree for Figure 4.8.

a reasonable line of play, given that all players use that tie-breaking rule. But, in a real game we rarely know the tie-breaking rule that our opponent is using. In fact, the choice of a tie-breaking rule is part of a strategy for play. In Hearts, for instance, good players will often save the Queen of Spades to play on the player with the best score. Thus, it is reasonable, and perhaps even required, that we take into account our opponents strategy if we wish to write an actual expert-level program.

We illustrate the implications of theorem 4.1 in Figure 4.8. Each player holds 2 cards, as indicated, and three possible outcomes of the play are shown. The winning card of each trick is underlined. If cards are played from left to right in your hand by default, Player 1 can lead the A♠, and Player 2 will not drop the Q♠, as in play (a). However, if Player 2 breaks ties differently, this could be a dangerous move, resulting in Player 1 taking the Q♠, as in (b). But, if Player 1 leads the 3♣, as in (c), Player 3 will be forced to take the Q♠.

We demonstrate the exact same situation in a game tree in Figure 4.9. In this figure Player 2 is indifferent to how he breaks the tie at (a), because either way he will get 0 points. But, if Player 1 assumes he will break it to the left at node, and he

instead breaks it to the right, Player 1 will take 13 points. But, if Player 1 chooses the move towards node (b), he is guaranteed to get 0 points, regardless of Player 2's tie-breaking rule.

One possible tie-breaking rule we have found effective for such situations has been to assume that our opponents are going to try to minimize our score when they break ties. This obviously has a flavor of the paranoid algorithm, and it will cause us to try to avoid situations where one player can arbitrarily change our score.

Explicit tie-breaking rules, however, have detrimental effects on the pruning algorithms considered in Chapter 5. An alternate approach to tie-breaking is to order moves based on how we estimate ties should be broken. In Hearts, for instance, we would always consider playing the Queen of Spades first, if a higher spade was already in the current trick. Similarly, if no higher spades have been played, we should always consider playing the Queen of Spades last. This provides an advantage in that we don't need to explicitly worry about ties when searching a game tree, but we do need to incorporate tie-breaking into our move-ordering function. This will be effective as long as we can accurately estimate how ties should be broken before searching.

4.4 Conclusions

In this chapter we have demonstrated two different decisions rules that can be used for multi-player games, and also demonstrated some of the theoretical and practical properties of these algorithms. In Chapter 5 we will continue by analyzing how we can calculate these decision rules as efficiently as possible.

Chapter 5

Pruning Algorithms

Closely connected with the choice of decision rules for search is the efficiency by which we can calculate a given decision rule. Since the games we are interested in usually have game trees that grow exponentially in depth, any techniques that will allow us to search deeper into the game tree while preserving the result of the decision rule will be very worthwhile. In general, the deeper we can search the more accurate our evaluation is going to be, so a deep search with a simple decision rule usually outperforms a shallow search with a more complicated decision rule. Furthermore, if we can search to the end of a game tree, we can base our decision on exact outcomes rather than heuristic estimates.

5.1 Alpha-Beta Pruning

Alpha-beta pruning was first developed in the context of two-player games. An early implementation is described in [Newell et al., 1958], but comprehensive analysis if its effectiveness was not done until later [Knuth and Moore, 1975]. Alpha-beta pruning is based on a derived window of values between which a player's scores are guaranteed to fall. Because the paranoid algorithm reduces a multi-player

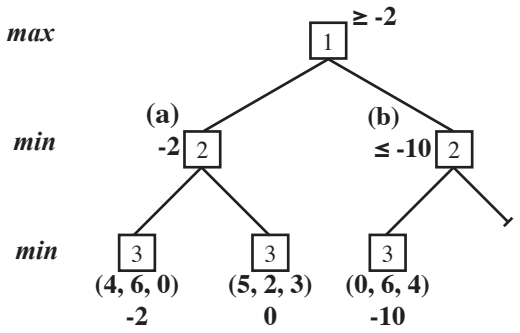

Figure 5.1. A 3-player paranoid tree fragment.

game to a two-player game, alpha-beta pruning works under the paranoid algorithm as it does under any two-player game. The only difference is the asymptotic analysis of the performance of paranoid. So, we present alpha-beta pruning and its analysis here in the context of the paranoid algorithm.

An example of alpha-beta pruning is seen in Figure 5.1. The values in this figure are the same as in Figure 4.2. Player 1 at the root is trying to maximize his score while the other players are trying to minimize it. After searching the subtree at (a), Player 1 knows he can get -2 points by moving to (a). This means that the minimax/paranoid value of the root of the tree will be no lower than -2. So, the window of possible values for the tree has been reduced to the interval [-2, ∞]

After searching (b), Player 2 computes a partial minimax/paranoid value of -10 for Player 1. Since Player 2 is trying to minimize the score, we know that he will never select a move that has a value greater than -10 for node (b). But, at the root, Player 1 will never select a node that has value less than -2. Thus, Player 1 will never choose to move towards node (b), and we can prune away the remaining children of (b), because any value there can never affect the minimax/paranoid value of

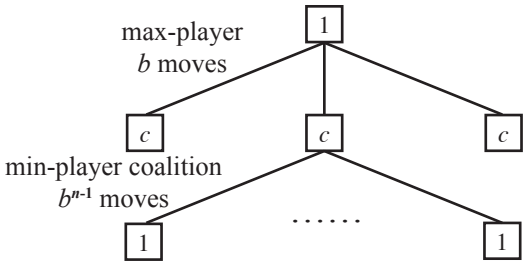

Figure 5.2: Best-case analysis of alpha-beta pruning in paranoid algorithm.

the tree.

Given a game tree that we are searching to depth d, which has a branching factor of b, we would normally have to search b^d nodes to calculate the minimax value of the game tree. However, using the alpha-beta pruning algorithm we can reduce this to $b^{d/2}$ in the best case and $b^{3d/4}$ in the average case. [Pearl, 1984]

5.1.1 Best-Case analysis of Paranoid

The analysis of the best-case performance for the paranoid algorithm [Sturtevant and Korf, 2000] is similar to the analysis of the best-case performance of alpha-beta. However, instead of just having a max and min player in the game, the min player is actual a coalition of multiple players.

To calculate the minimum number of nodes that need to be examined within the game tree, we need to consider a strategy for min and a strategy for max. Min and max will play on the compressed tree in Figure 5.2, where max is to move at the root, with a branching factor of b, and min moves next, with a branching factor of b^{n-1}. Min is the combination of the n-1 players playing against the first player. As will always be the case, the max-player is first to move, with a branching factor of

b. The *n*-1 opponents each have a choice of *b* moves, so as a combined player they will in general have to consider all possible ordering of moves between them, which total b^{n-1}. In some domains such as card games it may be possible for the coalition to reduce this branching factor by reasoning about how their moves combine, but we can't make this assumption in general.

Since max can define its own strategy, the max player must only look at one successor of each max node in the tree. The min strategy is not known, so all possible successors of each min node must be examined. Suppose the compressed tree is of depth D. Max will expand 1 max node and b^{n-1} min nodes at alternate levels, meaning that there are $b^{(n-1)\cdot D/2}$ leaf nodes in the optimal max-player strategy. Similarly, when min searches the tree, the min strategy must look at only one successor of each min node, and all successors of each max node, so min will look at $b^{D/2}$ leaf nodes in the optimal min-player search tree. We assume that each player has an equal number of turns in the game tree. Given two players in the compressed game, D must be even, meaning we don't have to consider the floor or ceiling in the exponent.

The minimum number of leaf nodes examined by both strategies combined will be $b^{(n-1)\cdot D/2} + b^{D/2} - 1$ nodes. We subtract one because there must be one common leave node between the two strategies. Asymptotically this is $O(b^{(n-1)\cdot D/2})$, which is the result for searching a standard minimax tree. But, D is the depth of the tree of Figure 5.2, which has all the players in the paranoid coalition combined together. Instead we would like our results in terms of the real tree that we will search.

For every 2 players in the compressed tree, there are n players in the original tree. So, to convert D to the actual depth of the tree, we multiply by $n/2$. If d is the actual depth of the tree, $d = D \cdot n/2$, and $D = 2 \cdot d/n$. Thus, we can re-write the asymptotic value as $O(b^{(n-1) \cdot d/n})$. For the case of $n = 2$, this is still $b^{d/2}$, the results of the analysis of standard two-player alpha-beta. For $n = 3$, we get $O(b^{2 \cdot d/3})$, and in general as the number of players increases the best-case performance will decrease.

Therefore, one of the best reasons to consider the paranoid algorithm is that it can provide large gains in search depth through pruning, and it is possible that those gains may offset the drawback of an unrealistic decision rule.

5.2 Max^n Immediate Pruning

Immediate pruning is the most basic form of pruning that can occur in a max^n game tree. It occurs when a player gets the maximum possible score (*a win*), and thus does not need to search any remaining children of the node it is currently searching. For multi-player games, this terminology was introduced in [Korf, 1991]. In most games, however, we cannot search deep enough to evaluate nodes as a win or a loss, and some games, such as card games, do not usually evaluate directly to a win or a loss.

5.2.1 Best-Case Analysis of Immediate Pruning in Max^n

To our knowledge, the best-case analysis of immediate pruning has not yet been done, so we present the analysis here. We assume that on every leaf node exactly one player will get a win, and all remaining players will lose. A value of 1 will

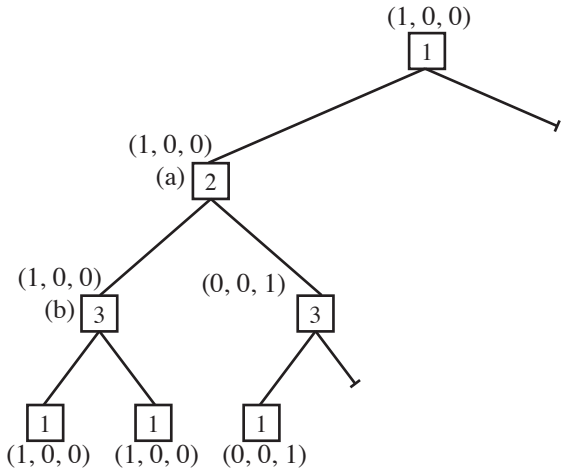

Figure 5.3: Analysis of immediate max^n pruning.

represent a win, while a value of 0 is a loss.

First, we consider possible upper bounds on the best case. If all leaf-nodes are a win for Player 1, the game tree will look exactly like it does in the best case of the paranoid algorithm, with Player 1 always only considering 1 move, and players 2 though n always considering all their moves. This would reduce the effective branching factor from b to $b^{n-1/n}$. However, as we will see in the next section, the best case for shallow pruning reduces the branching factor to $b^{½}$ for large b [Korf, 1991], so immediate pruning should clearly be able to do at least this well.

We can tighten the best-case upper bound using a *greedy* strategy to determine how to choose the best values for pruning in the game tree. That is, we will always choose values that will allow us to prune as high in the game tree as possible. We also assume that ties are broken to the left. Consider the tree in Figure 5.3. The highest prune we can make in the tree is at the root node, and to do so, Player 1 must have a win at that node. In the best case, this value will come from the left-most leaf

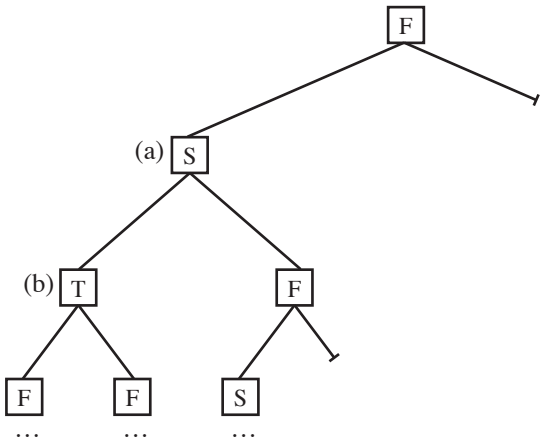

Figure 5.4: Generic analysis of immediate max^n pruning.

in the game tree, and so on the left-most branch Player 1 will always be able to immediately prune, while the remaining players will never be able to prune, because they must return a win for Player 1 as their max^n value.

After pruning the right branch of the root of the tree, the next highest node that might be pruned in the tree is (a), Player 2's node on the second level of the game tree. Every value at this node must be a loss for Player 2, so that the win is returned from the first branch of (a) for Player 1 at the root. So, the most immediate pruning that can occur at this node is for each remaining child to be a win for Player 3, so that Player 3 must only search one branch of each of his children. At node (b) on the third level of the tree, Player 3 is in a similar situation to his parent, except that all his children will be Player 1 nodes, and thus each value there should be a win for Player 1.

We first analyze the 3-player case, and then generalize it to *n* players. We do this by writing a recurrence for the number of nodes at any level in the tree in terms

of first-level (F) nodes, second-level (S) nodes and third-level (T) nodes. We demonstrated these three classes of nodes in Figure 5.4. This figure corresponds to Figure 5.3, except that nodes are labelled by their type instead of the player to play.

First-level nodes are nodes for which the left-branch is a win for the player moving at that node, and all remaining branches can be pruned. An example of a first-level node is the root of the tree in Figure 5.3. Second-level nodes are nodes for which the parent won on their first branch, so the remaining children besides the first child will win on their first branch. Node (a) in Figure 5.3 is a second-level node. Third-level nodes, are nodes for which all our children will win on the first branch. Node (b) in Figure 5.3 is a third-level node.

For each first-level node, we generate a single second-level node. Each second level node generates a single third-level node and (b-1) first level nodes. Finally, each third-level node generates b first-level nodes. So:

$$F(n) = (b - 1)\cdot S(n - 1) + b\cdot T(n - 1)$$

$$S(n) = F(n - 1)$$

$$T(n) = S(n - 1)$$

Which simplifies to:

$$F(n) = (b - 1)\cdot F(n - 2) + b\cdot F(n - 3)$$

To solve this recurrence, we must solve this equation for x:

$$x^3 - (b - 1)\cdot x - b = 0$$

An exact solution can be calculated for any b through use of the standard equation for the roots of a third order polynomial. But, in general, as b grows large,

the solution will grow on the order of $O(b^{1/2})$. One way to verify this is to notice that the last term of b is guaranteed to be asymptotically smaller than the $(b - 1)\cdot x$ term. Because we are interested in the asymptotic growth, we can ignore this term and solve:

$$x^3 - (b - 1)\cdot x = 0$$

$$x^2 = (b - 1)$$

$$x \approx b^{1/2}$$

For an n-player game there will be n different types of nodes to consider. In the same way as the three-player analysis, a first-level node will generate a single second-level node. Each node type between 2..n-1 will generate a single node of the next level along with $(b - 1)$ first-level nodes. Finally, the n^{th} level node will generate b first-level nodes. So, the general recurrence will be solved by the equation:

$$x^n - (b - 1)\cdot x^{n-2} - (b - 1)\cdot x^{n-3} \ldots - (b - 1)\cdot x - b = 0$$

Again, we can see that the first two terms are the most significant asymptotically, and solving just for these terms we see that as b grows large the solution for x will grow on the order of $O(b^{1/2})$. This means that in the best case there are significant gains possible from immediate pruning in multi-player games; on the same order as the gains from alpha-beta pruning for two-player games. Additionally, we can show that this is the optimal way to choose values to prune the game tree.

Theorem 5.1: No assignment of values to a max^n game tree can produce more immediate pruning than the *greedy* assignment strategy.

Proof: Suppose there is a node n to which a greedy strategy assigns a winning max^n

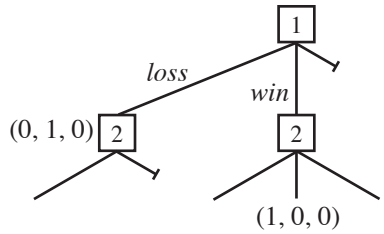

Figure 5.5: Sub-optimal ordering for immediate pruning a max^n tree.

value, but some better algorithm does not. Let n also be the first (highest) such node that occurs in the game tree. First, the other strategy cannot gain more pruning above this node in the search because, by definition, the greedy strategy has already pruned any nodes that could possible be pruned. Thus, for this new strategy to be better than greedy it must somehow be able to prune more nodes below n by not pruning n immediately than it could if it just pruned n.

There are two ways this might happen. First, by choosing a value for the first child for n that will not result in an immediate prune, but choosing same later value that will result in an immediate prune. We demonstrate this in Figure 5.5. It should be obvious from the figure that if we are going to immediate prune a node we should always do so on the first branch, since we will never have to search the *loss* branch if we do. Not doing so is just guaranteed to make us expand more nodes.

The second thing an algorithm can do is not prune n at all. In this case it must expand all b children of n. For each of these children, any algorithm must expand at least the first child of each of these nodes as well. In this scenario there are guaranteed to be at least b children and therefore b grandchildren of n, since every node has at least one child. This means there will be at least $2 \cdot b$ nodes in the two levels

following *n*. However, if we prune *n* immediately, there will only be one child of *n* and *b* grandchildren for (*b* + 1) nodes in the two levels following *n*. In addition, the pruning on each of these grandchildren is independent, so there is no way to use analysis from one grandchild of *n* to help prune another. This same argument follows for any number of levels following *n*. Following this analysis, not pruning *n* is guaranteed to create a larger subtree. Thus, there is no strategy for choosing values to prune a max^n tree better than the greedy strategy, and our best-case analysis is the optimal case for immediate pruning. □

In the worst-case, even if every terminal nodes evaluates to a win or loss it is not difficult to build a tree for which immediate pruning will never be able to prune. The general strategy for building such a tree is that each player gets a loss on their first (*b*-1) children, and a win on their last child. This causes the final return value for each player to be a win for that player and a loss for the parent of that node so that pruning will never occur.

5.3 Shallow Max^n Pruning

[Luckhardt and Irani, 1986] originally noted that a single player makes his decision based only upon his own component of the max^n value their children, disregarding the other components. So, pruning was originally conceived in the context of delaying the evaluation of some components in an *n*-tuple. While this is a reasonable idea in theory, in practice it is usually of little value. This is because it usually takes little or no additional work to compute all components of the max^n value once

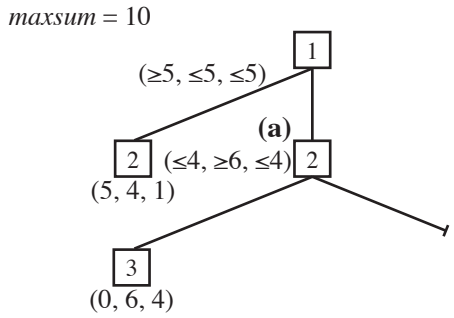

Figure 5.6: Shallow pruning in a 3-player max^n tree.

one component has been computed. This is particularly true if one player's score in a max^n *n*-tuple is based the other players' scores, meaning that it may not be possible to do the calculations separately, and that calculating the max^n value for one player explicitly gives you the max^n value for all players. Finally, delaying evaluation of all components of the max^n value means that we must know the state to which the max^n value corresponds, which, in most cases, is more complicated than just calculating the actual max^n value.

[Korf, 1991] first proposed a shallow pruning algorithm for pruning away entire nodes in a max^n tree, given simple restrictions on the actual max^n values involved. Shallow pruning refers to cases where a bound on a node is used to prune branches from the child of that node.

The minimum requirements for pruning a max^n tree with shallow pruning are a lower bound on each players' score and an upper bound, *maxsum*, on the sum of all players' scores. We demonstrate this in Figure 5.6. In this figure, all scores have a lower bound of 0, and *maxsum* is 10.

Player 1 searches the left branch of the root in a depth-first manner, getting

the max^n value (5, 4, 1). Thus, we know that Player 1 will get at least 5 points at the root. Since there are only 10 points available, we know the other players will get no more than 5 points at this node. At node (a), Player 2 searches the left branch to get a score of 6. Because *maxsum* is 10 we know that Player 1's score will never exceed 4 at node (a), so Player 1 will never choose to move towards (a) at the root, and the remaining children of (a) can be pruned.

In the best case, shallow pruning will reduce the asymptotic branching factor from b to $\frac{1}{2}\cdot(1+\sqrt{4b-3})$, which converges to $b^{d/2}$ as b gets large, while in the average case, no asymptotic gain can be expected from shallow pruning a game with more than two players. [Korf, 1991] The basic problem with shallow pruning is that it works by comparing the scores of only 2 out of n players in the game, and it is unlikely that 2 players will have the sum of their scores exceed *maxsum*. This contrasts with the large average-case gains available from alpha-beta pruning in 2-player minimax.

Given this, it is worth considering whether it is in practice possible to achieve the best case for shallow pruning. In section 5.3 we will consider this problem, followed in section 5.4 by a discussion of what the limits are to shallow max^n pruning in practice.

5.4 Achieving Shallow Pruning Best Case

It is not difficult to determine that there is a game tree for which the best-case for shallow pruning occurs in practice. Furthermore, the exercise provides useful

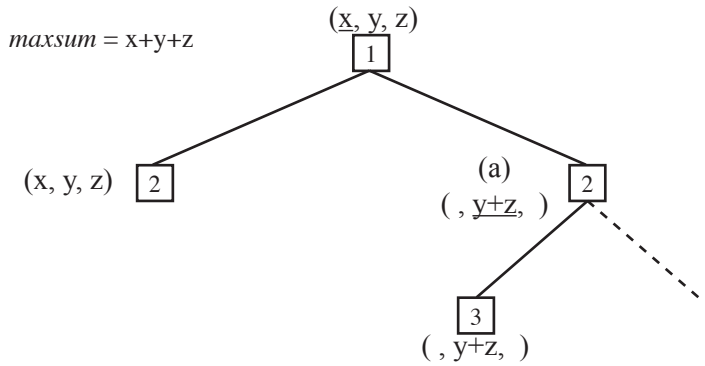

Figure 5.7: Basic structure for shallow max^n pruning.

insight into the nature of max^n game trees. In fact, it will lead us to show that for many real games it is impossible for the best case to occur in practice. We assume, without loss of generality, that the game tree must be constant-sum. If this isn't the case the result of the analysis still holds, it is just slightly more complicated.

To begin our analysis we consider the tree in Figure 5.7. In this tree the players get arbitrary scores on the left branch. Then, given these scores, Player 2 must get a score of $y + z$ at (a) so that Player 1's score of x at the root and Player 2's score at (a) sum to *maxsum*. Max^n pruning can only occur in a sub-tree that has this basic structure, meaning that the relevant portions for pruning will never involve a larger or smaller subtree. This means that we can and must duplicate this tree structure to build larger game trees that are pruned optimally. In a larger tree we always choose the max^n values for the tree so that we can prune. We do this in Figure 5.8.

In this figure we have not drawn a complete game tree. Instead we have drawn multiple copies of the subtree from Figure 5.7 as it would occur in a larger optimal tree. This tree provides the minimum framework necessary for understand-

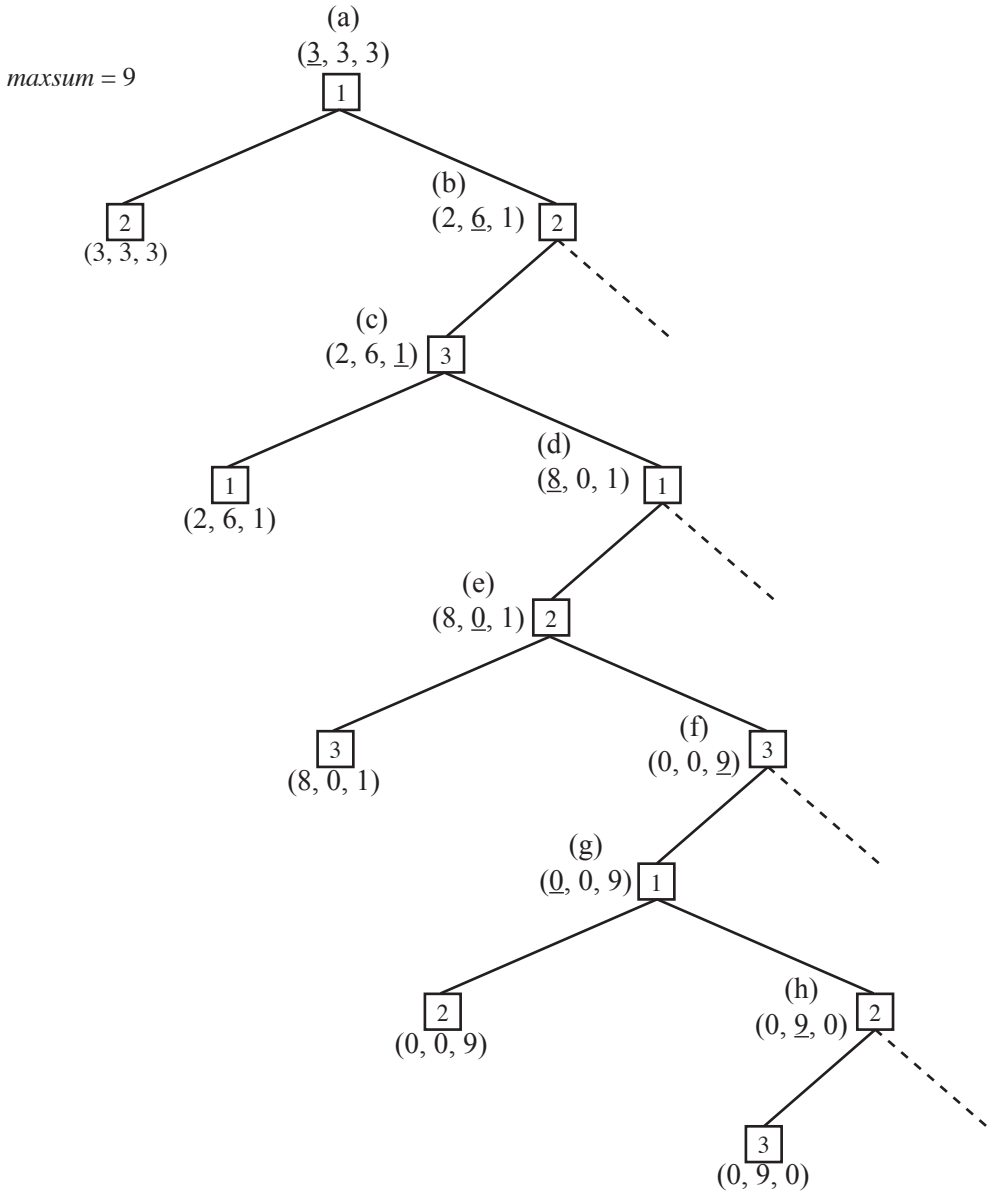

Figure 5.8: Best case tree for shallow max^n pruning.

ing what will happen to a max^n game tree in general if we are to choose the max^n values so that we get the best case for shallow pruning.

Beginning at the root of Figure 5.8, we have a duplicate of Figure 5.7 at node (a) in the tree. We can choose any value for the max^n value of the game tree, but we start with values that are evenly split among the players in the game to help illustrate our point. Because Player 1 can get 3 points after searching the left branch of (a), we know that Player 2 must get at least 6 points at (b) to be able to shallow prune the children of (b). Given that, the max^n value of (c) must be chosen such that Player 2 gets at least 6 points. We choose to give Player 1 two points and Player 3 one point, although the ultimate results in the tree will be the same regardless of how we split the values. Node (c) also happens to be the root of a new subtree which resembles Figure 5.7.

Since Player 3 has 1 point at (c), Player 1 must have at least 8 points at (d) to be able to shallow prune at (d), and Player 1's max^n value at (e) must be at least 8. In this case we chose to give Player 2 zero points and Player 3 one point. Because Player 2 has 0 points at node (e), Player 3 must have 9 points (*maxsum*) at node (f) to be able to prune there. Finally, because Player 1 has 0 points at node (g), Player 2 must get 9 points at node (h).

This succession of moves illustrates that in a tree for which shallow pruning always occurs when possible, the scores for the players in the tree will always converge to 0 and *maxsum*. Although we chose the max^n values in this example to make the values converge on *maxsum* as quickly as possible, it will always occur,

regardless of how we choose our max^n values. If, for instance, we change Player 2's score to 1 at node (e), then instead having the values at (f) converge to maxsum, they will converge on the left branch of (e).

There are two consequences to this result. First, the best-case analysis of shallow pruning assumes that immediate pruning never occurs. However, given a discrete evaluation function we have shown that if we build a large enough best-case tree for shallow pruning, immediate pruning will *always* occur. Since we are interested in using computers to generate such trees, this will generally be the case. This means that the previous analysis of the best case of shallow pruning is only an upper bound on the best case, as it does not consider the pruning that must occur from immediate pruning in the tree. While both shallow pruning and immediate pruning have the same asymptotic growth, immediate pruning will produce smaller trees in practice.

Secondly, this guarantees that in many games the best-case can never occur in practice. This is easy to see for trump-based games such as Spades, based on Figure 5.8, assuming we can search the entire game tree. At node (g), Player 3 must have taken all the points in the game, but at node (h), Player 2 must have taken all the points in the game. But, in most trump games this is impossible, assuming we are searching a reasonably large portion of the game tree. This is because the player with the highest card in the trump suit *must* take at least one trick, as there is no higher card in the deck. So the highest card in the trump suit will always win the trick on which it is played. Thus, it is impossible for the player holding this card to

get 0 points, which conflicts with the situation either at node (g) or (h).

In a sense these results are not surprising. The average-case model in [Korf, 1991] predicts that the max^n values in the tree will average out as the search gets closer to the root of the tree, and thus no pruning will occur at the top of the tree. From this analysis we see that the only way this can be prevented is if most of the other values in the tree converge on 0 or *maxsum*.

In the next section we continue our discussion of shallow pruning by developing general bounds that describe additional constraints on whether shallow pruning will occur in a game tree in practice.

5.5 Shallow Pruning Limits

While the necessary requirements for shallow pruning are a lower bound on each players score and an upper bound on the sum of all players' scores, these bounds alone are not sufficient to allow pruning in any game tree. To maximize the potential pruning it must be the case that the maximum possible player score is equal to the maximum sum of scores [Sturtevant and Korf, 2000]. This is the case in a game like Spades, but it isn't for Hearts. Because this isn't necessarily intuitively obvious, we will spend some time discussing the surrounding issues. In our discussion we assume that the static evaluation function for these games is based on the number of points taken in the game. While it is possible to base the evaluation function on other game attributes, we hope that this discussion will show that the issue is larger just how we tune our evaluation function.

5.5.1 Minimization versus Maximization

Throughout this thesis we deal with games that are usually described in terms of either maximization or minimization. Since minimization and maximization are symmetric, we briefly present here how the bounds used by pruning algorithms are transformed when we switch from one type of game to the other type.

There are four values we can use to describe the bounds on players' scores in a game. *Minp* and *maxp* are a player's respective minimum and maximum possible score at any given node. *Minsum* and *maxsum* are the respective minimum and maximum possible sum of all players scores. In Hearts, *minp* is 0 and *maxp* = *maxsum* = *minsum* = 26. In Spades, *minp* is also 0 and in a complete game tree *maxp* = *maxsum* = *minsum* = 17. To prune in any game, shallow pruning requires that *minp* and *maxsum* are bounded. We are interested in how these bounds change when the goal of a game is changed from minimization to maximization. The transformation does not change the properties of the game, it simply allows us to talk about games in their maximization forms without loss of generality.

The one-to-one mapping between the minimization and maximization ver-

minimization variable	s_1	s_2	s_3	$maxp_{min}$	$minp_{min}$	$maxsum_{min}$ & $minsum_{min}$
Hearts example minimization value	3	10	13	26	0	26
transformation	$-s_i + maxp_{min}$			$-maxp_{min} + \text{maxp}_{\text{min}}$	$-minp_{min} + maxp_{min}$	$-maxsum_{min} + n \cdot maxp_{min}$
Hearts example maximization value	23	16	13	0	26	52
maximization variable	s^1	s^2	s^3	$minp_{max}$	$maxp_{max}$	$maxsum_{max}$ & $minsum_{max}$

Table 5.9: The transformation between a maximization and minimization problem, and examples for a 3-player Hearts game.

sions of a game is shown in Table 5.9. The first row in the table contains the variable names for a minimization problem, followed by sample values for a Hearts game, where *n*, the number of players, is 3. The transformation applied to the values are in the third row: the negation of the original value plus $maxp_{min}$. This re-normalizes the scores so that *minp* is always 0. Since Hearts and Spades are constant-sum games, *maxsum* is always the same as *minsum*. The final rows contain the new score after transformation and the new variable names. The process can be reversed to turn a maximization game into a minimization game. In general this process corresponds exactly with changing the goal in Hearts from minimizing your own points to maximizing your opponents points.

Given the symmetry of minimization and maximization, there is also a duality in pruning algorithms. That is, for any pruning algorithm that works on a maximization tree, we can write the dual of that algorithm that works the same under the equivalent minimization tree. However, just changing the goal of a game from minimization to maximization does not create an isomorphic game that will have the same properties under minimization or maximization. The other parameter, *maxsum*, must also be calculated. Given these observations, we have not explicitly shown dual algorithms for pruning. Unless otherwise stated, all trees and algorithms presented here will be for maximization trees.

5.5.2 General Bounds for Shallow Max^n Pruning

Figure 5.8 shows a generic max^n tree. In this figure we have only included the values needed for shallow pruning. Other values are marked by a '•'. When Player 1

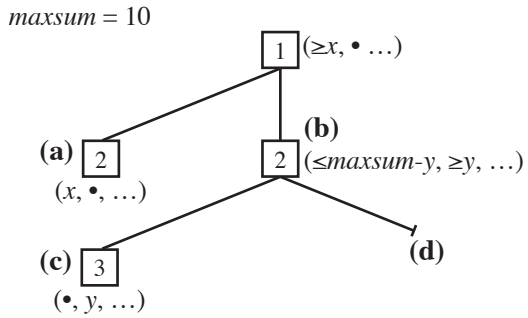

Figure 5.10: Shallow pruning in a 3-player max^n tree.

gets a score of *x* at node (a), the lower bound on Player 1's score at the root is then *x*. Assume Player 2 gets a score of *y* at node (c). Player 2 will then have a lower bound of *y* at node (b). Because of the upper bound of *maxsum* on the sum of scores, Player 1 is guaranteed less than or equal to *maxsum* - *y* at node (b). Thus, no matter what value is at (d), if *maxsum* - $y \leq$ x, Player 1 will not choose to move towards node (b) because he can always do no worse by moving to node (a), and we can prune the remaining children of node (b).

In the 3-player maximization version of Hearts in Table 5.9, maxsum is 52, and *x* and *y* will range between 0 and 26, meaning that we can only prune when 52 - $y \leq x$, which is only possible if $x = y = 26$. But, in this case we don't need shallow pruning to prune, because immediate pruning already tells us we can prune when a player gets *maxp*, which is 26 in this case. In Spades, *maxsum* is 16, and *x* and *y* will range from 0 to 16, meaning that we can prune when 16 - $y \leq x$.

Given these examples, we extract general conditions for pruning in multi-player maximization games. We will use the following variables: *n* is the number of players in the game, *maxsum* is the upper bound on the sum of players scores, and

maxp is the upper bound on any given players score. We assume a lower bound of zero on each score without loss of generality. So, by definition, $maxp \leq maxsum \leq n \cdot maxp$.

Theorem 5.2: To shallow prune in a max^n tree, $maxsum < 2 \cdot maxp$.

Proof: We will use the generic tree of Figure 5.10. To prune:

$$x \geq maxsum - y$$

By definition:

$$2 \cdot maxp \geq x + y$$

So,

$$2 \cdot maxp \geq x + y \geq maxsum$$

$$2 \cdot maxp \geq maxsum$$

However, if $maxsum = 2 \cdot maxp$, we can only prune when both x and y equal *maxp*. But, if $y = maxp$, we can also immediate prune. Because of this, we tighten the bound to exclude this case, and the theorem holds. □

We can now verify what we suggested before. In the maximization version of 3-player Hearts, $maxsum = 52$, and $maxp = 26$. Since the strict inequality of Theorem 5.1, $52 < 2 \cdot 26$, does not hold, we can only immediate prune in Hearts. In Spades, the inequality $17 < 2 \cdot 17$ does hold, so we will be able to shallow prune a Spades max^n tree. In fact, if $maxsum = maxp$, there will always be a legal max^n value for our child that will enable us to shallow prune the children of that node, so this is the ideal value we'd like for a game.

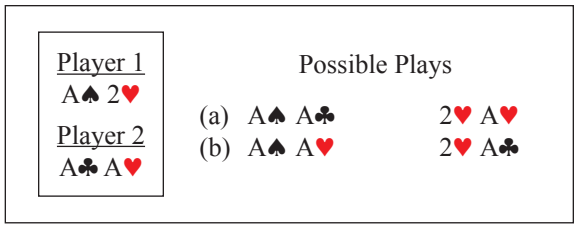

Figure 5.11: Different strategies for minimizing and maximizing.

5.5.3 Intuitive Approach

Speaking in terms of the games as they are normally played, it may seem odd that we can't prune in Hearts and we can prune in Spades, when it seems that the biggest difference in the games is that it one you try to minimize your score, and in the other you try to maximize it. While the preceding theorem explains the difference mathematically, there is another explanation that may be more intuitive.

To begin this discussion, let us consider the game of Hearts from two points of view. First, we consider the game as it is normally played, excluding the rule for shooting the moon. In this game one wishes to minimize their points, so we will call it $hearts_{min}$. Then, we will consider the game with the exact same rules, except that the goal is to maximize your points. We will call this game $hearts^{max}$. In $hearts_{min}$ we will not be able to use shallow pruning while in $hearts^{max}$ we will. The question is, if the only difference between $hearts_{min}$ and $hearts^{max}$ is the question of maximization or minimization, why can we use shallow pruning for one game and not the other?

The reason is that these are completely different games and are not isomorphic to each other. One way to see this is by considering the strategy you would use to play $hearts^{max}$ and the strategy you would use to play $hearts_{min}$. We demonstrate

in Figure 5.11. Given that Player 1 leads the Ace of Spades, in $hearts^{max}$ the correct strategy is (a), where Player 2 can take 2 points by saving the Ace of Hearts to take the second trick. But, in $hearts_{min}$ the best strategy is (b), where Player 2 will take no points. If two games are isomorphic, the same strategy for optimal play must be present across the isomorphism. Additionally, in a card game, an isomorphic mapping must somehow preserve the notion of leading and following within suits. In line (a) Player 1 follows suit on the second trick, while he doesn't in line (b). Thus, these lines of play cannot be preserved across an isomorphic mapping of games, so $hearts^{max}$ and $hearts_{min}$ must be fundamentally different games. Additionally, in $hearts^{max}$ *maxsum* is 26 and *maxp* is 26, so we will be able to shallow prune, while we have already shown that you can't shallow prune in $hearts_{min}$.

Given that they are different games, let us consider what one isomorphic dual of each game actually is. In $hearts^{max}$ you are trying to maximize your score, while the minimization dual is the game where you want to minimize the sum of your opponents scores. But, this is still exactly the same game. Similarly in $hearts_{min}$ you want to minimize your score, and the dual is the game where you want to maximize your opponents score.

So, while the most simple explanation of the difference between $hearts^{max}$ and $hearts_{min}$ is that one is a maximization game and the other is a minimization game, this is a deceptive description, as it leads one to think that the games are isomorphic to each other, when in fact they are two completely different games.

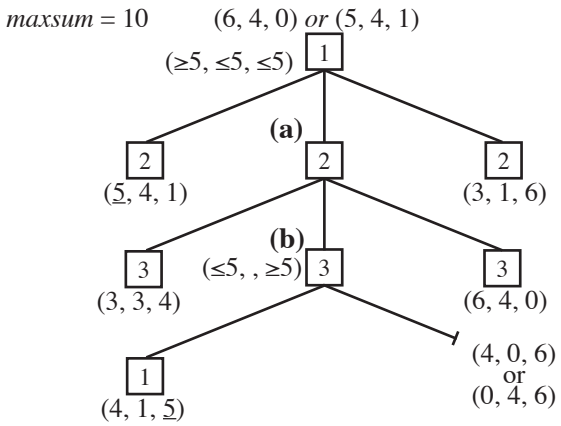

Figure 5.12: The failure of deep pruning.

5.6 Deep (Pair-wise) Pruning

Deep pruning refers to when the bound at a node is used to prune a grandchild or lower descendant of that node. There is more than one way one might try to do this. In this section we will discuss the pruning methods that have been devised where we compare the max^n value of two non-consecutive players within the game tree. In later sections we will discuss other methods where we consider the bounds for more than two players. To help distinguish these methods, we refer to the methods discussed here as pair-wise pruning.

[Korf, 1991] shows that, in the general case, deep pruning can incorrectly affect the max^n value of the game tree. We demonstrate this in Figure 5.12. After searching the first branch at the root, Player 1 is guaranteed at least a score of 5 at the root, and the other players are guaranteed a score no greater than 5. Then, after searching the left branch of node (b), Player 3 is guaranteed 5 points, and Player 1 is guaranteed no more than 5 points at node (b). So, we can conclude that the max^n value of node (b) will never be the max^n value of the game tree, because Player 1 is

already guaranteed a score of 5 at the root, and he can do no better than that at node (b). It is still possible, however, that the max^n value at (b) can affect the final max^n value of the tree.

For instance, if the actual max^n value of (b) is (4, 0, 6), Player 2 will prefer the move (6, 4, 0) at node (a), and this will be the max^n value of the game tree. But, if the max^n value of (b) is (0, 4, 6), Player 2 will prefer this value, and so Player 1 will choose (5, 4, 1) to be the max^n value of the game tree. Thus, in general deep pruning is not valid in a multi-player game.

5.7 Optimality of Max^n

The invalidity of deep (pair-wise) pruning raises the question of what the best possible max^n pruning algorithm is. [Korf, 1991] showed that given no additional constraints, "Every directional algorithm that computes the max^n value of a game tree with more than two players must evaluate every terminal node evaluated by shallow pruning under the same ordering." A directional algorithm [Pearl, 1984] is defined as one that examines the successors of any node in a fixed order without returning to re-search any branch of the tree. We assume for now that we do not know if we are on the last branch of a node, although we won't always make this assumption for a directional algorithm.

While max^n may be optimal under these conditions, we will now cover additional pruning techniques that we have developed which depend on additional constraints on the game tree or non-directional search to prune.

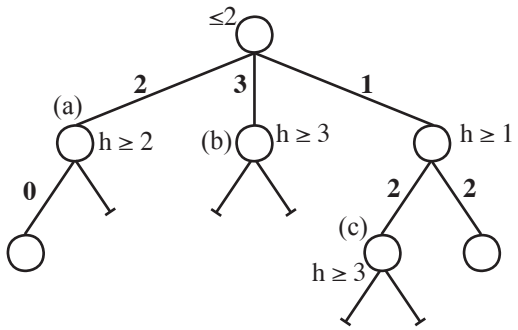

Figure 5.13: A single-agent depth-first branch-and-bound search tree.

5.8 Depth-First Branch-and-Bound Pruning

Branch-and-bound pruning is a common technique from single-agent and two-player [Prieditis and Fletcher, 1998] search which can be used to prune max^n game trees. It requires a monotonic heuristic, but many card games have natural monotonic heuristics. In Hearts and Spades for example, once you have taken a trick or a point you cannot lose it. Thus, an evaluation can be applied within the tree to give a bound on the points or tricks to be taken by a player in the game. We use the notation $h(i) \geq j$ to indicate that the monotonic heuristic is giving a lower bound score of j for player i, and $h(i) \leq j$ to indicate that the monotonic heuristic is giving an upper bound of j on player i's score. Suppose, for a Spades game, Players 1, 2 and 3 have taken 3, 2, and 6 points respectively. Then, $h(1) \geq 3$ because Player 1 has taken 3 points. Also, $h(1) \leq 9$ because maxsum (17) minus the other players' scores (8) is 9.

5.8.1 Single Agent Branch-and-Bound

The branch-and-bound algorithm is most commonly used in a depth-first search to prune single-agent minimization search trees, such as the trees that arise

in the Travelling Salesman Problem. In Figure 5.13, for example, we are trying to find the shortest path to a leaf from the root, where edges have non-negative costs as labelled. Since all paths have positive length, the cost along a path cannot decrease, giving a lower bound on the cost to a leaf along that path. Each edge is labelled with the cost of that edge. The heuristic limits on each node are the sum of the edge costs to that node. If unexplored paths through a node are guaranteed to be greater than the best path found so far, we can prune the children of that node in the tree.

In order to draw parallels between alpha-beta pruning, we will describe the pruning that occurs in the same terms that we use to describe alpha-beta pruning: immediate, shallow and deep pruning. In a two-player game, immediate pruning occurs when we get the best score possible, a win. In the presence of a heuristic, the best score possible is the best that we can get given the heuristic. In Figure 5.13, the heuristic at node (a) says the best score we can get is 2. Since we have a path to a leaf node of total cost 2 through the first child, we can prune the remaining children, as we have found the best possible path.

After finding the path with cost 2, we use that cost as a bound while searching subsequent children. At node (b), our monotonic heuristic tells us that all paths through (b) have cost higher than the bound of 2, so all children of (b) are pruned. This is like shallow pruning, since the bound comes from the parent of (b). Finally, at node (c) we can prune based on the bound of 2 from the best path so far in the tree and the monotonic heuristic cost at (c), which is like deep pruning.

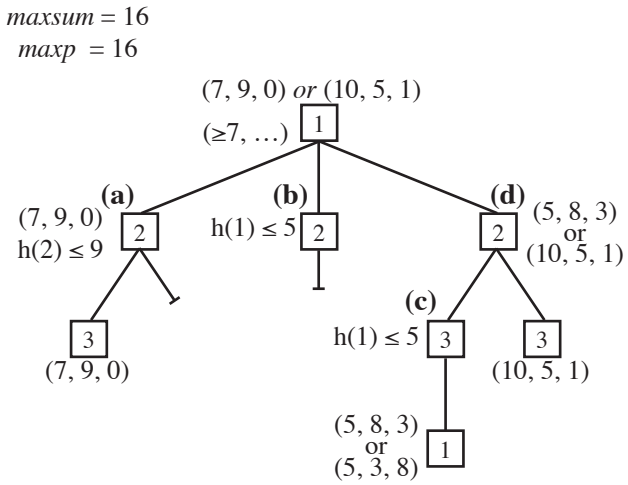

Figure 5.14: Branch-and-bound pruning in a max^n tree.

5.8.2 Multi-Player Branch-and-Bound

Branch-and-bound pruning can be used to prune a max^n tree as well, but under max^n it is limited by the same factors as alpha-beta pruning, namely we cannot use the bound at a node to prune its great-grandchild. As with deep alpha-beta pruning, while the max^n value of the pruned nodes will never be the max^n value of the tree, they still have the potential to affect it. We will demonstrate this here, but because the proof is identical to the proof of why deep alpha-beta pruning does not work [Korf, 1991], we omit the proof.

In Figure 5.14 we show a portion of a max^n tree and demonstrate how branch-and-bound can prune parts of the tree. Immediate pruning occurs at node (a). At the left child of (a), Player 2 can get a score of 9. Given the monotonic heuristic value of node 2, h(2) ≤ 9, we know Player 2 cannot get a better score from another child, and the remaining children are pruned.

Shallow pruning occurs at node (b) when the bound from the parent combines

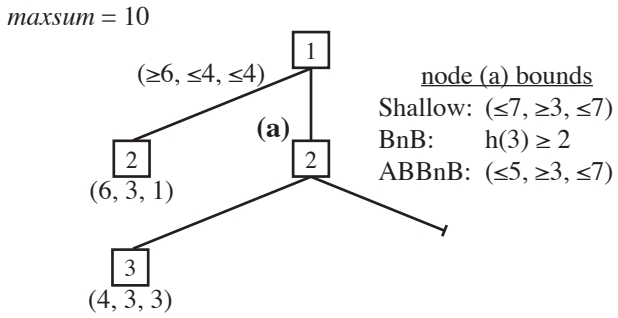

Figure 5.15: Alpha-beta Branch-and-Bound pruning in a 3-player max^n tree.

with the monotonic heuristic to prune the children of (b). Player 1 is guaranteed 7 or more at the root. So, when Player 1's monotonic heuristic at (b) guarantees a score of 5 or less, we prune all the children of (b), since Player 1 can always do better by moving to node (a).

Finally, deep branch-and-bound pruning, like deep alpha- beta pruning, can incorrectly affect the calculation of the max^n value of the game tree. The partial max^n value at the root of the tree in Figure 5.14 guarantees Player 1 a score of 7 or better. At node (c), Player 1 is guaranteed less than or equal to 5 points by the monotonic heuristic. Thus, we might be tempted to prune the children of (c), since Player 1 can do better by moving to node (a). But, this reasoning does not take into account the actions of Player 2.

Depending on which value we place at the child of (c), (5, 8, 3) or (5, 3, 8), Player 2 will either select (5, 8, 3) from node (c) or (10, 5, 1) from node (d)'s right branch to back up as the max^n value of node (d). Player 1 would then choose the root max^n value to be either (7, 9, 0) or (10, 5, 1). So, while the bounds on node (c) will keep it from being the max^n value of the tree, it has the potential to affect the max^n

value of the tree.

5.9 Alpha-Beta Branch-and-Bound Pruning

Now that we have two relatively independent techniques for pruning a multi-player game tree, we show how these techniques can be combined. Shallow pruning makes comparisons between two players' backed up scores to prune. Branch-and-bound pruning compares a monotonic heuristic to a player's score to prune. Alpha-beta branch-and-bound pruning uses both the comparison between backed up scores and monotonic heuristic limits on scores to prune even more effectively.

Looking at Figure 5.15, we see an example where shallow pruning applies. We have bounds on the root value of the tree from its left branch. After searching the first child of node (a) we get bounds on the max^n value of (a). We place an upper bound of 7 on Player 1's score, because Player 2 is guaranteed at least 3 points, and 10 (maxsum) - 3 = 7. This value does not conflict with the partial max^n bound on the root, so we cannot prune. We also have a bound from our monotonic heuristic, but because it is not Player 3's turn and because the lower bound on Player 3's score, 2, does not conflict with the upper bound, 7, we cannot use that by itself to prune either. But, if we combine this information, we can tighten our bounds. We know from backed up values that Player 2 will get at least 3 points and from our heuristic that Player 3 will get at least 2 points at (a). So, the real upper bound on Player 1's score at (a) is *maxsum* - score(2) - h(3) = 10 - 3 - 2 = 5. Since Player 1 can get 6 points on the left branch at the root of the tree, we then know he will never get a better score from (a), and we can prune the remaining children of (a).

So, in a *n*-player game where we normally only compare the scores of two players, we can now instead compare bounds for all n players by using the monotonic heuristic value for *n*-2 players and the results of the search for the other two players. That is, if we have a lower bound on Player *i*'s score from our parent, and Player *j* is to play at the current node, the upper bound on Player i's score at the next node is maxsum - score(j) - $\sum h(x)$ {for $x \neq i$ or j}. The technique of combining the monotonic heuristic values of some players with the backed-up evaluation of other players only works in a multi-player game, because in a two-player game there are no additional players for which to consider, reducing it to just plain alpha-beta. Following is the pseudo-code for alpha-beta branch-and-bound pruning.

```
ABBnB(Node, Player, parentScore)
{
       IF Node is terminal RETURN static value

       /*  shallow branch-and-bound pruning  */
       IF (h_up(Prev Player) ≤ parentScore)
              RETURN static value
       Best=ABBnB(first Child, next Player, 0)

       /* Calculate our opponents guaranteed points  */
       Heuristic = ∑h_low(n) [n≠Player or prev. Player]

       FOR each remaining Child
              IF (Best[Player]+parentScore+Heuristic ≥ maxsum) OR
                 (Best[Player] = h_up(Player))
                     RETURN Best
              Current = ABBnB(next Child, next Player, Best[Player])
              IF (Current[Player] > Best[Player])
                     Best = Current
       RETURN Best
}
```

5.10 The Constant-Sum Property in Multi-Player Games

In two-player games, we always assume that a game is strictly competitive and zero-sum. This means that a move that is good for our opponent will be equally

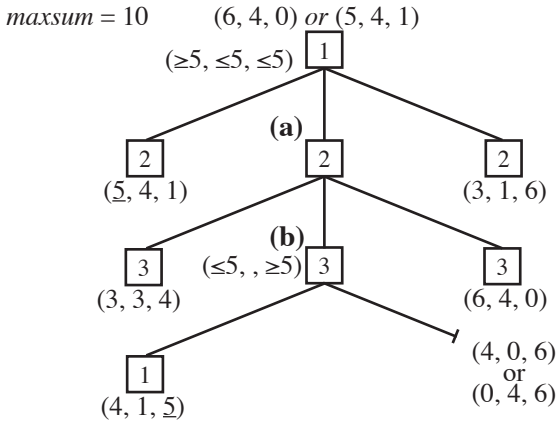

Figure 5.16: The failure of deep pruning.

bad for us, and vice-versa. In multi-player games we instead expect a game to become constant-sum, where the sum of all players' scores are constant. Most games are constant sum when you consider the final outcome, however they may not be during the actual search.

In Hearts, for instance, there can be as few as 0 and as many as 17 points played on the first two tricks. If we need a game to be constant-sum and our static evaluation function is non-negative, we can simply use the ratio of each player's score to the sum of all scores to make the max^n value of each node constant sum. But, it is not implicitly wrong if our evaluation isn't constant sum. The decisions made by max^n will simply reflect our evaluation function. In the case of Hearts, if our evaluation function is just the number of points taken so far, it can be considered that there is a implicit extra player in the game whose static evaluation is the number of points not yet played.

For the remainder of our discussion of pruning algorithms, we will assume that games are constant sum, knowing that we can usually make a game constant-

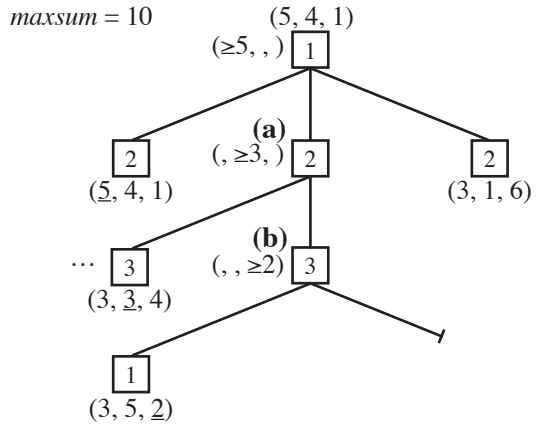

Figure 5.17: Combining max^n scores to limit value propagation.

sum if needed. In fact, it will usually be advantageous to do so, given the pruning benefits that are possible in a constant-sum game tree. We also assume that players' scores are bounded, which is usually the case when we run on a computer.

5.11 Limiting Max^n Value Propagation

In order to develop effective new algorithms that can prune deeper into a max^n tree, we must return to the analysis of why deep pruning failed in the first place. Deep pruning failed because nodes that couldn't be the max^n value of the game tree could still affect the max^n value of the tree. Thus, if we can restrict how max^n values will affect each other within the search tree, we may be able to prune more. The last-branch and speculative pruning algorithms both do this, however they take different approaches when they prune these nodes. Last-branch pruning is a directional special case of speculative pruning, which is a non-directional algorithm.

Looking at Figure 5.16 (which is the same as Figure 5.12), we know that

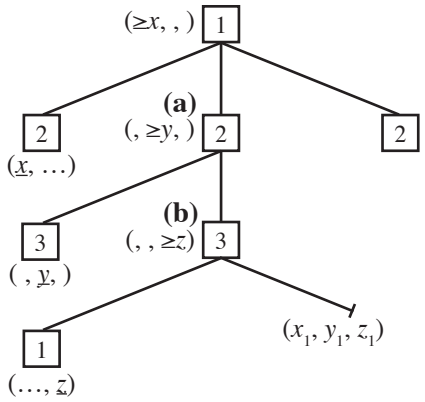

Figure 5.18: Combining scores to limit value propagation in general.

Player 1 will never get a better value than 5 at node (b). But, to prune at (b) correctly, we must show that Player 1 cannot get a better max^n value at the root from either node (b) or node (a), as the values at (a) may interact with unseen values at (b) to affect Player 2's, and thus Player 1's move. In this case, deep pruning failed because the value at the right child of (a) was better for Player 2 than a previous child of (a). If the children of (a) were ordered optimally for Player 2, or if there was no right child at (a), the deep prune could not have affected the max^n value of the tree.

While shallow pruning only considers two players' bounds when pruning, we can actually use *n* players' bounds in a *n*-player game. We demonstrate this in Figure 5.17. Before we search the second child of node (b) each player has already searched one or more branches. This provides a lower bound on each player's score. In this case, Player 1 has a lower bound of 5 from the left branch of the root, Player 2 has a bound of 3 from the left branch of (a), and Player 3 has a bound of 2 from the left branch of (b). The sum of these bounds is 10, which is greater than or equal to *maxsum*. We can thus show that any unseen value at (b) cannot be the max^n value

of the tree.

Theorem 5.3: Assuming we break ties to the left in a max^n game tree, if the sum of lower bounds for a consecutive sequence of unique players meets or exceeds *maxsum*, the max^n value of any child of the last player in the sequence cannot be the max^n value of the game tree.

Proof: We provide a proof by contradiction. Figure 5.18 shows a generic 3-player game tree. In this figure Player 1 has a lower bound of x at the root, Player 2 has a lower bound of y at (a), and Player 3 has a lower bound of z at (b). Given that these values sum to *maxsum*, assume there is a value v at the right child of (b) which will be the max^n value of the game tree.

Let $v = (x_1, y_1, z_1)$. For v to become the max^n value of the tree, each player must prefer this move to their current move. Since ties are broken to the left, z_1 must be strictly better than z, y_1 must be strictly better than y, and x_1 must be strictly better than x. Thus, $z_1 > z$, $y_1 > y$ and $x_1 > x$. So, $x_1 + y_1 + z_1 > x + y + z \geq maxsum$, and $x_1 + y_1 + z_1 > maxsum$. But, by the definition of *maxsum*, this is impossible. So, no value at the right child of (b) can be the max^n value of the game tree. By the same logic, where $z_1 = z$, the left child of (b) also cannot be the max^n value of the game tree. While this is the 3-player case, it clearly generalizes for n players. □

While we have shown that we can combine n players' scores to prove a max^n value will not propagate up a max^n tree, we must also show that a prune in this case will not affect the max^n value of the entire game tree. Last-branch and speculative pruning address this problem in similar ways. Neither algorithm, however, will

prune more than *n* levels away from where the first bound originates.

5.12 Last-Branch Pruning

When a sequence of players have bounds appropriate for pruning under Theorem 5.3, last-branch pruning guarantees that the prune will be correct by only pruning when the intermediate players in the sequence are searching their last branch.

We can see this in Figure 5.17. To prune correctly, we observe that after searching all the left children of node (a) Player 2 has only two choices: the best max^n value from a previously searched branch of (a), or the max^n value from (b). If Player 2 chooses the best max^n value from a previously searched child of (a), (3, 3, 4), Player 1 will get a lower score at (a) than his current bound at the root. Theorem 5.2 shows that the best max^n value at (b) for Player 3 and can be better than the current bound for Player 2 or for Player 1, but not for all three players. So, if Player 2 chooses a value from (b), it must also have a lower max^n value for Player 1 than his bound at the root. Thus, Player 1 will not get a better score at (a), and we can prune the children of node (b).

For last-branch pruning to be correct, in addition to the conditions from Theorem 5.3, Player 2 must be on his last branch, and the partial max^n value from Player 2's previously searched children must not be better for Player 1 than his current bound at the root. In the *n*-player case, all intermediate players between the first and last player must be searching their last branches, while Players 1 and *n* can be on any branch after their first one.

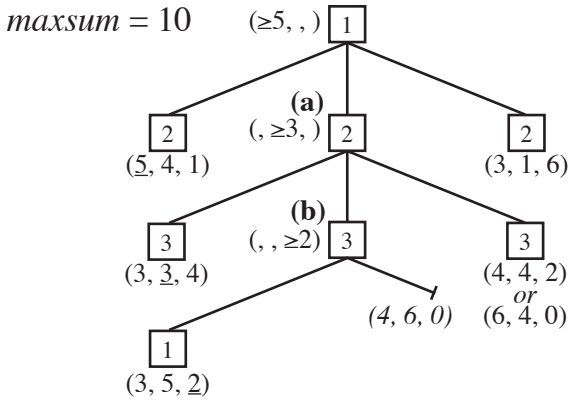

Figure 5.19: Speculative pruning a max^n game tree.

Last-branch pruning has the potential to be very effective, particularly in trees with low branching factor. Instead of only considering 2 players' scores, it compares *n* players' scores. In fact, when all nodes in the tree have the exact same evaluation, last-branch pruning will always be able to prune when players reach their last branches, while shallow pruning will never be able to.

The only drawback to last-branch pruning is that it only prunes when intermediate players between the first and last player are all on the last branch of their search. For a game with branching factor 2 this is already the case, but otherwise we can use speculative pruning.

5.13 Speculative Pruning

Speculative pruning is identical to last-branch pruning, except that it doesn't wait until intermediate players are on their last branch. Instead, it prunes speculatively, re-searching if necessary.

We demonstrate this in Figure 5.19. At the root of the tree, Player 1 is guaranteed 5 points. At node (a), Player 2 is guaranteed 3, and at node (b), Player 3 is

guaranteed 2. Because $5 + 3 + 2 \geq maxsum = 10$, we could prune the remaining children of (b) if node (b) was the last child of node (a).

Suppose we do prune, and then come to the final child of node (a). If the final child of node (a) has value (4, 4, 2), we know Player 1 will not move towards (a), because no value there can be better for Player 1. But, if the value at (a) ends up being (6, 4, 0), the partially computed max^n value of (a) will be (6, 4, 0). With this max^n value, Player 1 will choose to move towards node (b). Because this has the potential to change the max^n value at the root of the tree, we will have to search node (b) again using new bounds. This occurs when Player 2's nodes are ordered sub-optimally. With an optimal node ordering we will never have to re-search a subtree.

In general, we have to re-search pruned nodes if, on a mid-level branch, we find a new value for which both that player and the first player have better scores. If we wish to preserve the order of tie-breaking in the tree, we must also retain some information about the order of nodes expanded. Nodes that can be pruned by last-branch pruning will be always be pruned as part of speculative pruning. Following is the pseudo-code for speculative pruning:

```
specmax^n(Node, ParentScore, GrandparentScore)
{
       best = NIL; specPrunedQ = NIL;
       if terminal(Node)
              return static eval(Node);

       for each child(Node)
              // check to see if parent prefers this move
              if (best[previous Player] <= ParentScore)†
                     result = specmax^n(next child(Node),
                                   best[current Player], ParentScore);
              else
                     result = specmax^n(next child(Node),
                                   best[current Player], 0);
              if (best == NIL)
                     best = result;
              else if (result == NIL) // child was spec. pruned
                     add Child to specPrunedQ;
              else if (best[current Player] < result[current Player])
                     best = result;
                     // if we find a better move we have to re-search
                     if (best[previous Player] > ParentScore)
                            re-add specPrunedQ to child list;
              if (GrandparentScore+ParentScore+
                                   best[current Player] > maxsum)
                     return NIL; // speculatively prune
       return best;
}
```

As can been seen, it is reasonably simple to perform speculative pruning in practice. In the 3-player implementation, the specmax^n function takes 3 arguments, the current node, the best score at the parent node, and the best score at the grandparent node.

At the line marked † we check to see if our parent can get a better score from the partial max^n value of this node than from one of his previously searched nodes. If this is the case, we cannot use the parent's bounds to help prune the children of the current node.

When a node is speculatively pruned, the specmax^n function returns NIL. All nodes that have speculatively pruned children are added to a list of pruned nodes, and if a child is found with a max^n value that better for both the current node and the

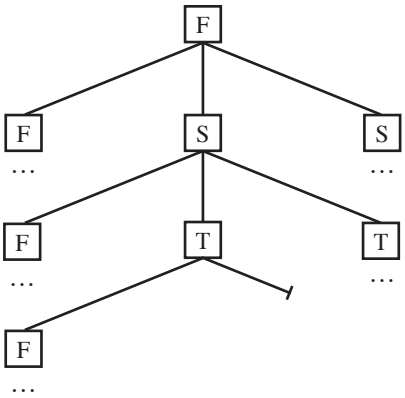

Figure 5.20: Analysis of speculative max^n pruning.

parent node, then the speculatively pruned nodes will have to be re-searched. This pseudo-code assumes that players always alternate plays in the tree, as in Chinese Checkers. In card games, where this may not be the case, we must also check to see if the last *n*-1 consecutive players in tree are unique.

5.14 Last-Branch and Speculative Max^n Best-Case Asymptotic Analyses

To analyze the best-case performance of speculative max^n pruning, we form a recurrence based on the type of nodes that occur in the search. We will do the analysis first for the case of 3-players, and then generalize it to *n* players. In the case of a 3-player game, we consider three types of nodes, which are irrespective of who is playing at that node. We have **F**irst-level nodes, **S**econd-level nodes, and **T**hird-level nodes. A node that has no bounds from its parents is a first-level node. A node with a bound from just its parent is a second-level node, while a node with bounds from both its parent and grandparent is a third-level node. We demonstrate this in Figure 5.20.

The left child of every node is a first-level node, because there are never

b	$b^{2/3}$	asymptotic b
2	1.5874	1.8393
3	2.0801	2.4675
4	2.5198	3.0000
5	2.9240	3.4755
10	4.6416	5.4191
1000	100.00	103.61

Table 5.21: Branching factor gains by speculative max^n in a 3-player game.

bounds on the first child of any node. The remaining children of a first-level node are second-level nodes. In the optimal case a third-level node will only have to expand its left-most child, and the remaining children will all be pruned. If we write out the ratio of nodes from one level to the next as a recurrence, we get the following equations, where b is the branching factor of the tree before pruning:

$$F(n) = F(n - 1) + S(n - 1) + T(n - 1)$$

$$S(n) = (b - 1) \cdot F(n - 1)$$

$$T(n) = (b - 1) \cdot S(n - 1)$$

Solving in terms of $F(n)$, we get:

$$F(n) = F(n - 1) + (b - 1) \cdot F(n - 2) + (b - 1)^2 \cdot F(n - 3)$$

The solution to this recurrence is the solution to the equation:

$$x^3 - x^2 - (b-1) \cdot x - (b-1)^2 = 0$$

The solution of this equation will give us the asymptotic branching factor as b grows large. Analyzing the general solution for a cubic equation will show that for large b this will approach $b^{2/3}$. In practice do not reach the limit until b is larger than most games we consider in practice. We give sample values for b given an op-

timal ordering of nodes in a 3-player game in Table 5.21. The first column contains sample values for b, the second column contains $b^{2/3}$, and the third column contains the actual optimal value of b.

For a general n-player game, our tree and recurrence will be similar. Each type of node in the game will produce a single first-level node, and (b - 1) nodes of the next level classification. So, the equation which solves the general recurrence is:

$$x^n - x^{n-1} - (b\text{-}1)^1 \cdot x^{n-2} - (b\text{-}1)^2 \cdot x^{n-3} - \ldots (b\text{-}1)^{n-1} = 0$$

As b grows large the solution for x in this equation will be on the order of $O(b^{n-1/n})$. The easiest way to see this is to consider which terms are asymptotically largest in the equation. Because in a recurrence of this form x will grow towards b^k, where $k < 1$, the largest terms must be the first term and the last term. Reducing to just these terms and solving yields the asymptotic result above.

In the case of $b = 2$, last-branch and speculative pruning are identical, so we can use these recurrences to solve for the best case of last-branch pruning when $b = 2$. Specifically, for the 3-player case in the best case our branching factor will be reduced to the solution of:

$$x^3 - x^2 - x - 1 = 0$$

Solving for x, we get 1.839. The general solution to these equations are related to the Fibonacci sequence.

The average case analysis of speculative and last-branch pruning is more complex. Assuming $\max^n$ convergence, we will still get some pruning, as we will

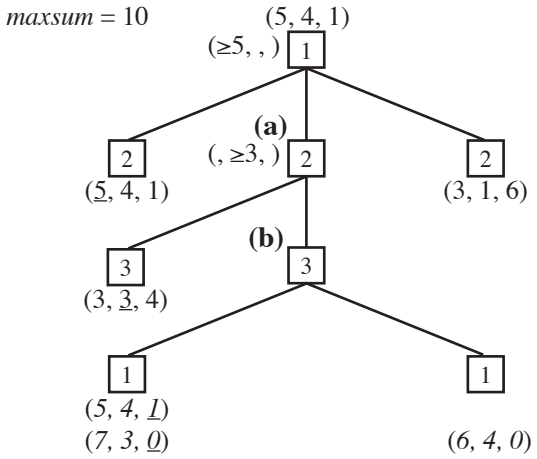

Figure 5.22: Discrete cut-off evaluations

expect consecutive player's scores to sum to *maxsum*, but we have not analyzed how the requirements for re-searching nodes will affect the average case complexity.

5.15 Discrete Evaluation Functions

For all these pruning techniques, it is possible to use tighter bounds for pruning when the evaluation function has discrete as opposed to continuous values. For speculative and last-branch pruning we can infer this from the proof of Theorem 5.3, but it will also work for shallow and branch-and-bound pruning. In this proof we see that, for a value to affect the max^n value of the tree, $x_1 > x$, $y_1 > y$, and $z_1 > z$. Suppose the minimum delta of a player's score is μ. Since all players in the game must do strictly better than their previous max^n value to change their move to a new max^n value, we can combine this into our bounds.

We demonstrate this in Figure 5.22. At the root of the tree, Player 1 is guaranteed a score of 5, and at node (a) Player 2 is guaranteed 3 points. In this example μ =

1, so for these players both to prefer to move towards (b) they must get at least 6 and 4 points respectively. Because *maxsum* is 10, we know if Player 3 gets more than 0 points, Players 1 and 2 can't both get better than their current best scores. So, instead of pruning when Player 3 gets 10 - 5 - 3 = 2 points, we can prune when Player 3 gets 1 point. It follows from this that we can always prune if $\sum scores \geq maxsum - \mu \cdot (n - 2)$, where $\sum scores$ are the current bounds for the players in the tree.

We can then use our tie-breaking rule to improve this. Because ties are broken to the left, we can prune if Player 3 gets 0 points at the left branch of (b) and Player 1 and 2 don't get 6 and 4 points respectively. If, for instance, the score is (7, 3, 0), Player 2 won't choose this value over the left branch of (a). In addition, Player 3 will only choose a better value than 0 from the unexpanded children of (b), which will meet our earlier conditions for pruning. Thus, we can also prune if $\sum scores \geq maxsum - \mu \cdot (n - 1)$ and if on the first branch of the node whose children are being pruned the other $n - 1$ players don't all have better scores than their current best bound.

5.16 Optimality of Last-Branch and Speculative Pruning

Given new pruning algorithms for max^n, we naturally must ask if they are the best possible, or if it is possible to do better. It ends up that both last-branch and speculative pruning are not optimal, in that there are similar techniques which can prune nodes which these techniques will not prune. But, whether these techniques can provide any real games in practice is another question. We demonstrate this for

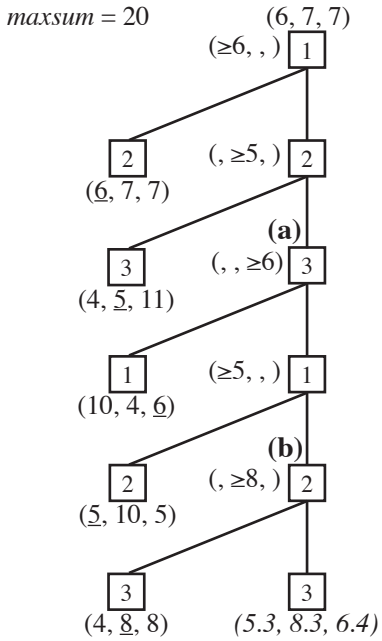

Figure 5.23: Legal pruning example not covered by last-branch pruning.

last-branch pruning with tree in Figure 5.23. In this tree it is never the case that three consecutive players have bounds that sum to *maxsum* (20), but the best bounds from all players anywhere in the tree do sum to *maxsum*. These bounds come from Player 1 at the root, 6, Player 3 at (a), 6, and Player 2 at (b), 8. Also, each player always has a better score on their left branch than their current (partially computed) score on their right branch. Finally, we will make no assumptions about a minimum μ by which scores can change.

In this case, we can try any legal value for the right child of (b), and it will not become or affect the max^n value of the game tree. The most interesting value to consider is (5.3, 8.3, 6.4). This value can propagate up to the top of the tree, but Player 1 will not choose it over (6, 7, 7) at the root. There are similar methods that we can use for speculative pruning, but we will not demonstrate them here. The

main interest for exploring such algorithms would be to define the optimal max^n pruning algorithm.

5.17 Approximate Deep Pruning

While effective, the descriptions of last-branch and speculative pruning here still do not prune more than *n*-branches away from where any bound originates, unlike alpha-beta which can prune an arbitrary distance away from the node at which the bound originated. The issue is further complicated in games like card games, as players will not always play consecutively. (Each time a trick is taken, the lead jumps to the player who won the trick.) When this happens, the bounds must be reset, and so pruning is more limited. Also, the more players in the game, the more complicated the process becomes.

It is also possible to come up with many special cases in which bounds can be passed across more than *n* levels, such as when a player plays at two consecutive levels in the tree. However, this can be quite complicated in practice. While we can describe many such algorithms, we would not want to implement any of them.

A solution to this dilemma is to simplify the speculative pruning procedure so it looks more like alpha-beta pruning:

```
approxdeepmax^n(Node, allBest)
{
       myBest = allBest;
       if terminal(node)
              return static eval(node);

       for each child(Node)
              myBest[Node->player] = approxdeepmax^n(next child(Node),
                                                     myBest);
              if (∑myBest ≥ maxsum)
                     return myBest[Node->player];
       return myBest[Node->player];
}
```

This implementation of the algorithm eliminates re-searching and it keeps one set of bounds for all players that is copied and passed down through the game tree. Such an implementation is simple, and it allows us to prune beyond *n*-levels from where a bound originated, but it isn't guaranteed to be correct, meaning it isn't guaranteed to correctly compute the max^n value of the game tree.

If our nodes are ordered close to optimally, this approximate procedure will calculate the max^n value of the game tree correctly. But, in practice this isn't guaranteed.

Thus there is a trade-off at stake. Approximate deep max^n will be able to search deeper into a max^n tree than any other known algorithm at this point, but in doing so, it might make a mistake in its calculations. On the other hand, searching deeper is one of the best ways to improve play. So, the end question will be whether the gains found in deeper search offset the possibility of returning the incorrect max^n value of the tree. We will address this question further in our experimental section.

5.18 Summary of Multi-Player Game Pruning Algorithms

As we have covered a large variety of pruning techniques in this chapter, we conclude with a summary of the techniques in Table 5.24. This table lists each of the techniques that we have developed, along with the known analysis for each algorithm. Although we have not listed or mentioned it before, each of these algorithms uses a depth-first search strategy for asymptotic space usage of O(b·d), and has a worst-cast performance of $O(b^d)$

The best case of branch-and-bound algorithms comes from the fact that if our heuristic has perfect knowledge of the tree, and we have perfect ordering, we will only need to search the left branch of the tree, as everything else will be pruned. Obviously this will not usually occur in practice. We have not analyzed the average case.

We also have not yet been able to analyze the average case of last-branch or speculative pruning. But, because they compare the bounds of all players in the game, they should have better average case performance than shallow pruning.

In the bottom of the table we have listed sample games and the techniques which can be used in those games. Note that we are only listing whether a technique can or cannot be used; this says nothing about the effectiveness of the technique. As we have shown, for instance, Shallow pruning will work in Spades, but we can never achieve the best case in practice. For card games, we are assuming that the static evaluation function is related to the points taken in the game, which is usually the case.

	Decision Rule							
	Paranoid	Max^n						
Algorithm	Alpha-beta	Immediate	Shallow	BnB	ABBnB	Last-Branch	Speculative	Approx. Deep
Time Complexity								
best case	$b^{d\cdot(n-1)/n}$	$b^{d/n}$	$\frac{1}{2}\cdot(1+\sqrt{4b-3})$	$b\cdot d$	$b\cdot d$	1.839 (for b=2)	$b^{d\cdot(n-1)/n}$	$O(b^{d\cdot(n-1)/n})$
average case		$o(b^d)$	b^d			$o(b^d)$	$o(b^d)$	$o(b^d)$
Game								
Abalone	✔	✔				✔	✔	✔
Chinese Checkers	✔	✔				✔	✔	✔
Cribbage	✔	✔		✔		✔	✔	✔
Hearts	✔	✔		✔		✔	✔	✔
Pinochle	✔	✔	✔	✔	✔	✔	✔	✔
Spades	✔	✔	✔	✔	✔	✔	✔	✔

Figure 5.24: Decision rules/algorithm complexity and domains for which they can be applied.

Chapter 6

Additional Techniques for Improving Performance

Besides the pruning algorithms considered in Chapter 5, there are many other techniques that have been developed to improve the performance of game programs. Not all these techniques have interesting new properties in multi-player games. We will discuss some of the techniques that are more interesting in multi-player games here, including iterative deepening, zero-window search, move ordering, opening books, end-game databases and transposition tables.

6.1 Iterative Deepening

Iterative deepening is a common technique used in search algorithms of all sorts. The basic idea is to break the search process into multiple iterations over the same tree, each of which is to a greater depth than the previous iteration. In a card game, for instance, we might search 1 trick deep on the first iteration, 2 tricks on the second iteration, and so on. Because most if not all of the search problems tackled in AI have exponentially large search spaces, we can search iteratively deeper into the tree with no asymptotic cost, as the cost of the last iteration will outweigh all

previous iterations.

Since we may not know ahead of time how expensive any search will be, this method has been widely used to guarantee that an answer from a shallow search is always available, and it means that we can cut off search at any time, and have the best answer we were able to calculate in the time provided.

Obviously this answer will only be as good as the quality of the search and evaluation function used for play. For games like chess, techniques like quiescence search were developed to account for some of the anomalies that resulted from fixed search depths. But, in general, we would expect the result of an intermediate search to be an approximation of the deeper search. Particularly for two-player search in trick-based card games, the result of a partial search can be quite strong.

Let us consider a two-player game for which we have a monotonically increasing evaluation function that increases at most 1 at each iteration in the search tree. An example of this would be if our iterative deepening corresponds to tricks in Spades, so that 1 trick would be available at the first iteration, 2 tricks at the second iteration, and so on. In an evaluation function with this property, there will be exactly one point available in the first iteration, exactly two in the second iteration, and exactly n points in the nth iteration.

Theorem 6.1: Given the conditions to follow, the minimax value of a game tree will only increase by some delta μ or stay the same over successive iterative deepening searches. This occurs when players' scores are monotonically increasing in the game, that there is a minimum delta, μ, by which scores can change, and that the

sum of all scores changes by exactly μ on each successively deeper iteration into the game tree.

Proof: The key to this proof is that on each successive iteration, a player cannot find a new line of play on the new iteration that will be better by a margin greater than μ.

We prove this is the case by induction. On the first iteration, there will only be μ points available in the game. Since μ is also the minimum change any score can make, by definition one player will get μ points, and the rest will get 0.

Now, after searching some depth d, assume we have a score of x, and our opponent has a score of y. (It will be the case that $x+y = d \cdot \mu$.) If we search to depth d+1, the resulting scores should either be $x+\mu$ and y, or x and $y+\mu$. This argument is symmetric, so we will show this by contradiction for the player with x points. Assume that at a search d+1 the player gets a score greater than $x+\mu$. Since the additional search depth can only increase his score by μ, there then must be a path of depth d by which he is getting at least $x+\mu$. But, if that is the case, this player would have played this line at depth d to get $x+\mu$ at that depth. Since minimax is guaranteed to find the best path available, this can't be the case, and the theorem holds. All other cases are symmetric, so the proof holds. □

Unfortunately Theorem 6.1 doesn't hold for a multi-player game using the $\max^n$ algorithm. Consider the hands in Figure 6.1. This is for the game of Spades, where players are trying to take as many tricks as possible. In (a) we consider the line of play that results from a search of depth 5. Player 1 can take the first 2 tricks,

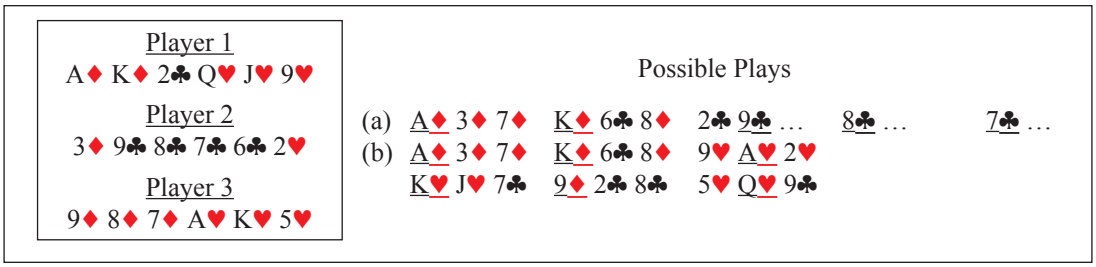

Figure 6.1: Effect of iterative deepening in max^n.

but after that he can't take any more, and so he doesn't care what card he plays. In this case we assume he plays the 2 of clubs, allowing Player 2 to take 3 tricks. So, the max^n value at this depth is (2, 3, 0). But when searching to depth 6, Player 1 discovers that he prefers to have Player 3 win the next trick after he wins the first two tricks. By doing so, Player 3 will eventually have to lead back a low heart, allowing Player 1 to take an extra trick. In this case, on the next iteration, the max^n value of the game tree (3, 0, 3). Although the sum of all players' scores only increased by 1, Player 2's score decreased by 2 and Player 3's score increased by 3.

In general, it is not possible to predict ahead of time how the scores in the game will change, and who might lose or win points. The reason this occurs in max^n game trees is due to tie-breaking. In line of play (a) in Figure 6.1 Player 1 has a tie in his analysis of what card to play on the third trick. But, on the next iteration of the game there are no ties in Player 1's analysis.

Searching to a different depth in a max^n game tree is going to change some max^n values in the game tree. This means that nodes that had their max^n value determined by a tie-breaking rule before may no longer be, and positions that didn't have ties before may have new ties introduced. Since changing a tie-breaking rule can

vastly affect the max^n value of the game tree, it follows that searching to a different depth in the tree can have the same effect.

In two-player games if we don't have a monotonic evaluation function there can be wide variations in node evaluation from one level to the next. This problem has generally been solved with quiescence search. That is, a search is not cut-off as long as there are pending actions that can greatly affect the static value of the node. Such actions would include moves into check or capture moves. But this additional search is needed because of the features of the game being considered.

Under max^n the volatility of max^n values with search depth is partly a property of the algorithm we use to search. This means that techniques like quiescence search are less likely to be effective under max^n.

6.2 Zero-Window search

Zero-window search, originally called Scout search [Pearl, 1984], originated in two-player games. The idea behind zero-window search is to turn a game tree with a range of evaluations into a tree where every leaf terminates with a win or a loss. This is done by choosing some value v, and treating a terminal node as a win for max if its evaluation is $> v$, and as a loss if it is $\leq v$. Combining this approach with a binary search will suffice to find the minimax value of a game tree to any precision. This assumption results in highly optimized searches in win-loss trees, where we can prune away most of the game tree. We will first demonstrate here how zero-window search can be combined with iterative deepening for highly effective

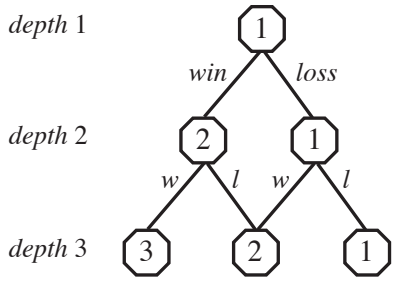

Figure 6.2: Decision tree for zero-window search limit.

iterative searches. Unfortunately, we cannot use zero-window search in a multi-player $\max^n$ tree, but we can use it in a paranoid search tree.

6.2.1 Zero-Window Iterative Deepening

Consider a two-player game which meets the requirements of Theorem 6.1, namely that for a given player they will either get the same score as the previous search or the previous score plus some constant μ.

This theorem means that when we are doing iterative searches in a suitable game tree, we do not need to do a binary search to find the minimax value of the game. Instead we can do a single search at each iteration. We demonstrate this with a decision tree in Figure 6.2. This tree is used to decide which bound to use on each successive iteration of a zero-window search. The value inside each node is the zero-window search limit used at that depth, and the left and right branches represent whether we won or lost at the previous depth. So, at the root of the tree we begin by searching depth 1 with a zero-window limit of 1. If the result is a win, we move down the left branch, and we will search depth 2 with a limit of 2. If we lose, we will still use a limit of 1 at search depth 2. In this way we can always use

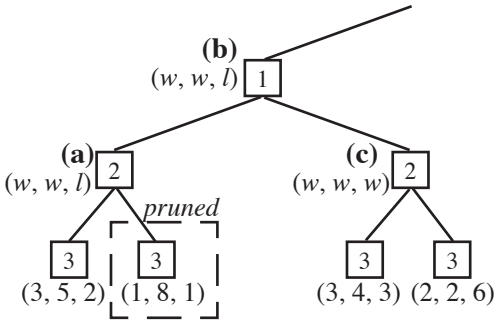

Figure 6.3: Finding bounds in a max^n game tree

a single zero-window search at each depth and still be guaranteed to calculate the exact minimax value of the tree at that depth.

Unfortunately this technique will not work with the max^n algorithm, as we cannot bound how a player's score changes from one depth to the next in a max^n tree. In fact, we can't even use zero-window search to determine whether a player's score will be above or below some bound, as we can in a two-player game.

6.2.2 Failure of Zero-Window Max^n Search

While there are limitations on pruning during the calculation of the max^n value of a tree, it is not immediately obvious that we cannot somehow prune more if we just try to calculate the bound on the max^n value of a tree, instead of the actual max^n value.

Suppose we want to consider all scores > 2 as a win. We illustrate this in Figure 6.3. Since Player 2 can get a score of 5 by moving to the left at node (a), Player 2 will prune the right child of (a), and return a max^n value of (w, w, l), where w represents a win and l represents a loss. However, at node (b), Player 1 would then infer that he could win by moving towards node (a). But, the exact max^n value of (a)

is (1, 8, 1), and so in the real game tree, Player 1 should prefer node (c) over node (a).

So, the only bounds we can get on the max^n value of a tree are those that come from a search with shallow pruning, which, given no additional constraints, is already optimal among directional algorithms.

[Korf, 1991] shows that "Every directional algorithm that computes the max^n value of a game tree with more than two players must evaluate every terminal node evaluated by shallow pruning under the same ordering." We can now expand this statement:

Theorem 6.2. Given no additional constraints, every directional algorithm that computes either the max^n value or a bound on the max^n value of a multi-player game tree must evaluate every terminal node evaluated by max^n with shallow pruning under the same ordering.

Proof: [Korf, 1991] has already shown that shallow pruning is optimal in its computation of the max^n value of a tree. If we replace backed-up max^n values in a game tree with just bounded max^n values, there may be ties between moves that were not ties in the original max^n tree, such as at node (a) in Figure 6.3. In fact, for any bound it is a trivial task to create a tree for which ties are broken differently with that bound than in the original tree. However, if we add new ties into the game tree, no tie-breaking rule will be able to always make the same choices for breaking ties in the new tree, as it did in the old tree. Thus, the underlying max^n value that we are trying to bound may change, and we can no longer guarantee an accurate bound on

its value. □

6.3 Move Ordering for Pruning Algorithms

In two-player games, alpha-beta pruning has an average case performance of $b^{3 \cdot d/4}$, but a best-case performance of $b^{d/2}$. Thus, there is an incentive to order the successors of each node as well as possible, as it is the successor ordering that makes the difference between the average and best-case performance. Much discussion and previous research went into alpha-beta pruning to estimate the size of game trees so that researchers would know whether or not the node ordering was sufficient for alpha-beta pruning to prune the game tree optimally.

As for multi-player games, node ordering can increase the amount of pruning that occurs under shallow pruning. But, given the average case model of shallow pruning and the fact that shallow pruning will never occur in many games no matter what ordering we use, we haven't invested very much effort in improving node ordering for shallow pruning. However, in speculative pruning we can guarantee better pruning results if we can order our successors closer to optimally.

While the obvious solution to this problem is to add domain-specific knowledge to order moves within a game tree, there have been other proposals. One such proposal is the History Heuristic [Schaeffer, 1989]. The history heuristic was first applied in the game of Chess. This is essentially a learning method that uses the result of offline play to learn the best ordering possible. It works by first building a table of all possible moves in the game. In Chess there are 64 possible piece posi-

tions on a board. For the chess implementation a table was set up for each possible *from* and *to* positions of a move, regardless of the piece that was moving.

Then, each time a search algorithm returns from a node with the best possible move at that node, it re-weights the returned move within the table. It then uses the weighting of nodes within the table to order the successors of a node. The weighting of moves in the table was initially zero. After a search to depth d had returned a particular move as the best move at that node, the table entry for that node was updated by adding 2^d to the current sum at that table entry. This is because the deeper the search that led to a certain move the more likely it is to be a good move in general. When new moves were considered, they were sorted from largest to smallest values based on the results in the table. Values should be learned offline through self-play at the same depths that will be used for online search.

[Schaeffer, 1989] reports that this technique was able to learn an ordering that was as good as the best hand-tuned ordering produced from expert knowledge of the game. Given the simplicity of this method, similar results in other domains would be welcome.

We implemented this for trick-based card games in the following manner. We created 53 tables of 52 values each. Each table represents the current winning card in the trick when the player in question is about to move, with one extra table for which card to play when they are in the lead. Each table entry began at 0 and was updated through self-play. We did not adjust the table when we were at a node that only had a single legal move. After playing multiple games we analyzed the tables

rank	♠	♦	♣	♥
A	0	15846	16378	316562
K	5081366	10732	3472	129116
Q	8392112	7630	4774	58678
J	3247574	5472	4402	51348
10	2931582	3212	12992	11432
9	2396322	7220	2820	24566
8	1540746	4122	1966	9246
7	1455590	3502	3086	8950
6	992640	764	1116	2070
5	490956	5872	1488	6730
4	431818	1546	1556	1450
3	145032	1296	280	910
2	56600	1152	832	620

Table 6.4: Learned move ordering for Hearts given an A♠ lead.

and found that it had learned fairly reasonable tables. We demonstrate some of the values in Table 6.4. This table shows values for the game of Hearts when the Ace of Spades has been lead. The first-choice card to play in this situation is the Queen, followed by the King. If you can't follow suit, the best plays are the Ace through Jack of Hearts. These are the expected ordering that should be learned.

Despite this, there are several issues that arise, particularly in multi-player games, when attempting to use such methods to order successors. First, for Hearts this set of tables is not sufficient to conveniently represent the information we might want to use when ordering our moves. If we know, for instance, that the only spades left are the ace and king, it is a reasonable move to lead the Queen of Spades. This is an easy situation to check with a custom ordering function, and will almost always be the correct move, but there is no simple way for such a method to learn this au-

tomatically. Similarly if the Queen of Spades has already been played, the order in which we consider moves may change drastically. The Queen of Spades is a special card in Hearts, and we can build multiple tables to account for whether it has been played or not, but there are still other issues beyond this.

The bigger issue is that ordering our moves to search the minimum sized max^n tree may actually cause us to play suboptimally. This goes back to one of fundamental issues with max^n, which is that randomly breaking ties in the tree can randomly affect the max^n value of the tree. If we want to prune the tree most effectively our tie-breaking rule must be implicitly defined by the way we order our successors. Thus, our successor ordering is relevant to both the tie-breaking rule we want to use in the game and whether our ordering can create a minimal search tree.

Let us specifically consider a situation in a multi-player game where there are multiple ways that we can a break tie within a game tree. Suppose we have two moves with the same max^n value, but in practice one move is much better than the other. Depending on the situation, the better move may require more analysis than the worse move. Thus, if our goal is to build the smallest search tree, we may end up preferring the worse move.

It is important to note that these aren't criticisms of the history heuristic, and it indeed learned something reasonably close to our own ordering. But, these issues will be relevant to any attempt to order nodes optimally in a multi-player game tree.

This isn't as much of an issue in two-player games because the way we break

ties can't change the minimax value of the game tree, so ordering our moves to create the minimal game tree is equivalent to ordering our moves from best to worst.

In experiments run with the history heuristic, there was no discernible difference in performance or node expansions while using the heuristic when compared to our static ordering function. We postulate that because it is quite easy to order our successors well in card games the history heuristic will not provide easy benefits. In Chinese Checkers, where we have done well ordering moves by how far they move our pieces across the board, we also expect that the history heuristic will not provide major benefits. But, in a game like Abalone, where the moves are more complicated and less easy to evaluate, the history heuristic may provide greater benefits. Our initial experiments in Abalone have proved promising, however we have not resolved all the issues involved well enough to present definitive results.

6.4 Memory Usage

All the search algorithms discussed in this thesis use relatively little memory, on the order of the search depth times the branching factor of the tree, O($b \cdot d$). That is because they only keep the current search path in memory at any one time. This means that the search can operate in a few kilobytes of memory, while a current computer will often have 500-1000 megabytes of main memory, and hundreds of gigabytes of disk storage. Since searching with these algorithms is not going to come close to using the full resources of the computer, we would like to develop techniques that will take advantage of the large memory of the computer to help

speed the search process.

We note, however, that processor speeds are currently growing much faster than memory bus speeds, meaning that the cost of accessing main memory is becoming more and more expensive, and there is no reason to expect that to change in the near future. Thus, there can be a high performance penalty for accessing main memory too often, if the consequent savings are too small.

There are three related techniques that have been used to take advantage of free memory on the computer, opening books, end-game databases, and transposition tables. Opening books and end-game databases are simply large tables of precomputed positions and their relative values, while transposition tables are dynamically calculated tables of values. We will discuss all three methods here within the context of multi-player games.

6.4.1 Opening Books

Opening books were first used in the context of Chess, where an analogous concept exists for human Chess players. There are sets of standard openings and responses that have been well analyzed, and expert players usually memorize as many such openings as they can. Thus, a computer can also take advantage of such tables, as they usually involve much more detailed computation than a computer will usually be able to make on any given move. This helps avoid making simple mistakes early in the game. In addition to using pre-made human tables, it is also possible to do offline search before tournament or other competitive play that can be saved in the form of an opening book.

Opening books are useful because many games always begin in the same starting position. This means that there are a relatively small number of calculations that can be done ahead of time. Opening books can also be used to help tune static evaluation functions. But games like Scrabble, or card games, which have no fixed starting position, will generally see little or no gain from an opening book, because the number of possible openings is related to the possible hands you can begin with, and enumerating these hands is much more expensive than doing the analysis from any given starting position.

The most obvious multi-player games for which opening books would be useful are Chinese Checkers and Abalone. In Chinese Checkers, the players usually have a few moves before their pieces begin to interact with each other, so one approach, besides doing a full search from the initial state, is to build an opening book from the single-agent space of moves for the optimal first few moves, and then to analyze those moves in relation to your opponents moves.

The biggest difficulty introduced by multi-player games is the issue of opponent modelling. The issue of opponent modelling cannot be completely ignored in a multi-player game, and an opening book will make implicit assumptions about the strategies our opponents are using. If our strategy and model of our opponents is adaptive over a game we will not be able to compute opening books for each combination of adaptive strategies.

This issue as a whole is beyond the scope of this thesis, but recognizing such a trait in game play, particularly repetitive game play, and adjusting to it, may invali-

date the calculations found in an opening book.

6.4.2 End-Game Databases

A similar method by which the memory of the computer can be leveraged to help search is in closing books. This method was most successfully applied in the implementation of Chinook [Schaeffer, et al, 1992], the best Checkers program in the world. One of the core pieces of this program is a closing book that contains all possible board positions in which there are 8 or fewer pieces on the board, and the exact win, loss, or draw value of those positions. This can be done efficiently using retrograde analysis. This is done by building a table of all the positions we wish to analyze. States for which the minimax value is known are marked. Then, for unknown states, the successors of that state are checked to see if they are a win, loss or draw. These values can be progressively backed up into the table until it is full.

Closing books are especially valuable in games where the number of combinations of pieces is small compared to the way that those pieces can play out the rest of the game. In Chess or Checkers it is possible that it will take a large search tree with many moves to finally play out a game position to a win or loss, but if those computations can be done offline, they can be stored at relatively little cost in comparison to the actual search. This contrasts greatly with games like card games, where the number of ways a game can be played out becomes much smaller as the game nears the end, but the total possible ways it could happen in any game is huge.

This brings us to the primary issues of closing books in multi-player games,

as compared to two-player games. At the close of a perfect information game, minimax is guaranteed to back up exact game-theoretic values. For minimax, this means that no matter what strategy our opponent uses, if we calculate a win, we are guaranteed to be able to win the game. For multi-player games, we will show how this might not be the case.

If we use the paranoid algorithm to do retrograde analysis, we have the same theoretical properties as minimax. So if such analysis indicates we will win, we are guaranteed a win no matter the strategy of our opponents. But, a loss under the paranoid algorithm is not a guaranteed loss. Instead, we must consider each of our opponents separately. Only a win for our opponent is a loss for us. This means there can be states for which we can never guaranteed a win or loss for any player in the game, because the actual result will depend solely on our opponents strategy.

If we choose to use the max^n algorithm for our retrograde analysis, every state in our table will evaluate exactly. But, implicit in our analysis is the tie-breaking rule that we use. It may be the case that a player that cannot win the game can decide by their actions who does win. What is more, we should probably distinguish between states in which a secondary player can make such a decision, and a state where we can guarantee ourselves a win.

Thus, while we can use closing books in a multi-player game, the issues surrounding closing books are more complicated than in two-player games. The exact resolution to these issues will be determined by the game being played. For the sake of illustration, we will discuss what we might do to create a closing book for the

three-player version of Chinese Checkers.

In Chinese Checkers we cannot distinguish the end-game by the number of pieces on the board, as players' pieces are never removed from the board. Additionally, for every winning state for each player there are roughly 2 trillion ways the other players can arrange their pieces on the board. So, it is infeasible to directly build a table like can be done in Checkers. Instead there are a few alternate approaches.

The approach we favor is to look at the single-player variation of Chinese Checkers. In this game only one player has their pieces on the board, and the goal is to simply get your pieces across the board in as few moves as possible. For a variation of Chinese Checkers with a slightly smaller board (49 legal positions and 6 pieces for each player), we have solved this problem exactly with a 14MB database of positions. Then, returning to the multi-player version of the game, this database can be used as both a partial opening and closing book for the game. As player's pieces close in on the end-game state, this table will have the exact number of moves needed to finish the game, and for states close to the end of the game, it will have a close approximation of that value. For the full-size game board (81 legal positions and 10 pieces for each player) such a table would take about 1.9 TB. This is on the upper range of the capacity of many modern machines, but it isn't out of the question.

Other possible approaches could require most of one's pieces to be in one half of the board, but allow a few pieces to be anywhere on the board, or to do similar analysis with several of one's opponent pieces also on the board. None of these

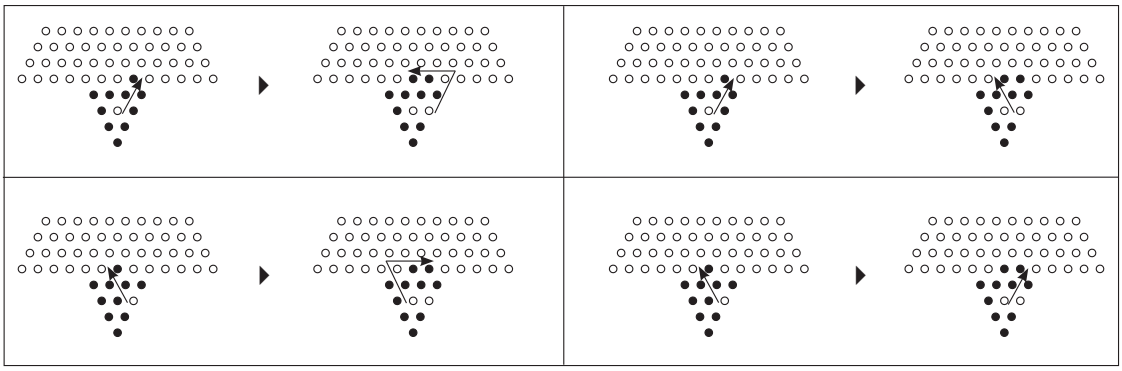

Figure 6.5: Four possible combinations of moves to get to the same state in Chinese Checkers.

methods look exactly like what we would find in traditional end-game databases, but they use the main idea behind the technique, and should boost the performance of a game implementation.

6.2.3 Transposition Tables

Although we often speak of game trees, most games are actually graphs. That is, there are states that can be reached by more than one path from the root. Thus, it is advantageous to be able to detect these states in order to avoid redundant searches. We demonstrate this in Figure 6.5.

This figure shows one portion of a Chinese Checkers game state. Only one player's pieces are shown on the board, as this example could be for a game with any number of players. We can see that in each of the four quadrants, there are two moves by the player that lead to the same final state of the board. While we might not be able to detect the moves that lead to the same state in advance, we can save the result of any calculations that occur below this state in the game tree, and when we return to the state again, instead of re-searching, we retrieve the saved results.

Not all moves will result in transpositions in the search space, so it is essen-

tial that the cost of lookup be small. This can normally be done by storing the entries in a hash table, which has constant-time lookup. It is important, however, that the search does not try to lookup or store nodes near the leaves of the search, as there will be little benefit to finding those states in the table, but there will be high latency costs from looking up and memory costs of storing the state in main memory. In addition, the state of the algorithm when searching a particular node must be stored, as the bounds from pruning that were in affect when the node was pruned must be compared to the current bounds being used.

In terms of multi-player games, transposition tables work essentially the same as they do in two-player games, with a few caveats. First, the order that moves are considered must be the same in all portions of the game tree. This is due to the fact that tie-breaking has the ability to arbitrarily affect the max^n value of the game tree. If we are not consistent in the way we break ties from one part of the tree to the next, transpositions from one part of the tree will potentially be stored with different results than would be found in the other part of the game.

Another issue in multi-player games is that it takes transpositions much longer to manifest themselves than in two-player games. In a three-player game of Chinese Checkers, for instance, the first possible transposition occurs after a player has their second move, or depth 4 in the game tree. Given no pruning, if we artificially limit the branching factor to 10 (it is well over 100 in the mid-game), we will have to search 10,000 moves before we begin to detect transpositions. If we are limited to 1 million moves per turn, each transposition will save a maximum of 100 moves,

and in a four-player game it would only save 10 moves.

Before speculative pruning was developed the max^n algorithm, due to its lack of pruning in Chinese Checkers, saw little gain from transposition tables. But, speculative pruning can greatly benefit from transposition tables, as they will minimize the cost of re-searching the children of a node.

To summarize, there are three things to note about transposition tables in multi-player game trees. First, they require that we be consistent with our node-ordering. Second, they can be less effective than in two-player games, due to the fact that it takes more moves for a transposition to occur. Finally, speculative pruning can benefit from transposition tables, as they can offset the cost of re-searching portions of the game tree.

Chapter 7

Experimental Results

Although the theoretic properties of decision rules and pruning algorithms are interesting on their own, we would also like to write programs which use these algorithms to play games well. In this chapter we first describe the framework we have developed for testing multi-player games and algorithms, and then describe the results of experiments on those games. If the reader is not interested in a high-level description of the implementation, they may skip to section 7.2.

7.1 Experimental Framework

To be able to implement algorithms and games as easily as possible we have developed a set of specialized C++ classes for this task. There are many design decisions involved in writing such a framework, a few of which we discuss here. First and foremost, our framework is structured to make it as easy as possible to share code between different domains.

This is crucial when we want to run a set of similar experiments across many different domains. It is not only tedious to re-write new code for every algorithm in every new domain, but it is also error prone. As the number of algorithms consid-

ered grows, it also becomes difficult to maintain consistent implementations across different domains.

Given our framework, 80-95% of the code used for any game is completely generic. This means that the bulk of the code for running and testing any game only has to be written once. Even more importantly, once an algorithm is working correctly, we don't have to worry about re-implementing it for any new game we come up with, as the current implementation will work fine. It also means, however, that more thought must be put into the code and design to assure that it can robustly handle variations is the number of players in the games or other such things that can vary from domain to domain.

The cost of this design decision is that our code will never be the fastest code possible, as it must be able to handle any game. So, we will not be able to optimize our code to take advantage of some domain-specific features. In a card game, for instance, we can represent a move with an 8-bit value, while a move in Chinese Checkers needs a 16-bit value, and there are other games where we need even larger values to represent moves.

There are four major abstract components that are used for any game. These are a *game*, a *game state*, an *algorithm*, and a *player*. We describe each briefly here.

Game: The Game class is one of the simplest in our architecture. A game deals with the high-level play of a game, managing the addition of players, overall game scoring, the repetition of hands, and other similar activities. The generic im-

plementation of a game will automatically play multiple hands until scoring levels are met. In Hearts, for instance, hands will automatically be played until one player has 100 points, while Cribbage will continue until a player has 121 points.

Game State: A Game State is the description of exactly one state of a game. A game state is modified by applying and undoing moves on that game state, and it also keeps track of whose turn it is in the game. The game state class also provides a list of legal moves for the current player.

Algorithm: The algorithm class is used for implementing decision rules. There is common code for doing iterative deepening searches, so that a simple algorithm implementation will automatically do iterative searches. An algorithm provides many of the same functions that a game state does, passing the calls through to the game state. But, in the process it keeps track of the number of moves applied, the depth of search, and other similar metrics.

The standard algorithm code also provides three ways to limit a search. Any search can be limited by search depth, search time, or the number of nodes expanded. When a search iteration is cut off, the results of the previous iteration are returned.

Because algorithms are generic, we have also defined a Monte-Carlo simulation algorithm that takes as an argument another algorithm, and the number of samples to use. The Monte-Carlo algorithm then runs the algorithm passed to it on the game state as many times as specified, averaging the results together and returning the best overall move.

	Player 1	Player 2	Player 3
1	max^n	max^n	paranoid
2	max^n	paranoid	max^n
3	max^n	paranoid	paranoid
4	paranoid	max^n	max^n
5	paranoid	max^n	paranoid
6	paranoid	paranoid	max^n

Table 7.1. The six possible ways to assign paranoid and max^n player types to a 3-player game.

Player: The role of a player in the game is to define the custom static evaluation function used in the game. The player must also define any heuristics that can be used to prune a search. This makes it easy to implement multiple evaluation functions and test them against each other as separate players in the game.

7.2 General Experimental Setup

To test multi-player games experimentally we have written a game engine as described in the last section that contains a number of algorithms and techniques such as the paranoid algorithm, max^n, zero-window search, transposition tables and monotonic heuristic pruning. New games can easily be defined and plugged into the existing architecture without changing the underlying algorithms.

We first present the general outline of our experiments applicable to all the games, and then we will present the more specific details along with the results.

Our experiments involve 3, 4, and 6-player games while comparing 2 different algorithms. In a 3-player game, there are $2^3 = 8$ different ways we could assign the algorithm used for each player. However, for competitive analysis we are not

interested in games that contain exclusively max^n or exclusively paranoid players, leaving 6 ways to assign each player to an algorithm. These options are shown in Table 7.1. So, we ran our 3-player experiments 6 times, once with each distribution in Table 7.1. For card games, that means that the same hand is played 6 times, once with each possible arrangement of cards. For Chinese Checkers, this varies who goes first, and what player type goes before and after you.

Similarly, in a 4-player game there are $2^4 = 16\text{-}2 = 14$ ways to assign player types, and in a 6-player game there are $2^6 = 64\text{-}2 = 62$ ways to assign player types.

7.3 Chinese Checkers

We ran our experiments in Chinese Checkers on two different versions of the game. Our early experiments were run on a slightly smaller board than is normally used, where each player has 6 pieces instead of 10. Boards this size are commercially available, but as common as the full size board. Our more recent experiments have been performed on the regular board as described in Chapter 2.

7.3.1 Simplified Chinese Checkers

In the version of Chinese Checkers with a smaller board a player will have, on average, about 25 possible moves (in the 3-player game), with over 50 moves available in some cases. The full game often has more than 100 possible moves during the mid-game.

Besides reducing the branching factor, this smaller board also allowed us to create a lookup table of all possible combinations of a single player's pieces on the

board, and an exact evaluation of how many moves it would take to move from that state to the goal assuming no opponent pieces on the board. The table is the solution to the single-agent problem of how to move your pieces across the board as quickly as possible. This makes a useful evaluation for the two-player version of Chinese Checkers. However, as additional players are added to the game, this information becomes less useful, as it doesn't take into account the positions of one's opponents on the board. It does have other uses, however, such as measuring the number of moves a player would need to win at the end of the game.

Because only one player can win the game, Chinese Checkers is a zero-sum, or constant-sum game. However, within the game, the static evaluation is not necessarily constant-sum. Our static evaluation function is based on the distance from each piece to the goal, the proximity of pieces to each other, the number of pieces in the goal area, and the maximum distance from any piece to the goal.

In our 3-player experiments, we played 600 games between the max^n and paranoid algorithms. To avoid having the players repeat the same order of moves in every game, ties near the root of the search tree were broken randomly. We searched the game tree iteratively, searching one level deeper in each successive iteration. These results are originally from [Sturtevant, 2002], so they were not run with speculative pruning.

We report our first results at the top of Table 7.2. We played 600 games, 100 with each possible configuration of players. If the two algorithms played evenly, they would each win 50% of the games, however the paranoid algorithm won over

		Paranoid	Max^n
3-player 250k nodes	games won	60.6%	39.4%
	moves away	3.52	4.92
	search depth	4.9	3.1
4-player 250k nodes	games won	59.3%	40.7%
	moves away	4.23	4.73
	search depth	4.0	3.2
6-player 250k nodes	games won	58.2%	41.8%
	moves away	4.93	5.49
	search depth	4.6	3.85

Table 7.2. 6-piece Chinese Checkers statistics for max^n and paranoid

60% of the games it played.

Another way to evaluate the difference between the algorithms is to look at the state of the board at the end of the game and measure how many moves it would have taken for each player to finish the game from that state. When tabulating these results, we've removed the player who won the game, who was 0 moves away from winning. The paranoid player was, on average, 1.4 moves ahead of the max^n player.

Finally, we can see the effect the paranoid algorithm has on the search depth. The paranoid player could search ahead 4.9 moves on average, while the max^n player could only look ahead 3.1 moves. This matches the theoretical predictions made in section 3.2; Paranoid is able to look ahead about 50% farther than max^n.

We took the same measurements for the 4-player version of Chinese Checkers. With 4 players, there are 14 configurations of players on the board. We played 50 games with each configuration, for a total of 700 games. The results are in the

		Paranoid	Max^n
250k nodes, fixed branching factor	games won	71.4%	28.6%
	moves away	2.47	4.4
	search depth	8.2	5.8
fixed depth search	games won	56.5%	43.5%
	moves away	3.81	4.24

Table 7.3. 3-Player 6-piece Chinese Checkers statistics for max^n and paranoid.

middle of Table 7.2. Paranoid won 59.3% of the games, nearly the same percentage as in the 3-player game. In a 4-player game, paranoid should be able to search 33% farther than max^n, which these results confirm, with paranoid searching, on average, 4-ply into the tree, while max^n was able to search 3.2-ply on average. Finally, the paranoid players that didn't win were 4.23 moves away from winning at the end of the game, while the max^n players were 4.73 moves away. This gave max^n a chance to get closer to the goal state before the game ended.

In the 6-player game, we again see similar results. We played 20 rounds on each of 64 configurations, for 1280 total games. Paranoid won 58.2% of the games, on average 4.93 moves away from the goal state at the end of the game, while max^n was 5.49 moves away on average. In the 6-player game, we expect paranoid to search 20% deeper than max^n, and that is the case, with max^n searching 3.85 moves deep on average and paranoid searching 4.6 moves on average.

Because of this, we conducted another experiment with the 3-player games. In this experiment we again played 600 total games, limiting the branching factor of each algorithm, so that only the six best moves were considered at each branch,

according to the move ordering function. We chose to limit the branching factor to six moves because this allows reasonable depth searches without an unreasonable limitation on the possible moves. If we limited the branching factor to just two moves, for instance, there wouldn't be enough variation in moves to distinguish the two algorithms.

The results from these experiments are found in Table 7.3. Under these conditions, we found that paranoid did even better than max^n, winning 71.4% of all the games even though max^n was able to search much deeper than in previous experiments. The paranoid algorithm could search 8.2 moves deep as opposed to 5.8 for max^n. At the end of the game, paranoid was, on average, only 2.47 moves away from finishing, as opposed to 4.4 for max^n.

Finally, we played the algorithms against each other with a fixed depth search. In this experiment, both algorithms were allowed to search 4-ply into the tree, regardless of node expansions. In these experiments the paranoid algorithm again was able to outperform the maxn algorithm, albeit by lesser margins. Paranoid won 56.5% of the games played, and was 3.81 moves away at the end of the game, as opposed to 4.24 moves for max^n.

These results show that the paranoid algorithm plays better Chinese Checkers both because it can search deeper, and because its analysis produces better play.

7.3.2 Full-Board Chinese Checkers

We did similar experiments with the full game of Chinese Checkers. Because the branching factor is fairly high, at each node we only considered the 10 best

3-Player Full Chinese Checkers

	percent wins	avg. remaining cost	avg. search depth
Speculative Max^n	35%	6.66	6.24
Paranoid	65%	3.54	8.05
Approx. Deep Max^n	48%	5.53	7.56
Paranoid	52%	4.75	8.04

Table 7.4: Max^n variations versus paranoid in Chinese Checkers.

moves according to our move ordering function. This is a reasonable restriction to make, as this ordering heuristic often gives us an optimal ordering when using speculative max^n.

On the larger Chinese Checkers board we ran two experiments. First, we played speculative pruning against paranoid with a 500k node limit, and then we played approximate deep pruning against paranoid, also with a 500k node limit. Approximate deep pruning is not guaranteed to calculate a correct max^n value for the game tree, but it can prune more than speculative max^n. Our program expands 35-45k nodes per second on a 500Mhz G4 processor. The results of these experiments are in Table 7.4. We include the estimated cost for a losing player to get his pieces into the goal state after the game ended, along with the average search depth.

In these experiments paranoid won 65 percent of the games it played against speculative max^n, while only 52 percent of the games it played against approximate deep max^n. On average paranoid could search depth 8.0 against either algorithm, but approximate deep max^n could search 7.56 ply on average, while speculative max^n could only search 6.24 ply into the game tree.

We can see that approximate deep pruning did much better relative to para-

Chinese Checkers expansions at depth 6

Plain Max^n	1.2 million
Speculative Max^n	100k
Approx. Deep Max^n	61k
Paranoid	25k

Table 7.5: Average expansions by various algorithms in Chinese Checkers.

noid than speculative max^n. There are two reasons why this is occurring. First, in Chinese Checkers our node ordering is very good among the 10 best moves, so the chance of incorrectly pruning a node drops. Second, approximate deep pruning has an average search depth of 7.56, while speculative max^n has an average search depth of 6.24. This is important, because at depth 7 a player can look ahead from his first move to his third move. Particularly in the opening and end-game this is very important, allowing the computer to set up more complex jump moves not seen when the average search depth is less than 7. This is much more important that the extra ply that paranoid can search over approximate speculative max^n.

In addition to comparing the performance of paranoid to speculative max^n, we also used Chinese Checkers to measure how effectively pruning each algorithm can prune in practice. We did this by measuring the number of node expansions at depth 6 by both plain max^n with no pruning, speculative max^n, approximate deep max^n and paranoid. The results are in Table 7.5. On average, speculative max^n expanded an order of magnitude fewer nodes than regular max^n, reducing node expansions at depth 6 from 1.2 million to 100k. In many cases speculative max^n was examining the minimum possible game tree. Approximate deep max^n expands even

3-Player Abalone

		percent wins	average score	average depth
fixed depth 4	Speculative Max^n	58%	3.93	-
	Paranoid	42%	4.04	-
500k node limit	Speculative Max^n	36%	3.41	4.01
	Paranoid	63%	4.04	4.79
500k node limit	Approx. Deep Max^n	39%	3.55	4.09
	Paranoid	61%	3.99	4.77

Table 7.6: Max^n variations versus paranoid in Abalone.

fewer nodes, only looking at 61k nodes on average, while paranoid only looks at 25k nodes on average at this depth.

7.4 Abalone

Abalone is similar to Chinese Checkers, in that they are both perfect-information games. However, Abalone is in some ways a much more complicated game than Chinese Checkers. First, the average branching factor is much higher. (There are over 40 possible opening moves in the 3-player game.) But, more importantly, it is much more difficult to order moves in the game. This means that when we attempted to artificially lower the branching factor of the game, we ended up ignoring the best moves in the game, resulting in very poor play. We are not experts in the game of abalone, so we recognize that there may be useful components missing from the static evaluation function we used in the game. Regardless, it is useful to have another data point for comparison.

We ran two different experiments to compare paranoid and max^n in 3-player abalone, a fixed-depth search and a node-bounded search. We performed a fixed-

depth search 4-ply into the game tree; from a player's first move to their second. For the node-bounded search, we bounded node expansions at 500k nodes. Our evaluation function is based on the number of pieces we have, the number of pieces we have pushed off the board, the proximity of our pieces to the center of the board, the proximity of our pieces to the edge of the board, and how well our pieces are grouped together.

The results of these experiments are in Table 7.6. The average score is the number of pieces that a player manages to push off the board during each game. (6 pieces is a win.) At fixed depths max^n won a larger percent of the games played, although its average score was slightly lower than paranoid. This is because the standard deviation of max^n's score, 1.62 is larger than the standard deviation of paranoid's score, 1.38. Max^n seems to be a slightly stronger decision rule than paranoid at fixed depths, which makes sense, as it is easy for two players to gang up on the third in Abalone. But, with a 500k node search limit, paranoid is able to search deeper than max^n, leading to a large improvement in wins. Because our search depth is limited, approximate deep max^n can search marginally deeper than speculative max^n for a small gain in performance, winning 39% of the games as opposed to the 36% that speculative max^n won.

7.5 Perfect-Information Card Games

For the card games Hearts, Spades and Cribbage we deal a single hand and then play that same hand six times in order to vary all combinations of players and

cards. If max^n and paranoid play at equal strength, they will have equal scores after playing the hand 6 times. For both games we used a node limit of 500k nodes per play. These games were played with all cards in each hand face up, allowing all players to see all cards.

For the 3-player games of Hearts and Spades we played 100 hands, 6 times each for a total of 600 games. In Hearts we also run experiments with the 4-player version of the game. For the 4-player game we used 70 hands, played 14 times for each arrangement of players, for 980 total games. Our search was iterative, as in Chinese Checkers. But, since points are only awarded when a trick is taken, we didn't search to depths which ended in the middle of a trick. We used a hand-crafted heuristic to determine the order that nodes were considered within the tree.

In Spades we either prefer to lead high or lead low. If we are following a lead, we first consider the lowest card that will win the trick. If we can't win, we play low. Our static evaluation function is based on the tricks taken, plus an analysis of the card left in your hand, estimating which ones will take tricks and which ones won't.

In Hearts we order our successors so that we will drop the Queen of Spades when we can, and we avoid leading the Ace or King of Spades when the Queen is still out. Our static evaluation function take into account the points taken so far, and a analysis of how many card we have that are expected to take tricks, and how many cards we have that can duck tricks.

	3-Players average score	4-Players average score
Speculative Max^n	8.22	6.33
Paranoid	8.82	7.04
Speculative Max^n	8.11	-
Max^n	8.94	-

Table 7.7: Speculative max^n versus paranoid and max^n in Hearts.

7.5.1 Hearts

We ran several different experiments in Hearts, the first of which was to compare paranoid and speculative max^n. In all our experiments with Hearts, we included shooting the moon as part of the rules, although the computers didn't explicitly try to shoot the moon until they could search the remaining game tree to completion.

We first played paranoid against speculative max^n in three-player Hearts. Before we developed speculative max^n, paranoid was able to play better than max^n, but with the addition of speculative pruning max^n was then able to out-perform paranoid. For more on this, see [Sturtevant, 2002] and [Sturtevant, 2003]. The results of our experiments are in Table 7.7. After 100 games, played once for each combination of players on the table, paranoid averaged 8.82 points per game, while speculative max^n averaged 8.22 points per game. (Lower scores are better.) In the 4-player version of Hearts we got similar results. Over the games speculative max^n averaged 6.33 points per hand, while paranoid averaged 7.04 points per hand.

We measure the standard deviation to see if these differences are statistically significant. It is important, however, that we don't measure the standard deviation of each hand played, as that will just measure the variance in the cards dealt. Instead,

we need to look at all 6 games played on a particular deal of cards in the three-player version of hearts and the 14 games played per deal in four-player hearts. When we do this, we see that in 3-player hearts max^n's score had a standard deviation of 1.84, and paranoid had a standard deviation of 1.87. In four-player hearts max^n had a standard deviation 1.03 points per hand while paranoid had a standard deviation of 1.27 points per hand. The means that there is a greater statistical separation on scores in the four-player version, and that max^n is more likely to shoot the moon that paranoid, leading to a larger standard variation on paranoid's score.

In addition to these experiments, we also did a similar experiment to compare max^n with a tie-breaking rule to speculative max^n. The goal of this experiment was to see if a search with no pruning but a sophisticated tie-breaking rule would be able to outperform a deeper search with a less sophisticated tie-breaking rule. For the plain max^n implementation we broke ties to minimize the score of the player at the root of the tree, as this was experimentally seen to be quite effective of improving the play of max^n. This is because it will allow us to avoid situations like those we saw in Figure 4.9. The results from these experiments are also in Table 7.7. It ended up that in our actual experiments the added search depth was much better than a tie breaking rule, with speculative max^n averaging 8.11 points per hand, while max^n with a good tie-breaking rule averaged 8.94 points per hand.

7.5.2 Spades

We ran experiments in Spades similar to Hearts, but in Spades we are only interested in the three-player version of the game, because the four-player version is

		average score
3-players	Speculative Max^n	5.65
	Paranoid	5.68

Table 7.8: Speculative max^n versus paranoid in Spades.

played in two teams. We played 100 games with all possible arrangements of players at the table, using a 500k node search limit. Table 7.8 shows that the difference between the two decision rules in Spades is negligible. This is despite the fact that, on average, paranoid can search about 10 ply deeper than speculative max^n. It seems that this occurs because the general strategy is always to take tricks with high cards. We suspect that Spades, and the similar games like 8-5-3 are interesting because of the other interactions in the game, such as the passing of cards or the bidding, which we didn't model in these experiments.

7.5.3 Cribbage

In Cribbage we can search the entire game tree quite quickly, so the only comparison that needs to be made is between the max^n decision rule and the paranoid decision rule. We ran two sets of experiments. In the first set of experiments we just played out single hands and compared the scores of all players. In the second set of experiments we played out full games to 121 points. The results are in Table 7.9 and are similar for both experiments. In an average hand max^n got 6.89 points

		average hand score	average game score
3-players	Speculative Max^n	6.89	96.1
	Paranoid	6.55	91.4

Table 7.9: Speculative max^n versus paranoid in Cribbage.

			average score
3 players	40 models, 0.5 sec./model	Speculative Max^n	6.91
		Paranoid	10.32
	20 models, 1.0 sec/model	Speculative Max^n	6.43
		Paranoid	10.92
		Approx. Deep Max^n	7.25
		Paranoid	9.98
4 players	20 models, 1.0 sec/model	Speculative Max^n	5.34
		Paranoid	8.81

Table 7.10: Speculative max^n versus paranoid in Hearts.

while paranoid only got 6.55 points, with a standard deviation of 1.73. In full games, max^n averaged 96.1 points, while paranoid averaged 91.4 points, with a standard deviation of 12.97 for max^n and 12.14 for paranoid. On the full game, each algorithm averaged about 13.95 times their score in the partial game. These are small differences, but in the two-player game most games are decided by six points or less [Colvert, 1997], so this is actually a reasonably large gap.

7.6 Imperfect Information Card Games

Because we really play most card games as imperfect information games, in addition to our experiments with the perfect information variants of these games, we also have done experiments for the real version, using Monte-Carlo simulations to make our moves.

7.6.1 Hearts

We ran Monte-Carlo experiments in Hearts with two variations in sample size. In the first set of experiments we gave the computer 0.5 seconds to analyze each of 40 models, and in the second experiment we gave the computer 1 second

		average score	search depth
20 models 1 sec/model	Speculative Max^n	5.81	11.95
	Paranoid	5.52	13.04

Table 7.11: Max^n versus paranoid in Spades.

to analyze 20 models. We played 50 three-player games once for each possible ordering of players at the table, for a total of 300 games. The results from Hearts are in Table 7.8. When we used 40 models, speculative max^n averaged 6.91 points per hand, while paranoid averaged 10.32 points per hand. When we used 20 models, speculative max^n averaged 6.43 points, while paranoid averaged 10.92 points per hand. In the same table, we have results from playing approximate deep max^n against paranoid, where approximate deep max^n averaged 7.25 points a game to paranoid's 9.98 points. This is good, but it is not as good as speculative max^n.

While speculative max^n did slightly better than paranoid in perfect information games, it did much better in imperfect information games. When we analyze the hands played we don't see paranoid repeatedly making obviously bad moves. Instead it just seems that speculative max^n is consistently able to do analysis that gives it a slight advantage over paranoid, and in the long term this pays off well for speculative max^n.

We got similar results for the 4-player version of Hearts. These results are in the bottom of Table 7.10. In the 4-player game paranoid averaged 8.81 points per game, while speculative max^n averaged 5.34 points per game.

		average hand score	average game score
30 models	Speculative Max^n	7.16	113.74
	Paranoid	7.12	114.07

Table 7.12: Max^n versus paranoid in Cribbage.

7.6.2 Spades

In Spades running with imperfect information we saw our first difference between max^n and paranoid. These results are in Table 7.11. We gave each player 20 seconds to analyze 20 models on a G4 500Mhz machine before making a play. Max^n averaged 5.81 points per hand, while paranoid averaged 5.52, with a standard deviation of 0.36. Given the search limits, max^n had an average search depth of 11.95, while paranoid had an average search depth of 13.04.

7.6.3 Cribbage

We ran Monte-Carlo experiments in Cribbage for both full games and single hand play. In both experiments we used 30 samples. Table 7.12 shows the results of these experiments. While there was a small advantage for max^n in perfect-information cribbage games, that advantage becomes statistically insignificant in the true imperfect information game, with max^n averaging 7.16 points per hand, while paranoid averaged 7.12 points. On the larger game there was a small advantage for paranoid, which averaged 114.07 points per game versus 113.74 for max^n.

Because Cribbage has a relatively small search tree in the play, it is a game that might benefit from other imperfect-information techniques. For instance, instead of just generating Monte-Carlo models, it should be feasible in some cases to

calculate the exact probabilities of our opponent holding certain cards. Additionally, it may be worthwhile to do some level of meta-reasoning, where we model our opponents model of ourselves. In the perfect information version of Cribbage we know exactly what cards our opponents hold, so we know exactly how to get points or avoid taking points. In each Monte-Carlo model we use we will have the same perfect information. This means that we will never try to deceive our opponents, because we always expect them to know the cards we have. This is currently too computationally expensive to consider in other games, but Cribbage is a good candidate for further research on this topic, both for two-player and multi-player games.

7.7 Discussion

We have presented a lot of results here, and it can be somewhat difficult to digest and understand them all. While each result may bring to light particular details that may need to be explored in a particular domain, we want to look to broader trends in the data.

First, a larger question that we would like to answer is which algorithm is preferable, standard max^n, paranoid, or approximate deep max^n. We can't answer the question decisively, but we do have some intuition from these results. For the paranoid algorithm, it seems that as search depth increases, the benefit of a deeper search lessens. This is directly attributable to the fact that the deeper that paranoid searches, the more ways it will discover that its opponents are able to collude against it. Since true collusion is not taking place in any of our experiments, this is a flawed strategy.

But, in games like Chinese Checkers or Abalone where we are not searching very deep, our opponents have few opportunities to collude. Thus, paranoid provides a reasonably defensive line of play coupled with deeper search than is found in max^n. We expect that if we were able to search 10-20 ply into a Chinese Checkers or Abalone game that the advantage of paranoid would be greatly diminished if not lost completely.

Despite this, we have not been able to show that paranoid actually does worse as it searches deeper. We performed many different experiments where we varied the search depth of paranoid, and we were never able to find paranoid doing worse as it searched deeper. Instead it just seems that max^n takes better advantage of the additional search in a game tree.

With regard to pruning algorithms for max^n, it seems that approximate deep pruning has only limited potential, as it never outperformed both paranoid and regular max^n. The domains where it is most likely to be useful are those where we have a high-quality ordering function, which will reduce the errors made by incorrect pruning, along with those domains for which there is a large benefit to an extra ply or two of search, the benefit of which may outweigh the cost of the mistakes we make in our search.

However, in general it seems to make more sense to use an algorithm that has guaranteed theoretical properties such as max^n or paranoid than to use an algorithm that has no guarantees on the validity or correctness of the result it calculates.

7.8 State of the Art Play

7.8.1 Hearts

In addition to testing between different algorithms, we have also worked on optimizing a hearts program to be competitively stronger than any other program that bas been written. It is infeasible to compare against every shipping program, but we were able to run extensive tests against the commercially shipping program Hearts Deluxe™ (HD), by Freeverse software.

The computer program used to play Hearts for these experiments was taken directly from game framework, without too much specific optimization for the game of Hearts. We did, however, re-write our static evaluation function for these tests. We incorporated the same features described in the experimental results, along with a few special rules for when and how the Queen of Spades should be played. For instance, we reduce a player's static evaluation more than usual for taking the Queen of Spades when they had other spades left in their hand.

We also varied the number of models used for analysis during the game. For the first 10 tricks of the game we allowed the computer to search 280k nodes over all models. We began with 40 models and gradually decreased this to 13 models until there are 3 tricks left in the game, at which point we always search the game tree to completion on 30 models. Our computer program takes no more than a few seconds to make a move, which is reasonable for normal play. In practice we could probably extend this end-game analysis farther out, but we were primarily concerned with being able to quickly run these experiments, since we had to start and end each hand

and game manually.

Our program exhibits reasonable planning near the end of the game. In many situations, for instance, it will take the tricks it can't avoid first, before giving the lead to another player. This behavior is something that expert card players will do, but that computer programs rarely do. Most computer programs and beginning players tend to play their low cards too early in the game, trapping them with their high cards at the end of the game. Since most points come out in the later stages of the game, this is an expensive approach.

Our program gives some merit to preventing its opponents from shooting the moon in its evaluation function, but it doesn't actively try to model if an opponent is trying to do so. Instead, it gives a bonus to the player if they are able to split the points in the game. Similarly, the computer does not explicitly plan to shoot the moon from early in the game, but will do so later in the game if it finds it can.

One deficiency we see in our program is that it doesn't have a model of what information its plays convey to the other players in the game. So, our player is perfectly happy to lead the Ace of Spades when he also holds the Queen, something humans will usually avoid, because of the information it reveals to the other players. But, the HD computer players don't seem to take advantage of such information. This means that this type of mistake will not hurt us when playing another computer, while it can hurt us when playing a human.

The authors of Hearts Deluxe were kind enough to not only provide us with the source code to their computer players, they also worked to build a plug-in sys-

	average score/game	average score/hand
Our Program	55.8	5.16
Hearts Deluxe™	75.1	6.98

Table 7.13: Our Hearts program versus the commercial Hearts Deluxe program

tem that could be used to plug-in and test other players against their own. In addition, Apple Computer loaned us one of their top-of-the-line 1.42 GHz dual-processor machines for the sake of running our experiments. Because each Monte-Carlo experiment is independent, it is quite easy to parallelize a program for multiple processors.

We report the result of our experiments against their computer players in Table 7.13. In these experiments our computer program played against 3 of the HD players. The HD players can be set to different skill levels, so we chose to set them to the highest skill available, with average aggression. (Other options are aggressive and conservative.) The rule variations we used included -26 for shooting the moon, but we didn't enable card passing. Each game was played until one player had 100 or more points at the end of a hand. We played 90 games total, and our program averaged 55.8 points per round, while the other players collectively averaged 75.1 points. In these rounds we played a total of 984 hands. Our program averaged 5.16 points per hand, while our opponents collectively averaged 6.98 points per hand.

There are a few areas we see that our program could improve. First, our static evaluation function could be improved. Our program tends to play the Ace and King of Spades earlier than necessary, which seems to lead to taking the Queen more of-

ten that it might have to. Also, our program also tends to lead its high diamonds and clubs early in the game, when it sometimes should be playing spades to force out the queen. Our evaluation is completely hand tuned right now, but there is no reason that a better evaluation function couldn't be developed through a variety of learning methods.

Besides Hearts Deluxe, we have not tested our program against other existing computer players. We have, however, asked the authors of some of the other leading Hearts programs how they build their computer players. The only author to respond to our query was Vytas Kliorys, author of Turbo Hearts, a popular program for Microsoft Windows. He indicated that his program uses a weighted rule-based system to decide how to play, and that it doesn't look ahead farther than the end of the current trick and the first card of the next trick.

If these programs are indicative of the state of the commercial industry right now, which we suspect, there is no reason to believe that our program is playing at least as well, if not better than any program available.

7.8.2 Chinese Checkers

We are not aware of many programs that play Chinese Checkers. The ACM has held competitions for 2-player Chinese Checkers programs in Hong Kong recently, but it seems that almost no one has done work in variations of the game for three or more players. Thus, since we have what is essentially a perfect closing book for Chinese Checkers on the smaller board, we feel that it is not unreasonable to claim that our program which plays on this board is the best program in existence

for this domain. On the larger board we know there are improvements to be made, including the analysis of potential opening and closing books along with faster move generation.

Chapter 8

Contributions, Conclusions, and Future Work

8.1 Overview of Results

The work done for this thesis has greatly expanded our knowledge of the algorithms and techniques that can be used to play multi-player games. It has done this through the analysis of decision rules, the development of pruning algorithms, the analysis of other standard two-player game techniques, and experimental results. The work on decision rules has developed the properties of the max^n and paranoid algorithms, showing that in practice they both have deficiencies in play in practice. In addition, we have developed and analyzed pruning algorithms for these decision rules, including speculative max^n, the first pruning algorithm that provides considerable pruning gains for games such as Chinese Checkers which previously could not be pruned by any known methods.

The pruning algorithms we have developed include multi-player branch-and-bound pruning, alpha-beta branch-and-bound pruning, last-branch pruning, and speculative pruning. Alpha-beta branch-and-bound pruning combines previous work on shallow pruning [Korf, 1991] with monotonic heuristics to be able to prune

game trees well. Alpha-beta branch-and-bound pruning however, is not effective in many common multi-player games, being reduced simply to branch-and-bound pruning. But, last-branch and speculative pruning are the first algorithms that are expected to be effective in constant-sum multi-player game trees, together reducing the asymptotic branching factor of a n-player game from b to $b^{d\cdot(n-1)/n}$ as b grows large. Not only can these algorithms be applied to any constant-sum game, they are also expected to provide asymptotic reductions in b in the average case, which previous techniques such as shallow pruning did not.

We have also shown how tie-breaking is a pervasive issue in $\max^n$ game trees. If we incorrectly model how our opponents break ties within the game tree, we cannot bound the error that will result in our calculation of the $\max^n$ value of the game tree. Tie-breaking also means that we cannot use techniques like zero-window search in a multi-player game, and iterative deepening search is weaker in multi-player games because of tie breaking when compared to the results possible in two-player games. Tie-breaking is also important in end-game databases, as any database calculated ahead of time must implicitly use a pre-calculated tie-breaking rule.

Besides these theoretical results, we have used these techniques to build and test multi-player game programs in many domains, including state-of-the-art programs in Hearts and Chinese Checkers, where we have shown that our program does much better than the existing program Hearts Deluxe, averaging almost 20 points better per round then the regular computer players in that game.

8.2 Future Work

There are still areas that need further research to fully understand the challenges involved in multi-player games. One of these areas is in the area of average-case analysis of our pruning algorithms. While we have a vague idea that speculative max^n will be effective in the average case, and alpha-beta branch-and-bound will not be as effective in the average case, concrete models for analysis need to be developed to solve and test for the average case.

Solving the average case models will also be an important step in extending work in areas such as node-ordering, as we will be able to verify the relative effectiveness of node ordering algorithms based on the best-case and average-case performance models.

Also, approximate deep max^n pruning seems to have some promise in some domains. The correctness of approximate deep max^n will also vary depending on the effectiveness of our node-ordering algorithms. More extensive tests need to be done to understand exactly when approximate deep max^n is a worthwhile algorithm to use.

In the area of writing state-of-the-art programs for playing games, there are a few areas that need to be considered. The first area is in imperfect information. We have assumed that Monte-Carlo sampling is the best way to deal with imperfect information. This is a reasonable first-order approximation, but some recent work in Bridge [Ginsberg, 2001] has moved away from Monte-Carlo sampling in later stages of the game. Similar strategies may be needed in multi-player games. Regardless

of if this is the case, our work still stands for imperfect information games.

Another area we have not looked into are learning strategies for static evaluation functions, and more specific opponent modelling strategies. Max^n is relatively flexible in its ability to model its opponents, and we should be able to use max^n to easily implement our own preferences for competing against multiple opponents.

8.3 Max^n versus Paranoid

Minimax with alpha-beta pruning has generally dominated other two-player decision rules and algorithms, but there is no clear dominant algorithm in multi-player games at this time. However, our experience in this field has led us to postulate that the paranoid algorithm will generally do well in games where the opponents cannot collaborate together effectively. This may be a result of minimal search depth or simply a property of a game. In card games, for instance, it is generally quite easy to collaborate, and in these domains the paranoid algorithm has a large search depth advantage over max^n, but no performance advantage.

8.4 Conclusion

Multi-player games are fundamentally different from two-player games. They force us to consider the issue of opponent strategy, partially in the form of tie-breaking issues within the game tree, while such issues have generally been ignored by Artificial Intelligence researchers in two-player games. In addition, the complexity of an additional player in a game increases not only the amount of analysis needed for multi-player games, but also decreases the amount of pruning possible.

In this thesis we have presented a variety of new techniques to deal with this issues, perhaps most importantly developing the first widely-effective pruning techniques for multi-player game search. There are still many challenging and unanswered questions in this area, and we look forward to the research that will answer these challenges.

References

Anderson, S., Convergence of the Video Game Industry and CGI Visual Effects Industry, UCLA Association for Careers in Technology Information Session, April, 2003.

Baird, K., Hollywood finds video game tie-ins aren't kid stuff, Sacramento Bee, Feb 26, 2003.

Billings, D., Davidson, A, Schaeffer, J., Szafron, D., The Challenge of Poker, Artificial Intelligence Journal, vol 134(1-2), pp 201-240, 2002.

Billings, D., Peña, L., Schaeffer, J, Szafron, D., Using Probabilistic Knowledge and Simulation to Play Poker, Proceedings, AAAI-99, Orlando, Fl, 697-703, 1999.

Buro, M. The Othello Match of the Year: Takeshi Murakami vs. Logistello, Journal of the International Computer Chess Association 20(3): 189-193, 1997.

Colvert, D, Play Winning Cribbage, Starr Studios, 1997.

Epstein, Susan L., Game Playing: The Next Moves, AAAI-99 Invited Speaker.

Gibson, Walter B., Hoyle's Modern Encyclopedia of Card Games; Rules of All the Basic Games and Popular Variations, Doubleday, 1974.

Ginsberg, M, GIB: Imperfect Information in a Computationally Challenging Game, Journal of Artificial Intelligence Research 14, 303-358, 2001.

Ginsberg, M, GIB: Steps Toward an Expert-Level Bridge-Playing Program, Proceedings, IJCAI-99, 584-589.

Ginsberg, M, How Computers Will Play Bridge, The Bridge World, 1996.

Ginsberg, M, Partition Search, Proceedings AAAI-96, Portland, OR, 228-33.

Hoyle, E., and Frey, R.L., Morehead, A.L., and Mott-Smith, G, The Authoritative Guide to the Official Rules of All Popular Games of Skill and Chance, Doubleday, 1991.

Jones, A.J., Game Theory: Mathematical Models of Conflict, West Sussex, England: Ellis Horwood, 1980.

Knuth, D.E., and Moore, R.W., An analysis of alpha-beta pruning, Artificial Intelligence, vol. 6 no. 4, 1975, 293-326.

Korf, R.E. Multiplayer alpha-beta pruning. Artificial Intelligence, vol. 48 no. 1, 1991, 99-111.

Luce, R. and Raiffa, H., Games and Decisions, New York: John Wiley & Sons, 1957.

Luckhardt, C.A., and Irani, K.B., An algorithmic solution of N-person games, Proceedings AAAI-86, Philadelphia, PA, 158-162.

Mutchler, D, The Multi-Player Version of Minimax Displays Game-Tree Pathology, Artificial Intelligence 64 (2): 323-336, 1993.

Nash, J. F., Non-cooperative games, Annals of Mathematics, 54, 286-295, 1951.

Newell, A, Shaw, J, and Simon, H, Chess Playing Programs and the Problem of Complexity, IBM Journal of Research and Development 4, 2 (1958), 320-335. Also in E. Feigenbaum and J. Feldman (eds.), Computers and Thought, 1963, 39-70.

Parlett, David, A Dictionary of Card Games, Oxford Unviersity Press, 1992

Pearl, J., *Heuristics* Addison-Wesley, Reading, MA 1984.

Prieditis A.E., Fletcher, E., Two-agent IDA*, Journal of Experimental and Theoretical Aritificial Intelligence, v 10, 1998, pp 451-484.

Russell, S., Norvig, P., Artificial Intelligence: A Modern Approach, Prentice-Hall Inc., 1995.

Samuel, A. L., Some studies in machine learning using the game of checkers, IBM Journal of Research and Development, vol 3, 1959, 210-229.

Samuel, A. L., Some studies in machine learning using the game of checkers II-Recent progress, IBM Journal of Research and Development, vol 11, 1967, 601-617.

Schaeffer, J, A Gamut of Games, AI Magazine, Vol 22, No 3, Fall 2001, 29-46.

Schaeffer, J., Culberson, J., Treloar, N., and Knight, B., A world championship caliber checkers program, Artificial Intelligence, vol. 53, 1992, 273-289.

Schaeffer, J., The History Heuristic and Alpha-Beta Search Enhancements in Practice, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol 11, No 11, November, 1989, 1203-1212.

Shannon, C.E., Programming a computer for playing chess, Philosophical Magazine, 41(4), 256-275, 1950.

Sheppard, B, World-championship-caliber Scrabble, Artificial Intelligence, v.134 n.1-2, 241-275, January 2002.

Smith, Ian, (President of freeverse software and author of Hearts Deluxe™), Personal Communication, 1999.

Sturtevant, N., A Comparison of Algorithms for Multi-Player Games, *Proceedings of the 3rd International Conference on Computers and Games*, 2002.

Sturtevant, N., Last-Branch and Speculative Pruning Algorithms for Max^n, Proceedings IJCAI-2003, Acapulco, Mexico.

Sturtevant, N., Korf, R., On Pruning Techniques for Multi-Player Games, Proceedings, AAAI-2000, Austin, Tx, pp 201-207.

Tesauro, G. and Sejnowski, T.J., A parallel network that learns to play backgammon, Artificial Intelligence, vol 39, 1989, 357-390.

Tesauro, G. Temporal-Difference Learning and td-gammon, Communiciations of the ACM, 38(3): 58-68, 1995.

Turing, A., Strachey, C., Batest, M., and Bowden, B., Digital Computers Applied to Games, in Bowden, B., editor, *Faster Than Thought*, 286-310, 1953.