To the Graduate Council:

I am submitting herewith a thesis written by P. Susan McFaddin Williams entitled "Search Techniques for the Game of Double-Dummy Bridge." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Dr. David Mutchler, Major Professor

We have read this thesis
and recommend its acceptance:

Joan R. S. Blair

Bruce W. Char

Accepted for the Council:

Vice Provost
and Dean of the Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature P. Susan McFaddin Williams

Date December 1, 1989

SEARCH TECHNIQUES FOR THE GAME OF DOUBLE-DUMMY BRIDGE

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

P. Susan McFaddin Williams

December 1989

ABSTRACT

The game of Double-Dummy Bridge can be viewed as a two-team, perfect information game. In theory, the solution to this game can be found by exhaustively examining every possible play of the game. In practice, a computer is not able to solve such a problem due to time constraints. We develop, compare, and analyze techniques implemented in order to solve a Double-Dummy Bridge game without exhaustively examining every possible line of play of the game. Analysis of the techniques allows us to distinguish the performance of each of the techniques as applied to the Double-Dummy Bridge game. The techniques implemented here are valuable steps toward realizing a perfect computer Double-Dummy Bridge player.

Contents

1 INTRODUCTION 1

2 PRELIMINARY CONCEPTS 3

2.1 The Game of Double-Dummy Bridge 3

2.2 Minimax Search Algorithm 5

2.3 Searching and Pruning 8

3 EXPERIMENTAL DESIGN 12

4 DISCUSSION OF THE TECHNIQUES 28

4.1 Minimax Search Technique 28

4.2 Alpha-Beta Pruning Technique 35

4.3 Good-Ordering Technique 40

4.4 Strict Bucketing Technique 53

4.5 Internal Cutoffs Technique 58

4.6 Deferred Branching Technique 70

4.6.1 General Idea of Deferred Branching 70

4.6.2 A Possible Problem with Deferred Branching 74

4.6.3 Implementation 78

4.6.4 Analysis 82

4.7 Hashing Technique 88

5 SUMMARY 102

6 FUTURE IDEAS AND EXTENSIONS OF THIS WORK 105

BIBLIOGRAPHY 108

APPENDIXES 111

A Calculation of Lower Bound on Time It Takes to Solve a Full 52 Card Double-Dummy Bridge Game 112

B Rules for Intelligently Ordering the Nodes used by the Alpha-Beta with Good Ordering of Nodes Search Technique 114

C Pseudo Code for Marking Buckets in the Deferred Branching Search Technique 117

VITA 121

List of Tables

3.1 Frontier Development Table for Minimax Search Strategy 16

3.2 Approximate CPU Time per Node for the Search Techniques . . . 19

4.1 Frontier Development Table for Minimax Search Strategy 33

4.2 Frontier Development Table for Alpha-Beta Search Strategy . . . 39

4.3 Game Tree Complexity Comparison for Minimax vs. Alpha-Beta Search Techniques 43

4.4 Frontier Development Table for Alpha-Beta with Good Ordering of Nodes Search Strategy 48

4.5 Game Tree Complexity Comparison for Alpha-Beta with Random Ordering of Nodes vs. Alpha-Beta with Good Order of Nodes . . 51

4.6 Game Tree Complexity Comparison for Minimax vs. Alpha-Beta With Good Ordering of Nodes Search Techniques 52

4.7 Game Tree Complexity Comparison for Strict Bucketing vs. Alpha-Beta With Good Ordering of Nodes Search Techniques 59

4.8 Frontier Development Table for Strict Bucketing Search Strategy . 62

4.9 Game Tree Complexity Comparison for Internal Cutoffs vs. Strict Bucketing 67

4.10 Frontier Development Table for Internal Cutoffs Pruning Strategy 67

4.11 Frontier Development Table for Deferred Branching Pruning Strategy 86

4.12 Game Tree Complexity Comparison for Deferred Branching vs. Internal Cutoffs 87
4.13 Frontier Development Table for Hashing Technique 98
4.14 Game Tree Complexity Comparison for Hashing Technique vs. Deferred Branching 101

5.1 Comparison of Confidence Intervals for the Number of Nodes Expanded for Each of the Techniques 103
5.2 Game Tree Complexity Comparison for Accumulation of all Techniques vs. Basis Minimax Search Technique 103
5.3 Summary Information for the Number of Nodes Expanded in Double-Dummy Bridge Games of 16 Cards 104

List of Figures

2.1 Example Game Tree I 6

2.2 Example Game Tree II 9

3.1 Sample Legal One Suit Hand with 12 Cards 14

3.2 Sample Legal Two Suit Hand with 16 Cards 14

3.3 Sample Legal Three Suit Hand with 12 Cards 15

3.4 Sample Legal Four Suit Hand with 52 Cards 15

3.5 Game of Four Suits with Eight Cards Corresponding to the Game Tree in Figure 3.7 25

3.6 Game of Four Suits with Eight Cards Corresponding to the Game Tree in Figure 3.8 25

3.7 Game Tree of Four Suits with Eight Cards Corresponding to the Game Tree in Figure 3.5 26

3.8 Game Tree of Four Suits with Eight Cards Corresponding to the Game Tree in Figure 3.6 27

4.1 Game of Two Suits with Eight Cards Corresponding to the Game Tree in Figure 4.2 30

4.2 Minimax Game Tree Corresponding to Figure 4.1 31

4.3 Minimax Search Strategy - Average Number of Nodes Expanded . 34

4.4 Game of Four Suits with Eight Cards Corresponding to the Game Tree in Figure 4.5 36

4.5 Alpha-Beta Search Tree Corresponding to Figure 4.4 37
4.6 Alpha-Beta (random) vs. Minimax, 1 suit games 41
4.7 Alpha-Beta (random) vs. Minimax, 2 suit games 41
4.8 Alpha-Beta (random) vs. Minimax, 3 suit games 42
4.9 Alpha-Beta (random) vs. Minimax, 4 suit games 42
4.10 Example of Alpha-Beta (random) Search Tree with No Prunes . . 45
4.11 Example of Good Ordering Search Tree with Several Prunes . . . 46
4.12 Alpha-Beta (good) vs. Previous Techniques, 1 suit games 49
4.13 Alpha-Beta (good) vs. Previous Techniques, 2 suit games 49
4.14 Alpha-Beta (good) vs. Previous Techniques, 3 suit games 50
4.15 Alpha-Beta (good) vs. Previous Techniques, 4 suit games 50
4.16 Game of Two Suits with Eight Cards Corresponding to the Game Trees in Figure 4.17 and Figure 4.18 55
4.17 Good Ordering Search Tree of Figure 4.16 56
4.18 Strict Bucketing Search Tree of Figure 4.16 57
4.19 Strict Bucketing vs. Previous Techniques, 1 suit games 60
4.20 Strict Bucketing vs. Previous Techniques, 2 suit games 60
4.21 Strict Bucketing vs. Previous Techniques, 3 suit games 61
4.22 Strict Bucketing vs. Previous Techniques, 4 suit games 61
4.23 Game of Four Suits with Twelve Cards Corresponding to the Game Tree in Figure 4.24 and Figure 4.25 63
4.24 Search Tree of Figure 4.23 after Addition of Strict Bucketing Technique 64
4.25 Search Tree of Figure 4.23 after Addition of Internal Cutoffs Technique 65
4.26 Internal Cutoffs vs. Previous Techniques, 1 suit games 68
4.27 Internal Cutoffs vs. Previous Techniques, 2 suit games 68

4.28 Internal Cutoffs vs. Previous Techniques, 3 suit games 69
4.29 Internal Cutoffs vs. Previous Techniques, 4 suit games 69
4.30 Game Used for Examination of the Possible Problem with Deferred Branching 74
4.31 Search Tree of Figure 4.30 using Alpha-Beta (good) and Strict Bucketing Techniques 75
4.32 Search Tree of Figure 4.30 using Alpha-Beta (good) and Deferred Branching Techniques 76
4.33 Game of Two Suits with Sixteen Cards 79
4.34 Game of Four Suits with Twelve Cards Corresponding to the Search Tree in Figure 4.35 and Figure 4.36 83
4.35 Search Tree Corresponding to Figure 4.34 before Addition of Deferred Branching Technique 84
4.36 Search Tree Corresponding to Figure 4.34 after Addition of Deferred Branching Technique 85
4.37 Deferred Branching vs. Previous Techniques, 1 suit games 89
4.38 Deferred Branching vs. Previous Techniques, 2 suit games 89
4.39 Deferred Branching vs. Previous Techniques, 3 suit games 90
4.40 Deferred Branching vs. Previous Techniques, 4 suit games 90
4.41 Hashing Payoffs 94
4.42 Game of Four Suits with Twelve Cards Corresponding to the Search Tree in Figure 4.43 and Figure 4.44 95
4.43 Search Tree for Figure 4.42 before the Addition of Hashing Technique 96
4.44 Search Tree for Figure 4.42 after the Addition of Hashing Technique 97
4.45 Hashing vs. Previous Techniques, 1 suit games 99
4.46 Hashing vs. Previous Techniques, 2 suit games 99
4.47 Hashing vs. Previous Techniques, 3 suit games 100

4.48 Hashing vs. Previous Techniques, 4 suit games 100

Chapter 1

INTRODUCTION

The game of Double-Dummy Bridge, to be discussed in detail in section 2.1, can be viewed as a two-team, perfect information game. The problem is to obtain the outcome of the game when both teams are playing optimally. In theory, this problem can be solved by examining every possible play of the game. In practice, a computer is not able to solve such a problem due to time constraints. In fact, the exhaustive search of every possible play of the game would take over three million years (see Appendix A). In this paper, we develop, compare, and analyze techniques implemented in order to solve a Double-Dummy Bridge game without exhaustively examining every possible line of play of the game. Analysis of the techniques allows us to distinguish the performance of each of the techniques as applied to the Double-Dummy Bridge game.

Berklekamp in [1] attempts to solve this problem with the use of heuristics (rules of thumb), thus his method seems to allow the possibility of occasional errors. The work of this thesis does not include heuristics in order to eliminate the possibility of an error. Berklekamp relies heavily on the specifics of the game of bridge whereas the work of this thesis contains more general techniques which could be applicable to many games. There are published works that deal with other aspects of Contract Bridge, but none for the Double-Dummy game. These

include [2, 4, 5, 7, 11, 14, 16, 18, 19, 20, 21, 22, 23].

Chapter 2 introduces the game of Double-Dummy Bridge and the preliminary ideas needed to understand this work. Chapter 3 explains the experimental design of this project. Chapter 4 discusses and analyzes the techniques of this thesis in detail. Chapter 5 provides summary information and Chapter 6 discusses several possible extensions of this work.

The techniques developed and analyzed within this thesis result in limiting the exhaustive search of the possible lines of play such that double-dummy games of up to 32 cards can successfully be solved within a reasonable amount of time. Previous to the implementation of these techniques, only games with up to 16 cards can successfully be solved within a reasonable amount of time. A large amount of data was collected which enables analysis and characterization of various types of techniques. The techniques implemented here are valuable steps toward realizing a perfect double-dummy bridge player.

In summary, this thesis makes several contributions:

- Provides analysis of the effects of existing techniques on the game of Double-Dummy Bridge and provides comparison of the effects to predictions from models previously studied.
- Invention of a technique called Deferred Branching.
- Provides a large amount of code, 1850 *Lisp* lines and 1200 *shell* and *awk* lines, as a tool toward the goal of the world's best computer bridge player.

Chapter 2

PRELIMINARY CONCEPTS

2.1 The Game of Double-Dummy Bridge

Because the double-dummy bridge game is developed from the card game of Contract Bridge, a simplistic discussion of Contract Bridge is needed in order to explain the rules of double-dummy bridge. In a game of Contract Bridge there are four players, North, South, East, and West. A deck of 52 cards is dealt to the players, each player receiving 13 cards. After the bidding, which is not relevant to this work, the play begins with the player to the dealer's left playing a card, followed by the other three players playing a card. All players must follow suit if they have any card of the suit led by the first player, else a player must discard from any of his suits. When all four players have played a card, a trick has been played. The winner of the trick is the player who played the highest card of the lead suit. This player is then designated as having the lead, thus plays the first card of the next trick. This continues until all 52 of the cards have been played, that is, until 13 tricks have been played.

Bridge is a team game in which players North and South (N/S), and East and West (E/W), play as a team. The number of tricks won by East is added to the number of tricks won by West to get a E/W team score, and similarly for

the N/S team. So the team players work interactively to maximize the number of tricks their team wins.

There are several features of the Contract Bridge game that are not relevant to double-dummy bridge. One of those features is the bidding phase in which a contract is developed. The contract states the total number of tricks which must be won by the team who wins the contract. The contract also states which, if any, of the four suits, Clubs (♣), Diamonds (♢), Hearts (♡), or Spades (♠), is the trump suit. The trump suit is also a feature that is not relevant to this thesis as this thesis deals only with no-trump double-dummy games. One last feature of Contract Bridge not relevant to the double-dummy game is the scoring. An elaborate set of rules define the number of points awarded after a hand has been played. The details of the scoring rules are not necessary to understand the double-dummy bridge game. For a more detailed introduction to Contract Bridge, explore for example "Bridge For Everyone" by Shelly DeSatnick [3] or any of the many introductory bridge books.

A double-dummy bridge game is a puzzle for one person to solve. The deck of 52 cards is dealt into four hands, North, South, East, and West. The game begins with a card being led from West's hand. The player can see the cards in everyone's hand, thus double-dummy bridge is a perfect information game. The task is to figure out the play of the game as if it were a game of Contract Bridge. A solution to the puzzle is a discussion of the maximum number of tricks that the N/S team can take against the adversary E/W team and how the N/S team should play in order to take these tricks. The solver must consider all possible offensive and defensive moves while also considering the interaction between the members of the N/S and E/W teams.

2.2 Minimax Search Algorithm

The minimax search algorithm is the basis for many game tree searches in which one player's losses are the opponent's gains. Games of this type are called two-player zero-sum games. In this discussion of the minimax algorithm, larger payoff values indicate better situations. In order to choose the child who will maximize success, an evaluation of each of the children must be done. These values are found through a recursive process in which the values for internal nodes are known only after evaluation of the leaf nodes. When the values of the leaf nodes are known, they are passed up the tree to internal ancestor nodes.

The minimax technique assumes the opponent will choose the move that will minimize the success of the other player. That is, the path of the child with the lowest value is chosen. Thus, at alternating levels in the game tree, there is an alteration of maximizing and minimizing which captures the antagonistic nature of the two-player zero-sum game. The following is a basic summary of the minimax strategy:

```
if x is a minimizing node,
   minimax(x) = minimum (minimax(child1), ... , minimax(childn))
if x is a maximizing node,
   minimax(x) = maximum (minimax(child1), ... , minimax(childn))
if x is a leaf node,
   minimax(x) = evaluation of x.
```

The minimax value of the root node is the optimal score achievable against an opponent who similarly follows the minimax strategy. Suppose, as is illustrated by Figure 2.1, that player one, whom we will call MAX, has the choice of moving to A or B. If MAX moves to A, player two, whom we will call MIN, will choose to move to D since this will allow MAX to have only a score of one instead of four. If MAX moves to B, then MIN can hold MAX's score only to five. Thus it

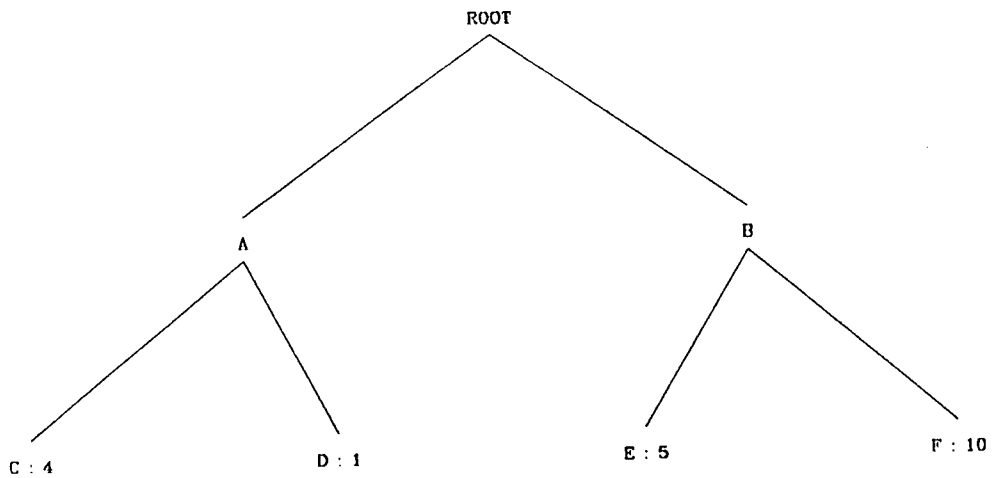


Figure 2.1: Example Game Tree I

would be advantageous for MAX to move to B. By following the minimax strategy, MAX is guaranteed the optimal score of five if MIN is also following the minimax strategy. If the opponent is using minimax, you cannot do any better than the score obtained by following the minimax strategy yourself.

However, in any true game there is room for error. If during a game, the opponent happens to blunder and choose a move that did not maximize his or her success, the value for the other player may be higher than the minimax value. In this situation, the opponent is obviously not following the minimax search strategy. Referring to Figure 2.1 again, if MIN had chosen to make a move in the game corresponding to leaf node F after MAX had moved to B, then MAX's score would have been 10 instead of 5. In a real game, if MAX follows the minimax strategy, MAX will always score at least the minimax value of the game tree no matter what strategy the opponent is following. MAX could possibly score higher than the minimax value of the game tree if the opponent is not following the minimax strategy. A further explanation of the minimax strategy can be found, for example, in "Artificial Intelligence" by Elaine Rich [15].

The minimax algorithm is a depth first search procedure described by the following "pseudo-lisp" code:

```
(Minimax (node)
  (cond ((node is a leaf)
            (return (evaluation node)))
        ((node is a minimizing node)
            (setq children (expand node))
            (setq minimum infinity)
            (dolist (child children)
               (setq minimum (min minimum (minimax child))))
```

```
        (return minimum))
    ((node is a maximizing node)
        (setq children (expand node))
        (setq minimum -infinity)
        (dolist (child children)
            (setq maximum (max maximum (minimax child))))
        (return maximum))))
```

The function "expand" returns a list of the children of node. The function "evaluation" returns the value if the node is a leaf.

2.3 Searching and Pruning

One of the most predominant techniques in the field of Artificial Intelligence is the state space search. This technique is computationally expensive as most exhaustive search algorithms are exponential. Considerable effort has been devoted to reducing the scope of search for many game tree searches. Normally, an exhaustive search procedure such as minimax is performed until some cutoff criteria such as a time constraint is met. Then, a heuristic evaluation is produced for the subtree which was not explored due to the cutoff criteria. The evaluation is then used to continue searching the game tree. Since estimates are used, the result obtained may not be the best answer. The goodness of the result depends on the goodness of the heuristic evaluators. One advantage, though, to using heuristic evaluators is that searches of game trees which could not be completed with an exhaustive search can be completed.

It is also of interest to compute true minimax values without exhaustive search using techniques of Artificial Intelligence and techniques derived from

known facts of the game in order to produce a search tree that is significantly smaller than an exhaustive game tree. A search tree such as this is formed by pruning unnecessary branches of the tree. Techniques of this nature back up bound information for nodes that have pruned some or all of their children. An example of this can be seen in the contrived game tree of Figure 2.2 in which player

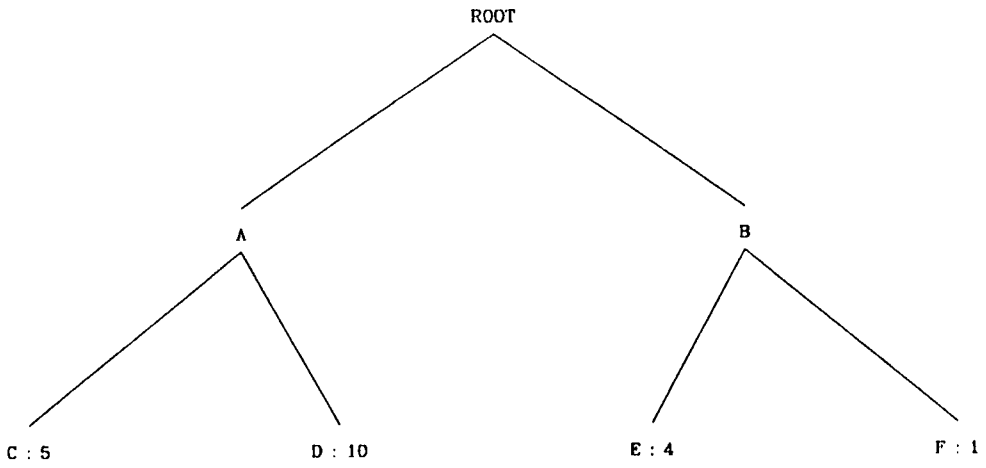


Figure 2.2: Example Game Tree II

one, MAX, wishes to maximize his score. For an exhaustive minimax search of this tree, six nodes must be visited to determine the minimax value. But, if we consider the information that we obtain throughout the search, we could obtain the same true minimax value by only visiting five nodes. We will see that it is

not necessary to visit node F.

The bound value that we return from a pruned node is sufficient to allow the ancestor to make a correct judgement of which child is the better choice. In the contrived game tree of Figure 2.2, player one, MAX, wishes to maximize his score. From search of the left subtree, MAX knows that his final score is at least five: if MAX chooses node A as his move, player two, MIN, will choose node C in order to minimize MAX's score. Now, let's examine the right subtree. If MAX chooses node B as his move, player two may choose either node E or F. Upon examination of the move to node E by MIN, we realize that a score of four or less will be MAX's score if the game follows this path. We know this upper bound before node F has been examined. Since MAX wishes to maximize his score, he will choose to move to node A. MAX knew a move to child A was a better choice than a move to child B without examination of node F. Thus the search tree is a subtree of the game tree.

The actual minimax value of the right subtree in Figure 2.2 is one. Thus, MAX would have still chosen A if the true minimax value had been found. The weaker statement that the minimax value of the right subtree is four or less is sufficient information to allow the correct decision to be made by MAX. Exhaustive search of the game tree is no longer necessary if we use the bound information. Similarly, we could produce an example where a child of MAX is pruned. Pruning seems to result in a relatively trivial amount of gain in this particular tree. Pruning one out of six nodes is approximately a 16% decrease in the number of nodes visited. But, a 16% decrease in the number of nodes visited in a game tree with, for example, 100,000 nodes is quite significant.

The prune that occurred in Figure 2.2 is an example of an alpha-beta prune [13, 15, 17] that will be discussed in Chapter 4. Several other techniques discussed in this thesis allow children to be pruned but for different reasons than the pruning

of node F from the tree of Figure 2.2. The similarity that must be noted is that the value backed up from a node, some of whose children have been pruned, is a bound on the actual minimax value of that node. The pruning, for whatever reason, insures that the value obtained through a search using bound information for pruning is the true minimax value of the game tree. We will discuss several pruning techniques in this thesis, all of which allow us to obtain the true minimax value without exhaustively searching the game tree.

Chapter 3

EXPERIMENTAL DESIGN

For the work reported here, several search techniques were implemented for the double-dummy bridge game. Some of the techniques implemented here were originally developed in other work for other games, while some of the techniques were developed as part of this work on the double-dummy bridge game. The basic algorithm used is the minimax search algorithm. Search limiting techniques were incorporated into the minimax strategy in order to reduce the search space for the double-dummy bridge game. Each technique is added on top of the accumulation of previous techniques. The question, "What is the maximum number of tricks that the N/S team can take against a perfect defense?", can be answered by the algorithms when given any random hand that is within the frontier. A frontier, a measure of the maximum size games that a technique can successfully complete in less than ten hours, was found after the addition of each technique. The formation of this frontier will be discussed later. There are no heuristic evaluators within these techniques. This criteria is set in order to gain only perfect information, not an estimate, of the number of tricks the N/S team can take against a perfect defense. Thus, in all cases the solution is optimal.

As we have already noted, the exhaustive minimax search for a full 52 card game of double-dummy bridge will take over three million years (see Appendix

A). The techniques that are implemented in this project are valuable attempts to produce pruning techniques that allow the 52 card game search tree to be searched within a reasonable amount of time. Some techniques result in more pruning than others. A new frontier is determined after the addition of each of the techniques. It should be noted that the techniques were built on top of each other. That is, new techniques are added to the code that contains all of the previously mentioned techniques. This is in order to gain all possible reduction of the game trees from all the techniques. The frontier defines the size of bridge hand that can be searched within a reasonable amount of time. For the development of this frontier, a reasonable amount of time is defined as less than ten hours of CPU time. Ten hours was chosen as it allowed jobs to be run overnight with results being available the next morning. The frontier gives us an approximation on the size of the games for which a particular technique or accumulation of techniques is able to produce results.

Several important variables were left as parameters to the search algorithms. These include the variables NUM- SUITS, NUM-CARDS, NUM-PLAYERS, and GAME-TYPE which are the values of the number of suits, the number of cards in each suit, the number of players in the game, and the game type. The GAME-TYPE variable will be discussed in detail later. These variables are left as parameters in order to gain experimental data for games with varying hand sizes and varying game rules. We allow these variations in order to determine the performance of each technique, in particular to define the frontiers for the techniques. Though NUM-PLAYERS is one of the parameters that may vary, this project discusses only games of 4 players. A normal game of bridge corresponds to NUM-SUITS equal to four, NUM-CARDS equal to 13, NUM-PLAYERS equal to four, and GAME-TYPE equal to B.

It should be noted that some hand sizes such as hands of two suits with

five cards per suit are not legal hand sizes. A hand is a legal hand size if each player, North, South, East, and West, is dealt the same number of cards. Thus the total number of cards must be a multiple of four. See Figure 3.1, Figure 3.2, Figure 3.3, and Figure 3.4 for sample legal hands of one, two, three, and four suits respectively. For example, in a game with only one suit, there must be one of {4, 8, 12, ... } cards in the hand for it to be a legal hand size.

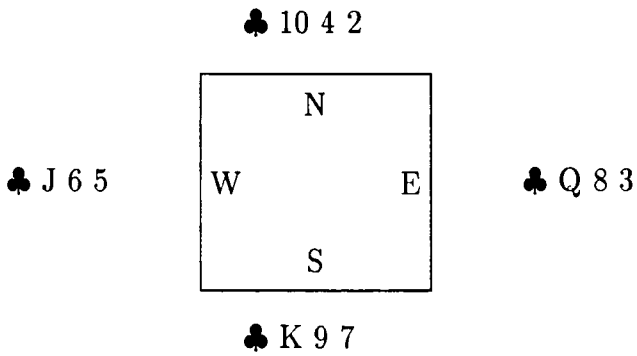


Figure 3.1: Sample Legal One Suit Hand with 12 Cards

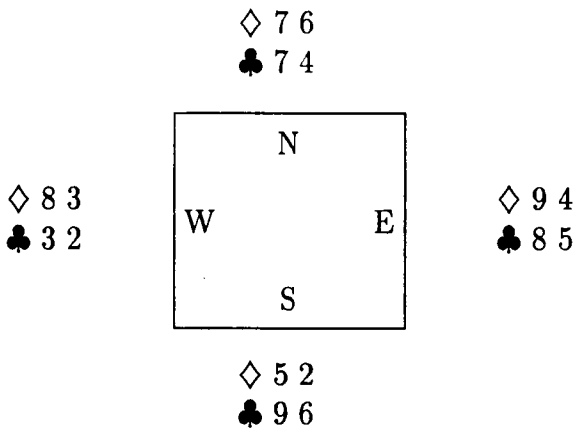


Figure 3.2: Sample Legal Two Suit Hand with 16 Cards

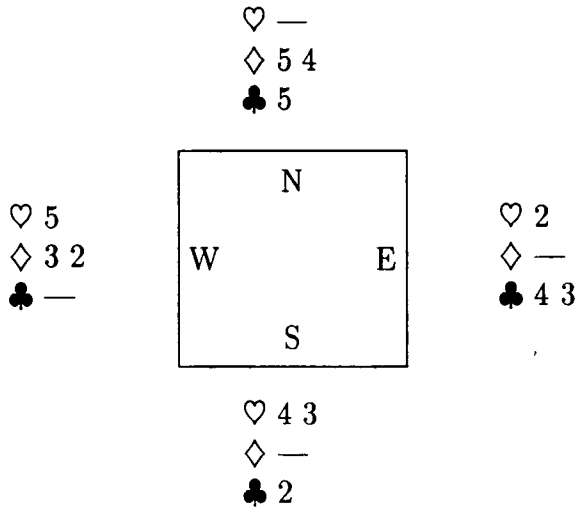


Figure 3.3: Sample Legal Three Suit Hand with 12 Cards

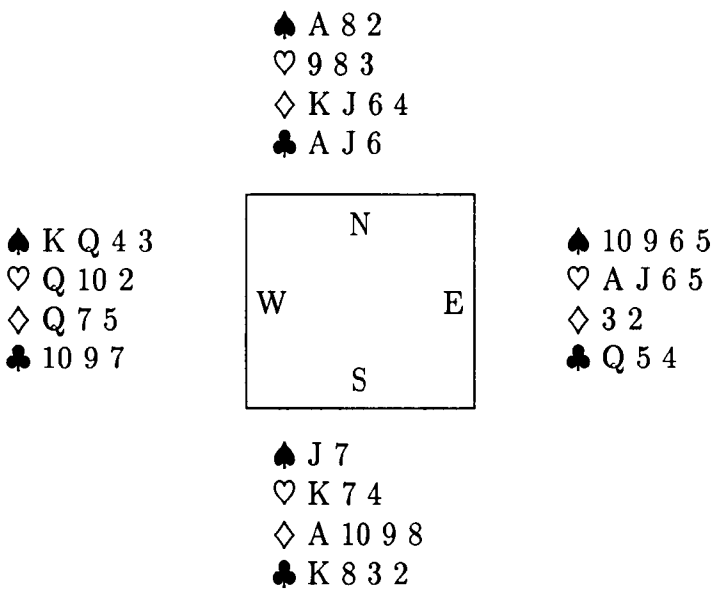


Figure 3.4: Sample Legal Four Suit Hand with 52 Cards

The frontiers for the techniques are discussed in Chapter 4 but a sneak preview is necessary here in order to understand the development of the frontier. The frontier consists of four subfrontiers, one for each of one, two, three, and four suit games. The subfrontier for one suit games is a number from {4, 8, 12, ... }. Suppose the subfrontier for one suit is 12. We then say that the average time to complete the search of the game tree for a one suit game with a maximum of 12 cards is less than ten hours. Furthermore, this means the average time to complete the search of the game tree of one suit games with more than 12 cards is less than ten hours. The other three subfrontiers are defined similarly for hands with two, three, and four suits.

Table 3.1 shows the frontier for the minimax search strategy. The average

Table 3.1: Frontier Development Table for Minimax Search Strategy

Number of Suits	Number of Cards				
	4	8	12	16	20
1	*	*	*	*	
2	*	*	*	*	
3			*		
4	*	*	*	*	

CPU time for ten randomly dealt games for each of the legal hand sizes was found. For the minimax strategy, this was a total of 130 games. We also ran 200 randomly dealt games for one particular size hand. Results for this special case are discussed in Chapter 5. The positions in Table 3.1 marked with *'s indicate that the game tree for this particular size game could on average be successfully searched within ten hours using the minimax search strategy. Empty positions in the table to the right of the right-most *'s indicate hand sizes for which the minimax search strategy could not on average complete the search within ten hours of CPU time.

Note that there are no *'s in the positions of three suit games that indicate illegal hand sizes. For example, the minimax search strategy could on average search a game tree where the game consisted of four suits with up to 16 cards but could not on average search the game tree of a four suit game with greater than 16 cards. The same process was followed with the same ten randomly dealt games of the various sizes for each of the other techniques, to be discussed later, in order to develop their frontiers.

Each of the search algorithms are equipped to play four different variations of the game of bridge, games A, B, C, and D. The game of bridge that this thesis is involved with is game B. The game of double-dummy bridge described in section 2.1 corresponds to Game B. Game A is a variation in which either North or South leads each trick instead of only the winner of the trick leading the next trick. At each of the depths in the tree that correspond to the lead move of a trick, there are as many branches as the lead player has cards. For game A, these branches include the possible leads that North could make plus the possible leads that South could make. Thus there are twice as many branches in game A at depths corresponding to lead moves of a trick. Game C is a variation in which either North or South is allowed to lead when either North or South has the lead. East leads only when East has the lead and West leads only when West has the lead. Game D is a variation in which either North or South leads when either North or South has the lead. Either North, South, or East leads when East has the lead, and either North, South, or West leads when West has the lead. The motivation for the implementation of these variations is that in a true bridge game, it would be valuable to know for instance how many tricks the N/S team could take if allowed to lead every trick. Information such as this would facilitate decision making in some aspects of the bridge game. The bulk of this project does not deal with decisions such as this. These variations were incorporated to enable

future extensions of this project.

For each game played, several data items were collected. This data includes CPU time, real time, the number of nodes expanded in the tree, the number of nodes generated in the tree, and the result of the search. The CPU time is used as an indicator for the development of the frontier. The real time is collected as supplemental information. The number of nodes expanded and the number of nodes generated in the tree are measures of the size of the tree used to evaluate the performance of the technique as a pruning mechanism. The nodes generated in the tree include all nodes in the tree, but the nodes expanded in the tree include only the nodes which have their children in the tree. The result of the search is a number indicating the number of tricks the N/S team can take against a perfect defense, a defense which follows the minimax strategy of always choosing the best move. In one particular technique, several additional data items were collected. These will be discussed along with the explanation of the technique in Chapter 4.

After the data for the randomly dealt games was collected, a process of statistical analysis was necessary. Through an automated process of Unix *shell* and *awk* programs, the entire collection of data for a particular technique was statistically evaluated. Approximately 1200 lines of various Unix *shell* and *awk* programs were written for the entire data collection process. Any of the following measures could be chosen as the basis for the statistical data collection: CPU time, real time, number of nodes generated, or number of nodes expanded. The minimum, maximum, and average values of the measure are found for each of the legal hand sizes. Also, this process calculated the 90%, 95%, and 99% confidence interval for the chosen measure. The 90% confidence interval is the interval within which the average of the measured value will lie 90% of the time, and similarly for the 95% and 99% confidence intervals. The data collection process took place over several months. Approximately 290 random games were played by each of

the various techniques with approximately 2.9 megabytes of data being stored.

These algorithms were implemented in approximately 1,850 lines of Common *Lisp* code. They were tested after compilation by an Ibuki compiler on Sun 3/60 workstations. The average CPU time per node varies with the different techniques due to the variation in difficulty and size of the implementations of the techniques. Table 3.2 lists the techniques, their approximate CPU time per node

Table 3.2: Approximate CPU Time per Node for the Search Techniques

Technique	CPU Time/Node (Sec)	Nodes/Sec
1. Minimax	0.022910256	43.6
2. Alpha-beta, random node order	0.027873252	35.8
3. Alpha-beta, good node order	0.030383393	32.9
4. Strict Bucketing	0.031812394	31.4
5. Internal Cutoffs	0.033474597	29.9
6. Deferred Branching	0.127009084	7.8
7. Hashing	0.155103386	6.4

expanded, and the average number of nodes expanded per second. This table gives us some indication as to which techniques take up the most CPU time. It should be noted that only a minimal attempt was made at code optimization for the algorithms in this thesis. A more detailed evaluation of the CPU time will be discussed with regard to forming the frontiers of each of the techniques.

Though the CPU time is an important measure when dealing with real problems such as the game of double-dummy bridge, other measures are important to distinguish the performance of the strategies. These measures include the number of nodes expanded, the number of nodes generated, and the number of terminal nodes in the game tree. Because implementation on different machines or in different languages greatly affects the outcome of time calculations, time is

not the measure with which this thesis compares the performance of the techniques. The CPU time, however, is necessary for the formation of the frontiers for the techniques. Although the average number of terminal nodes is the standard yardstick for the complexity of the search, we choose to use the average number of nodes expanded as the measure of complexity because expansion of nodes, not evaluation of terminal nodes, is where the major portion of the work is done in these algorithms. We will compare the average number of nodes expanded for the pruning techniques to the average number of nodes expanded for the minimax strategy to gain a performance measure for the techniques.

The information needed to play the bridge game is kept in two main structures. The first of these is called the INFO structure. It stores the values of the number of suits, the number of cards per suit, the number of players, and the game type for the particular game being played. The *Lisp* structure INFO looks like:

```
(defstruct INFO
   num-suits
   num-cards
   num-players
   game-type).
```

The second structure is called the NODE structure. The search tree is built of the NODE structures. The NODE structure contains the information needed to keep up with the play of the game for the particular path on which the node lies. Therefore, each NODE structure will contain different information according to its position in the tree. The NODE structure contains the following information: the cards that have been played thus far, the randomly dealt hands that are being

played, the minimax type of the node, the number of tricks won so far on the path to the node, the hand which currently has the lead, the suit that has been led in the current trick, and the next hand from which a card must be played. The *Lisp* structure NODE looks like:

```
(defstruct NODE
    cards-played
    hands
    mmtype
    tricks-won
    lead
    suit-lead
    next-hand).
```

The **hands** element of the NODE structure is particularly interesting as it holds the information about the cards available in the North, South, West, and East hands. The randomly dealt hands that represent the game being played are implemented as an $n \times m$ two dimensional array, where n is the the number of suits and m is the number of cards per suit. Array element n, m tells us which player, South, West, North, or East has the mth card of the nth suit. The entries in the array are integers 1, 2, 3, and 4 where 1 designates that South has the card, 2 designates that West has the card, 3 designates that North has the card, and 4 designates that East has the card. For example, in a game with four suits and seven cards per suit, the game is represented in the following array:

```
m    0   1   2   3   4   5   6
n  ---------------------------
0  | 1 | 2 | 4 | 2 | 3 | 3 | 1 |
1  | 2 | 4 | 4 | 2 | 1 | 1 | 3 |
2  | 1 | 3 | 1 | 3 | 4 | 2 | 2 |
3  | 3 | 2 | 1 | 4 | 4 | 4 | 3 |
   ---------------------------
```

Or, to make it clearer the array represents the game in the following way:

```
suits  cards:  2   3   4   5   6   7   8
-----          ---------------------------
Clubs          | S | W | E | W | N | N | S |
Diamonds       | W | E | E | W | S | S | N |
Hearts         | S | N | S | N | E | S | W |
Spades         | N | W | S | E | E | E | N |
               ---------------------------
```

This latter representation of the **hands** array will be used throughout the thesis.

After a trick is played, the array is altered to indicate the cards which were played during the trick. Cards must be marked as played so that they will not be chosen to be played in the future. To indicate that a card has been played, the array element corresponding to that card is multiplied by ten. For example, if the 5, 6, 4, and 2 of ♣ are played in a trick, the array would be altered to indicate this by multiplying each of the array elements that correspond to these cards by ten. The array would then look like:

m	0	1	2	3	4	5	6
n							
0	10	2	40	20	30	3	1
1	2	4	4	2	1	1	3
2	1	3	1	3	4	2	2
3	3	2	1	4	4	4	3

Denoting played cards by multiplying by a factor of ten was chosen so that we could tell who the owner of a card was although that card has been played. We merely divide by ten to get either 1, 2, 3, or 4 which are mapped as explained above to South, West, North, and East respectively. In the representation of the hands array to be used throughout this thesis, we will denote a played card by including a (P) in the array element corresponding to the played card. The motivation for waiting to the end of a trick to mark cards as played as opposed to marking the cards played immediately after each card is played will be explained later when the technique of Strict Bucketing is discussed.

The trees produced for the double-dummy bridge game are not of a uniform branching factor. Two hands of the same size could produce two very different CPU times and very different shaped trees. In one randomly dealt hand, a player may not have any cards of one of the suits. But, in another randomly dealt hand of the same size, the player may have cards from all of the suits. In the former hand, he would have to discard when the suit of which he has no cards is led. When discarding, a card may be played from any available suit, thus there are many choices at this point in the game. These choices turn into branches in the game tree. Thus this game tree will be significantly larger than a game tree in

which no discards were necessary. Figures 3.5 and 3.6 are both games of four suits with eight cards. Figures 3.7 and 3.8 are the minimax game trees corresponding to Figures 3.5 and Figure 3.6 respectively. Notice that there are 45 nodes expanded in the game tree of Figure 3.5 but there are 79 nodes expanded in the game tree of Figure 3.6. Though the games are the same size, quite different game trees result.

Chapter 2 covered the preliminary concepts needed in order to understand this work. Chapter 3 explained the experimental design of this work. Now, in Chapter 4, we will look in detail at each of the techniques implemented for the work of this thesis.

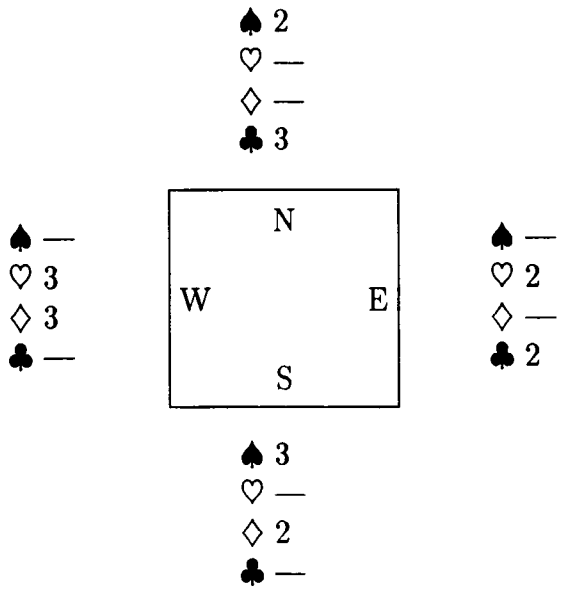


Figure 3.5: Game of Four Suits with Eight Cards Corresponding to the Game Tree in Figure 3.7

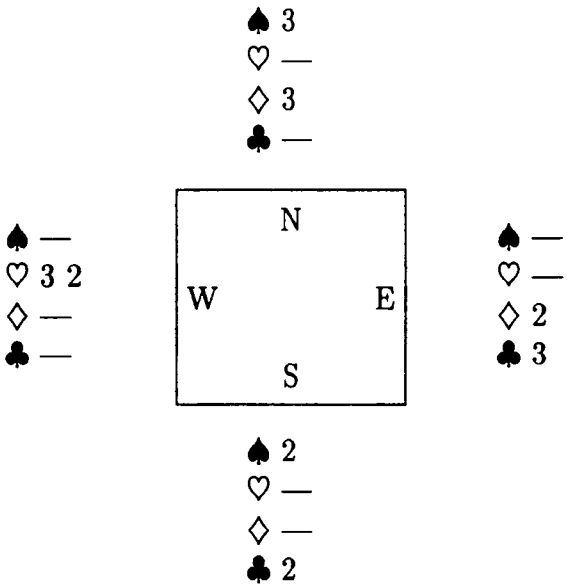


Figure 3.6: Game of Four Suits with Eight Cards Corresponding to the Game Tree in Figure 3.8

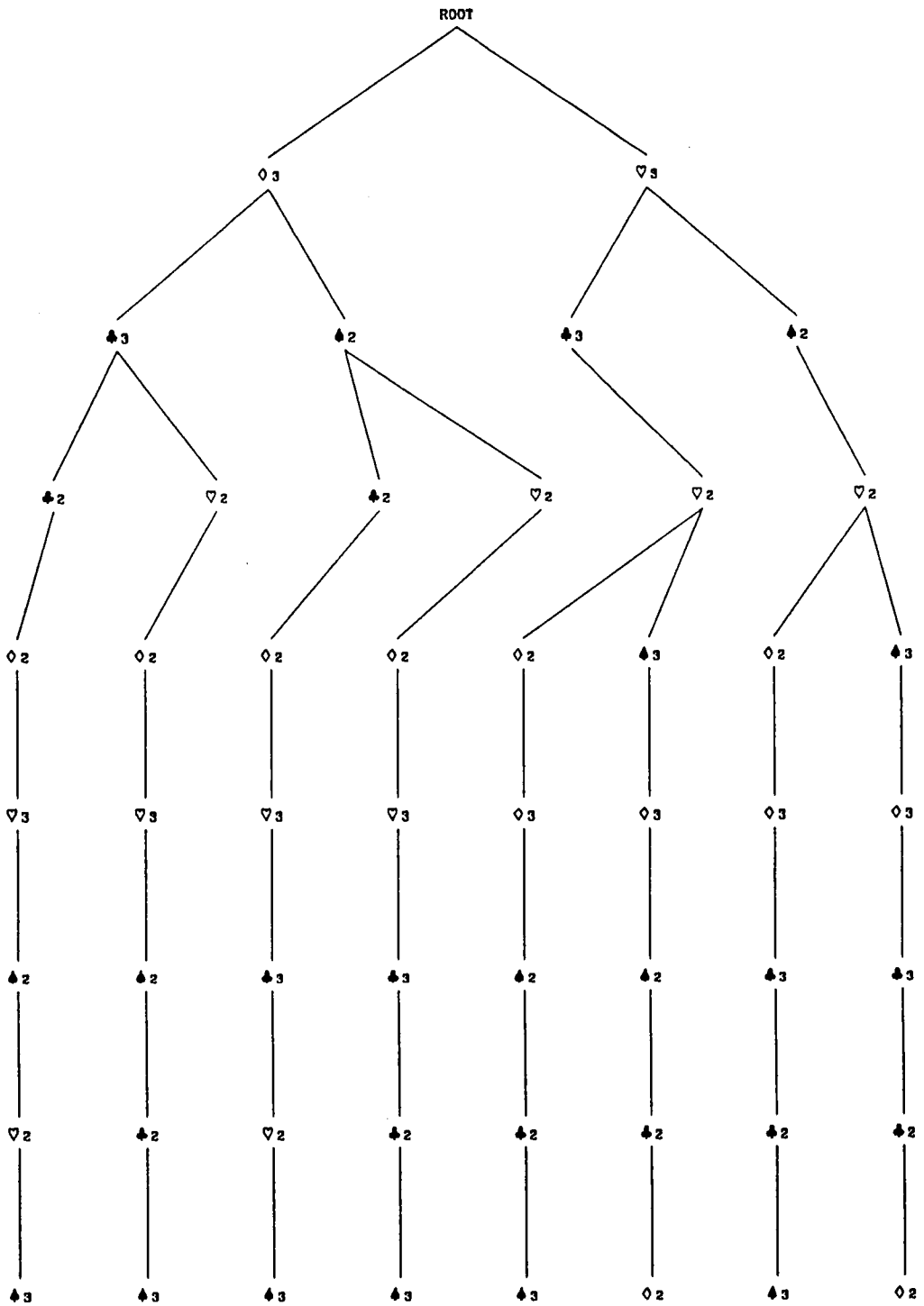


Figure 3.7: Game Tree of Four Suits with Eight Cards Corresponding to the Game Tree in Figure 3.5

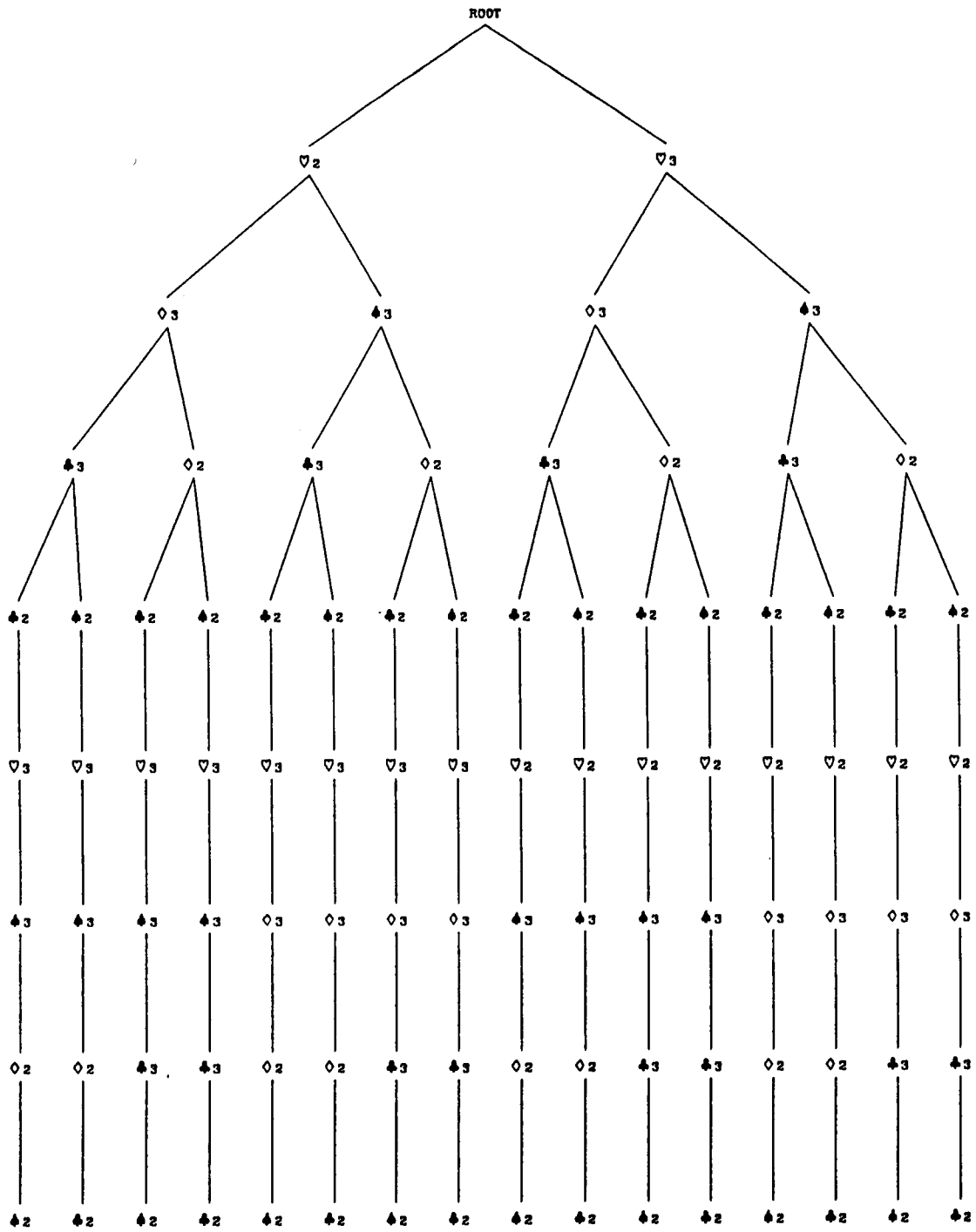


Figure 3.8: Game Tree of Four Suits with Eight Cards Corresponding to the Game Tree in Figure 3.6

Chapter 4

DISCUSSION OF THE TECHNIQUES

4.1 Minimax Search Technique

The minimax technique is the basic search technique for this project, as it is for many two-player zero-sum games. As was discussed in Chapter 2, the minimax score is the optimal score a player can receive against an opponent who similarly uses the minimax search strategy. Since the minimax technique was developed for two-player games, you may ask how the double-dummy bridge game can use such a technique. It is because of the team nature of bridge. Since the North and South players, and similarly the West and East players, work cooperatively to maximize the number of tricks the team wins, double-dummy bridge can be directly implemented by the minimax algorithm. There is still an alteration of maximizing and minimizing, the only difference is that the alteration occurs between four players instead of two.

There is however one important notion that must be taken into consideration. In the normal minimax strategy, there is an alteration of maximizing and minimizing at each consecutive level of the game tree. This alteration corresponds to the antagonistic strategy of the two players. However, in the game tree

of double-dummy bridge, there may be two consecutive levels of the tree where maximizing occurs, or similarly two consecutive levels of the tree where minimizing occurs. In particular, this happens when the last player of a trick wins the trick, thus forcing him to play the first card of the next trick.

The solution to this problem was straightforward. Normally, the minimax algorithm decides if it is at a maximizing or minimizing level by the current depth of the tree. Even depths refer to maximizing levels and odd depths refer to minimizing levels, or vice versa depending on the particular game. The minimax algorithm applied to the double-dummy game makes the decision to maximize or minimize by determining which of the four players is currently moving. That is, if it is either North or South's move, the game tree is at a maximizing level. This corresponds to the N/S team trying to maximize the number of tricks they win. If it is West or East's move, the game tree is at a minimizing level. This corresponds to the E/W team trying to minimize the number of tricks the N/S team take. Thus, the decision between choosing the child with the maximum value, maximizing, or choosing the child with the minimum value, minimizing, depends not upon the current depth of the tree, but upon the player currently moving.

Let's follow the game tree of Figure 4.1. Figure 4.2 is the game tree corresponding to the game in Figure 4.1. Initially, West leads. Thus, he has the choice of playing either his ♣ 3 or his ♢ 4. Notice the branch in the game tree of Figure 4.2 corresponding to West's choice. Just as the left to right minimax strategy does, we will follow the leftmost branches of the game tree first. The leftmost branch of the game tree corresponds to West playing the ♣ 3. Since West led a ♣, North must follow suit if he has a ♣ card. Since both of North's cards are in the ♣ suit, North also has a choice of which card to play. Again, notice the branch in the game tree corresponding to this choice. Again, we follow

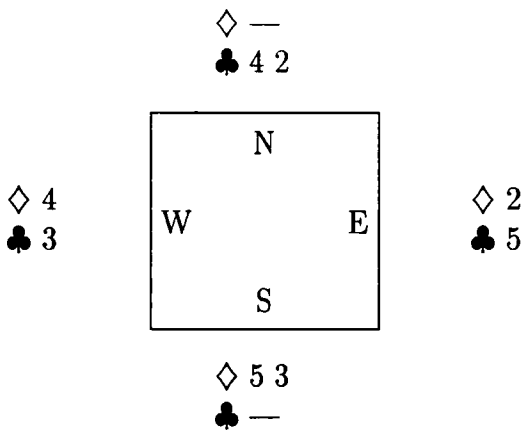


Figure 4.1: Game of Two Suits with Eight Cards Corresponding to the Game Tree in Figure 4.2

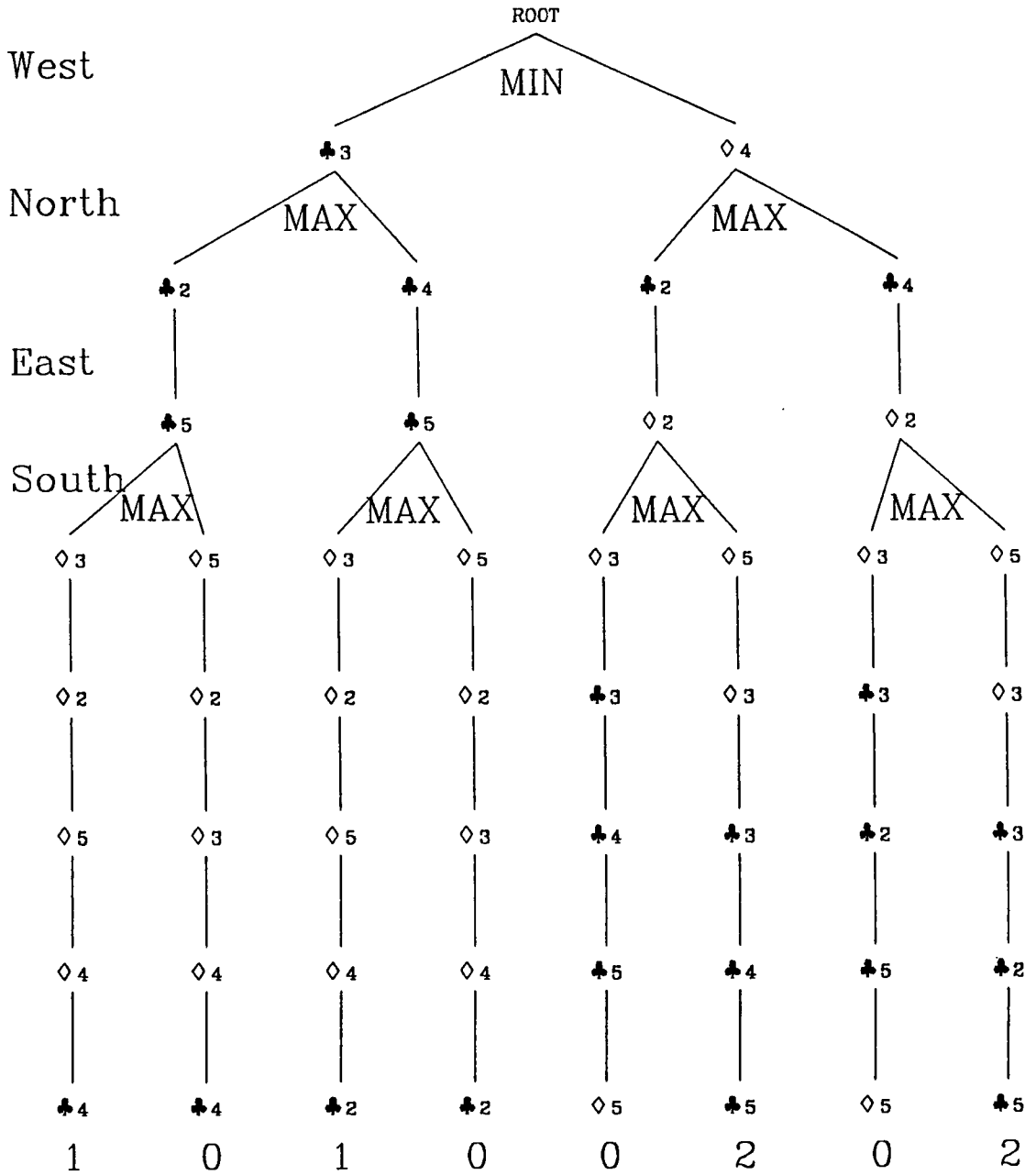


Figure 4.2: Minimax Game Tree Corresponding to Figure 4.1

the leftmost branch corresponding to North playing the ♣ 2. Now it is East's turn to play a card. East must follow suit and play his ♣ 5. Notice there is no branching in the game tree at the level corresponding to East's move. Now, South must finish the trick by playing a card. Since South has no cards of the ♣ suit, he must discard. That is, he may play any card from his hand. South does not have a chance to win this trick since he does not have a card of the lead suit. The number of tricks won by the N/S team on each path and the MAX's and MIN's are included in Figure 4.2 as an aid in the explanation of this game tree, but will not be included on future trees analyzed here.

To summarize, the cards of this first trick along the leftmost branch were ♣ 3, ♣ 2, ♣ 5, and ♢ 3. The winner of the trick is East as he played the ♣ 5. Thus, East leads the next trick. Since each player now has only one card, there is no branching in the game tree for this next trick. The cards played are ♢ 2, ♢ 5, ♢ 4, and ♣ 4. The winner of this trick is North since he played the ♢ 5. The minimax value of the terminal node is one since one trick may be won on this path by the N/S team. The next node examined in the depth first search is the ♢ 5 discarded by South at the end of the first trick where ♣ 3, ♣ 2, ♣ 5 have been played. On this path, the winner of the first trick is still East. But the winner of the second trick on this path is West. The minimax value of the terminal node on this path is zero since zero tricks are won by N/S on this path. This path is obviously not as good for the N/S team as the first path. Since South discarded his ♢ 5 in the first trick, he is not able to win the second trick as he did on the first path.

The N/S team gets a better score if South discards the ♢ 3. We say the minimax value of the ♣ 5 node is one. This value is passed up the tree until the ♣ 3 is reached. At this point, we know the minimax value of the left subtree of the ♣ 3 is one. We must traverse the right subtree to determine the minimax value

of the ♣ 3 node. Since the children of the ♣ 3 node correspond to North moves, the ♣ 3 is a maximizing node. Thus, if the minimax value of the right subtree is less than one, one will be returned as the minimax value of the ♣ 3 node. The minimax value of the right subtree of the ♣ 3 node turns out to be precisely one. Thus, a one is returned to ROOT as the minimax value of the left subtree of the game tree. The minimax value of the right subtree of ROOT is two. The search of each possible path of this game tree results in a final minimax value of the root node which corresponds to the maximum number of tricks that the N/S team may take. Since the ROOT is a minimizing node, the maximum number of tricks the N/S team may take is one.

Since the minimax search technique is the basic search strategy of this project, it is interesting to note the frontier data obtained for the minimax search strategy. Table 4.1 is the frontier table for the minimax strategy developed from

Table 4.1: Frontier Development Table for Minimax Search Strategy

Number of Suits	Number of Cards				
	4	8	12	16	20
1	*	*	*	*	
2	*	*	*	*	
3			*		
4	*	*	*	*	

the average CPU times of each of the legal hand sizes. Notice that in the case of hands with three suits, only hands with up to twelve are doable in less than ten hours of CPU time. Also note that hands of three suits with four, eight, or sixteen cards are illegal hand sizes, thus there are no *'s in these positions. The exponential explosion that occurs as games with larger size hands are attempted can be seen in Figure 4.3. Notice that the y axis is a logarithmic scale. From Table

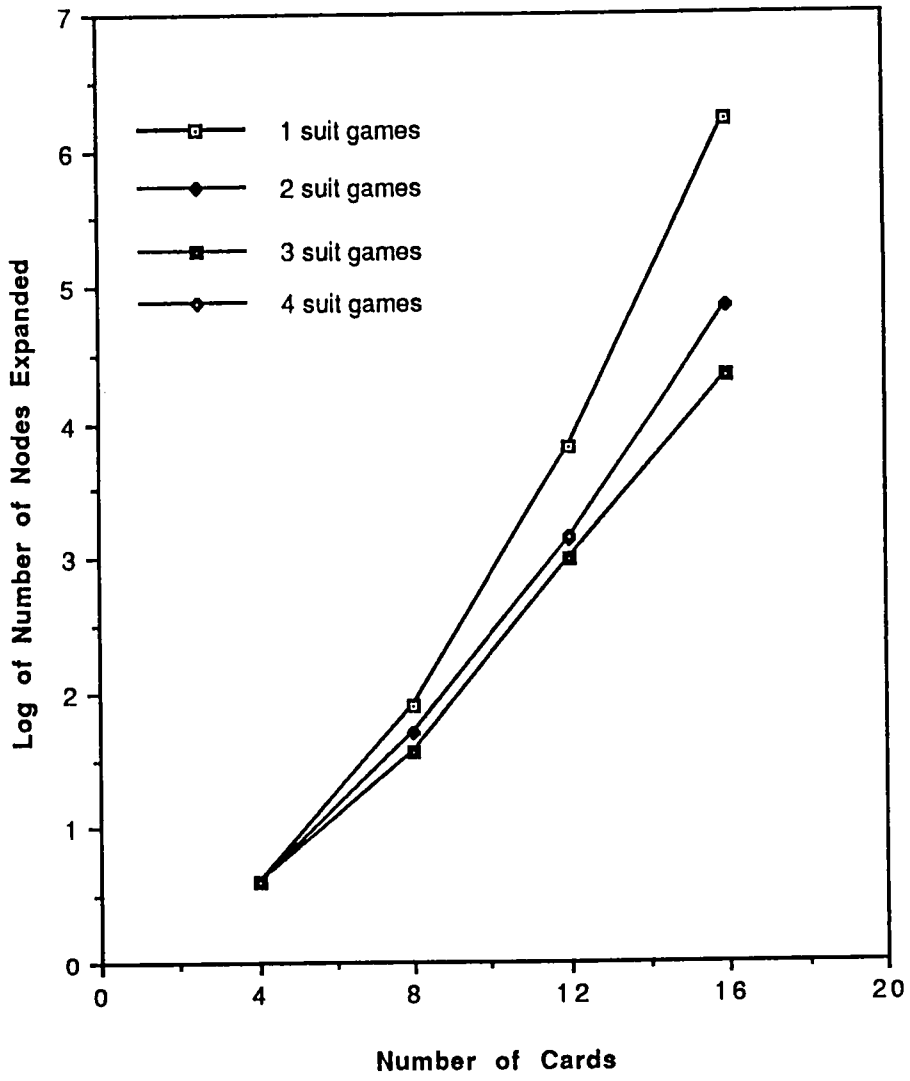


Figure 4.3: Minimax Search Strategy - Average Number of Nodes Expanded

4.1, we see that in a game of four suits, as in a double-dummy game, sixteen cards is the frontier from which the other pruning techniques must start. The frontier must be expanded from sixteen to fifty-two in order for the full double-dummy bridge game to be solved. The following search methods prove to be useful towards reaching this goal.

4.2 Alpha-Beta Pruning Technique

The alpha-beta pruning technique [13, 15, 17] is useful for limiting searches without loss of information. The alpha-beta cutoff or alpha-beta pruning principle is based on two threshold values that are kept throughout the game tree search. These two threshold values provide information that allows pruning of some of the children of a node. The two threshold values relate to the two antagonistic players. One of the threshold values, called ALPHA, denotes the best value that the maximizing player, MAX, has been able to obtain at any maximizing node that is an ancestor of the current node of the search. Therefore, ALPHA is a lower bound on the value that MAX may obtain. The other threshold value, called BETA, denotes the least value that the minimizing player, MIN, has been able to obtain at any minimizing node that is an ancestor of the current node of the search. Therefore, BETA is an upper bound on the value that MIN may obtain.

Since these two threshold values are kept and updated throughout the search, the value of ALPHA will get larger and the value of BETA will get smaller throughout the search. Let's follow the game of Figure 4.4 to see prunes occurring in a real game tree. The search tree corresponding to Figure 4.4 is Figure 4.5. Before searching begins, an initial value must be given to ALPHA and BETA. These initial values should be out of the range of valid results for this tree. We

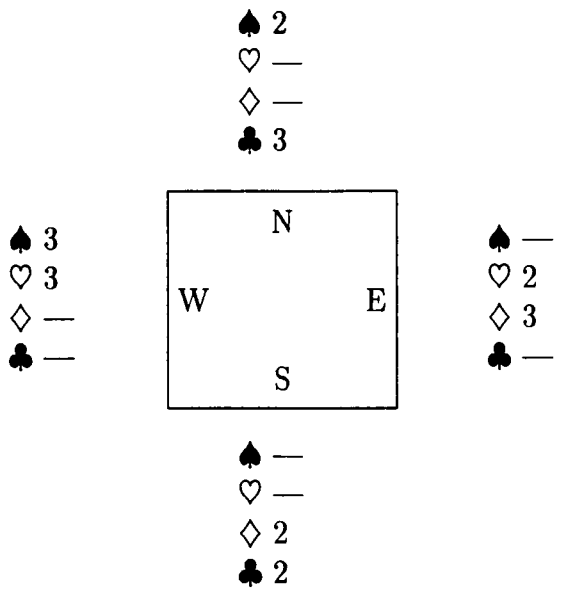


Figure 4.4: Game of Four Suits with Eight Cards Corresponding to the Game Tree in Figure 4.5

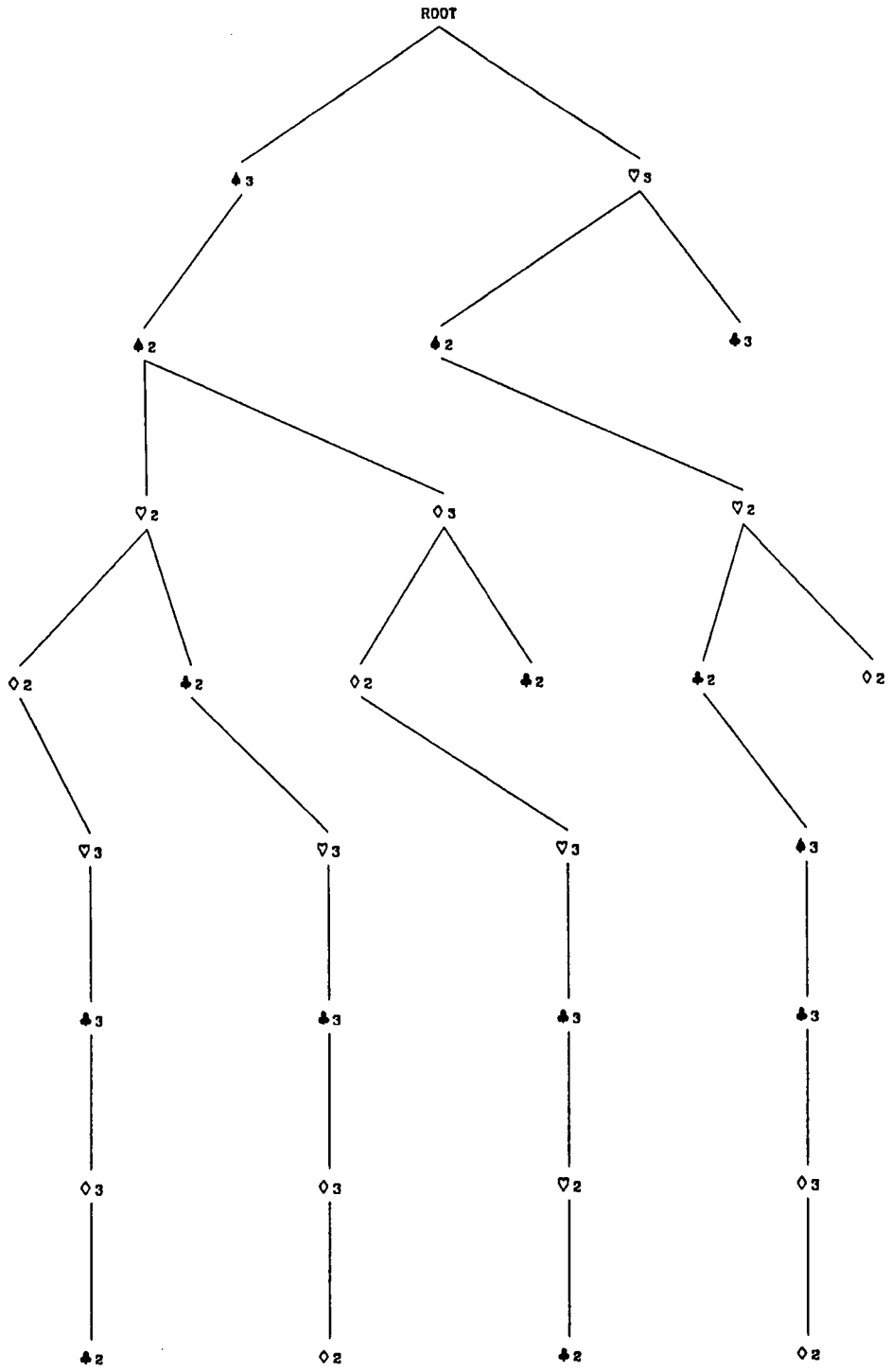


Figure 4.5: Alpha-Beta Search Tree Corresponding to Figure 4.4

must remember that in the double-dummy game tree, the value we are returning is the number of tricks the N/S team may take. Since we are looking at a game of four suits with two cards per suit, valid results include 0, 1, and 2. Thus we should set ALPHA to a value less than zero and our BETA value to a value greater than two. We will choose ALPHA equal to -100 and BETA equal to 100.

It is important to note that the order of the children of the nodes were randomly generated for this technique. That is, in the tree of Figure 4.5 the ♠ 3 was randomly chosen to come before the ♡ 3 for the children of the root. The relevance of this fact will become clear as the next technique deals with trees in which the children are put in a "good" ordering. Only 24 nodes were expanded in the search tree of Figure 4.5 as compared with 42 nodes being expanded in an exhaustive minimax search. When the depth-first alpha-beta search procedure reaches the ♢ 3 node whose children are the ♢ 2 and the ♣ 2, the ALPHA value is still equal to -100 but the BETA value has become 0. Upon search of the left subtree from the current node, a value of 0 is returned. Since this is a maximizing node we update our current ALPHA value to 0. But since BETA is also 0 we know that the opponent can prevent the N/S team from getting a score higher than 0. Thus there is no need to search the right subtree of the ♢ 3 node. There are two other alpha-beta prunes in this tree that occur for similar reasons. The alpha-beta cutoff criteria is summarized as follows:

```
if at a minimizing node and a child's value <= ALPHA then
     terminate search of the rest of the children and
     return that child's value
if at a maximizing node and a child's value >= BETA
     terminate search of the rest of the children
     and return that child's value
```

The alpha-beta pruning technique has been analyzed in great detail for

game trees with a uniform branching factor. Judea Pearl explains in "The Solution for the Branching Factor of the Alpha-Beta Pruning Algorithm and its Optimality" [13] that for a given search time allotment, the alpha-beta pruning technique on average allows the search depth to be increased by a factor of 4/3 over that of an exhaustive minimax search. As explained in Chapter 3, the game trees for the double-dummy game are not of a uniform branching factor. Thus, the theorem by Pearl is not directly applicable to the trees analyzed here. We expect favorable results nonetheless from the application of alpha-beta pruning to the double-dummy bridge game.

We begin our analysis of the alpha-beta pruning technique on our game trees with a look at Table 4.2, the frontier development table for the alpha-beta

Table 4.2: Frontier Development Table for Alpha-Beta Search Strategy

Number of Suits	Number of Cards						
	4	8	12	16	20	24	28
1	-	-	-	-	*		
2	-	-	-	-	*	*	
3			-			*	
4	-	-	-	-	*	*	

technique. Positions marked with -'s indicate the frontier before the addition of the alpha-beta pruning technique and positions marked with *'s indicate the expansion of the frontier due to the addition of the alpha-beta pruning technique. We see from Table 4.2 that we were able to extend our frontier, in comparison with the frontier for the minimax strategy in each of the cases. We were able to search 3/2 deeper than the exhaustive minimax search. This is better than the average increase of 4/3 proposed by Pearl in [13]. Further analysis is needed to evaluate fully the performance results for the alpha-beta technique.

We see from analysis of Figure 4.6, Figure 4.7, Figure 4.8, and Figure 4.9 that the complexity of the search tree is significantly decreased when compared with the minimax strategy. Notice that this is a logarithmic scale. It is interesting to note the percentage decrease in the number of nodes expanded as compared to the minimax strategy. Table 4.3 summarizes this information. In particular, for a hand of four suits with five cards per suit, the alpha-beta strategy decreased the number of nodes expanded by 99%. Even with this 99% decrease in the number of nodes expanded, there is only a 50% expansion in the frontier. This is due to the exponential increase in the size of the game tree as larger hands are attempted. The frontier for the alpha-beta technique, Table 4.2, reflects this decrease as the frontier is expanded for hands of one, two, three, and four suits.

4.3 Good-Ordering Technique

The efficiency of the alpha-beta technique depends upon the ordering of the nodes in the tree. Recall Pearl's explanation in [13] that on average the alpha-beta technique allows search to go 4/3 deeper than the minimax strategy in the same amount of time for trees with uniform branching factor. Unfortunately, if the nodes are ordered in the worst possible way, the alpha-beta search strategy evaluates the same nodes that the minimax strategy does [6]. That is, the alpha-beta search is exhaustive with nodes ordered in the worst possible way.

Knuth and Moore's "An Analysis of Alpha-Beta Pruning" [6] provides an analysis of the alpha-beta pruning algorithm for game trees of uniform branching factor. Again, the results of this study are not directly applicable to the game trees for the double-dummy bridge game because the double-dummy bridge game trees do not have a uniform branching factor. We do though aspire to gain results similar to the favorable results of their study. The Knuth and Moore study showed

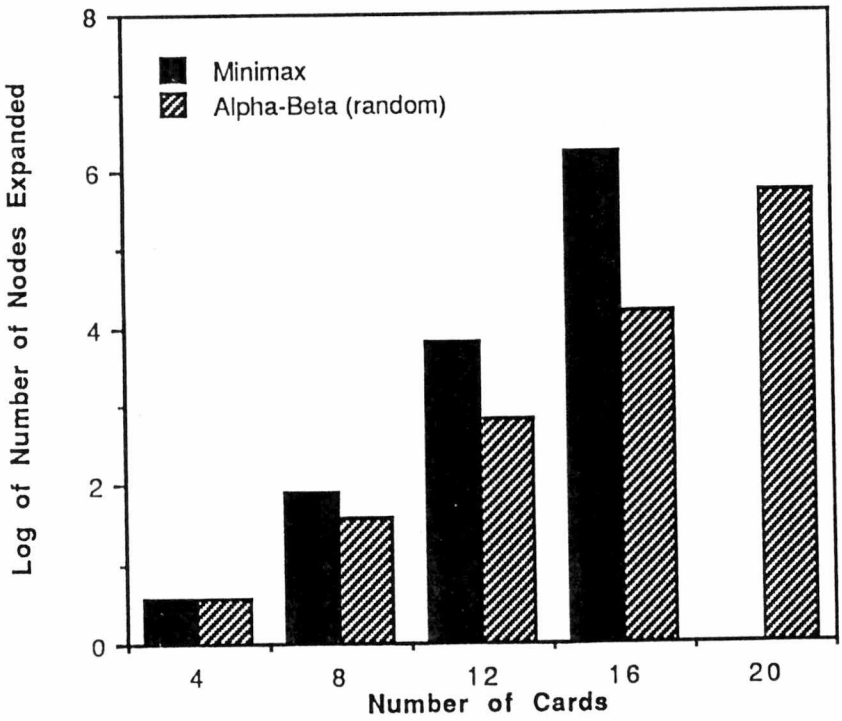


Figure 4.6: Alpha-Beta (random) vs. Minimax, 1 suit games

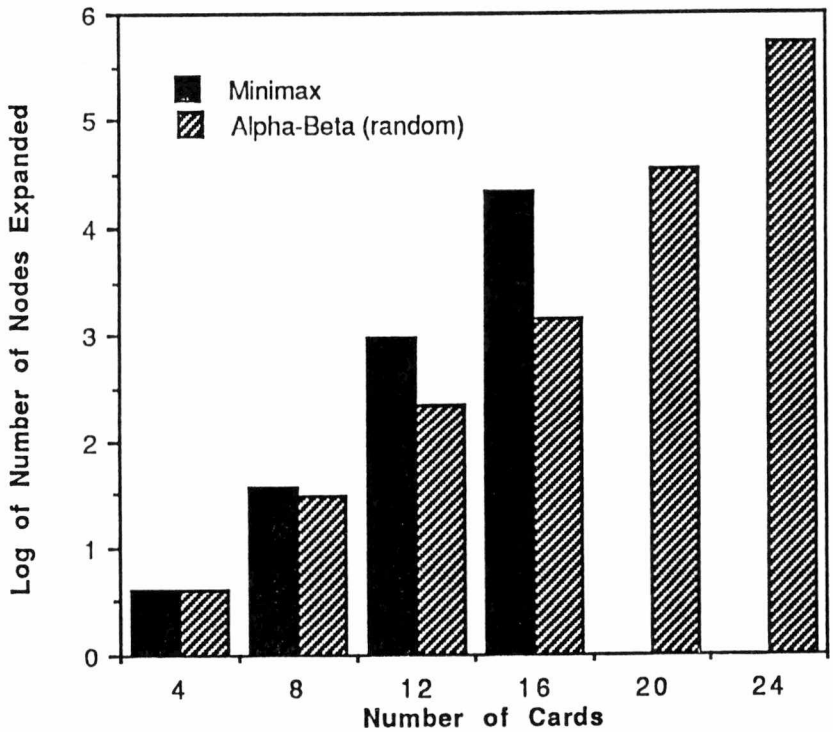


Figure 4.7: Alpha-Beta (random) vs. Minimax, 2 suit games

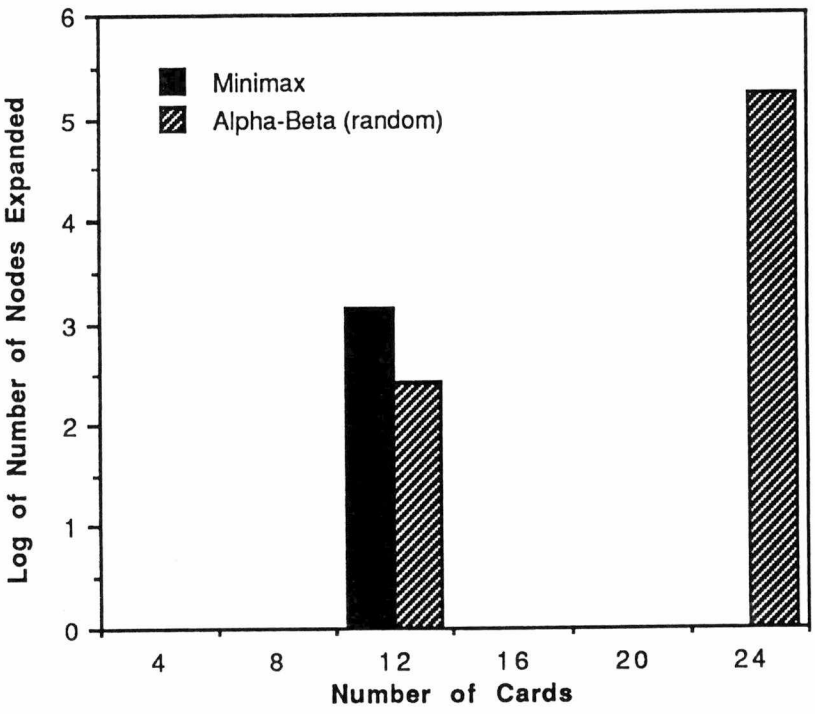


Figure 4.8: Alpha-Beta (random) vs. Minimax, 3 suit games

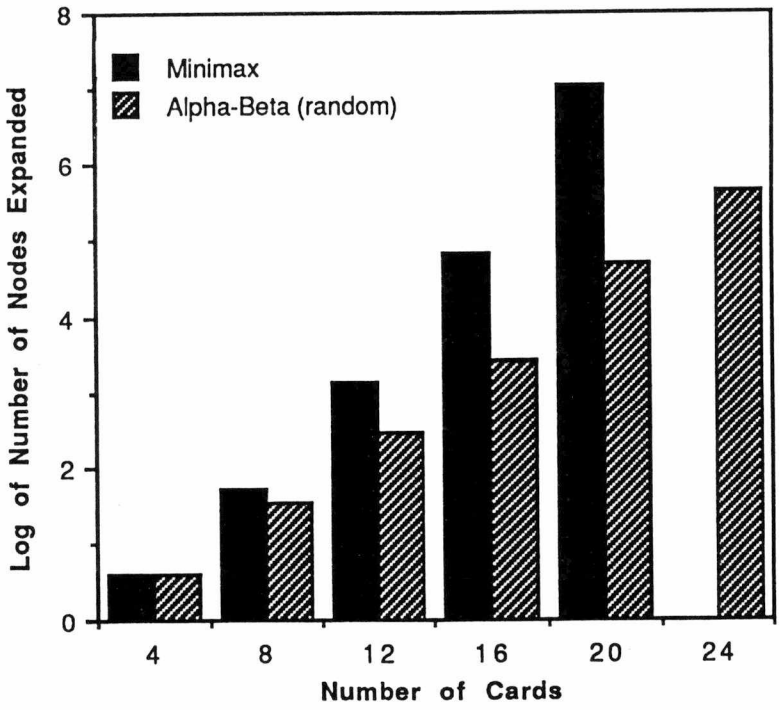


Figure 4.9: Alpha-Beta (random) vs. Minimax, 4 suit games

Table 4.3: Game Tree Complexity Comparison for Minimax vs. Alpha-Beta Search Techniques

Number of Suits	Number of Cards In The Game	Number of Nodes Expanded
1	4	0%
1	8	48%
1	12	90%
1	16	99%
1	20	—
2	4	0%
2	8	14%
2	12	77%
2	16	93%
2	20	—
2	24	—
3	12	81%
3	24	—
4	4	0%
4	8	34%
4	12	79%
4	16	96%
4	20	99%
4	24	—

that for a game tree of depth d with uniform branching factor b and nodes ordered such that the first successor of every node is optimum, the least number of nodes examined on level d is $b^{d/2} + b^{d/2} - 1$. The number of terminal nodes examined in the worst case is b^d just as in the minimax search. Pearl in [13] notes the interesting corollary that the number of terminal nodes considered by a search to depth d using alpha-beta pruning with nodes perfectly ordered is approximately equal to twice the number of terminal nodes generated by a search to depth $d/2$ with the minimax search technique.

Figure 4.10 shows the search tree produced by the alpha-beta search strategy for the game of Figure 4.1 on page 30. For the tree in Figure 4.10, the children of each node were randomly ordered. Notice that no pruning occurred, and thus Figure 4.10 is an exhaustively searched game tree just as is the game tree that the minimax strategy produced, Figure 4.2 on page 31. If the children of each node are reordered, we may get a different search tree. For example, Figure 4.11 is also the search tree for the game of Figure 4.1 on page 30. But, only 29 nodes were expanded. Compare this to the 43 nodes expanded in Figure 4.10. The children of each node were ordered such that paths corresponding to good moves by each of the players are traversed first. The nodes are ordered according to rules developed by an expert bridge player.

We attempt to add an ordering to our nodes to realize the payoff of good ordering. A bridge expert must produce a set of rules to govern the ordering of children in the game tree. The bridge expert that developed the set of ordering rules applied here was Dr. David Mutchler of the University of Tennessee Computer Science Department. Mutchler is a Life Master in the American Contract Bridge League who competed actively in tournament competition for approximately five years. Though no longer active in tournament competition, Mutchler is probably among the top fifty bridge players in Knoxville, Tennessee.

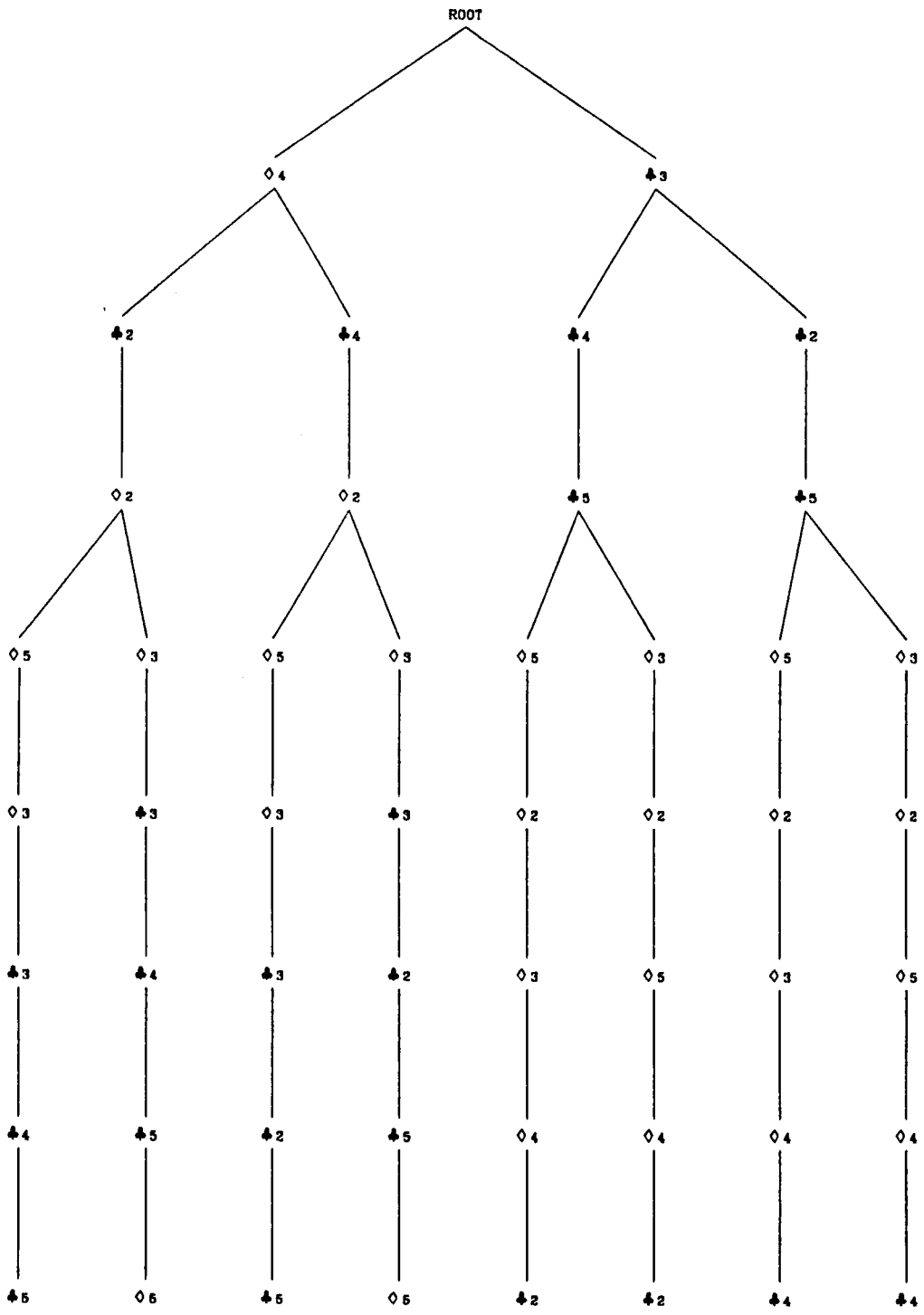


Figure 4.10: Example of Alpha-Beta (random) Search Tree with No Prunes

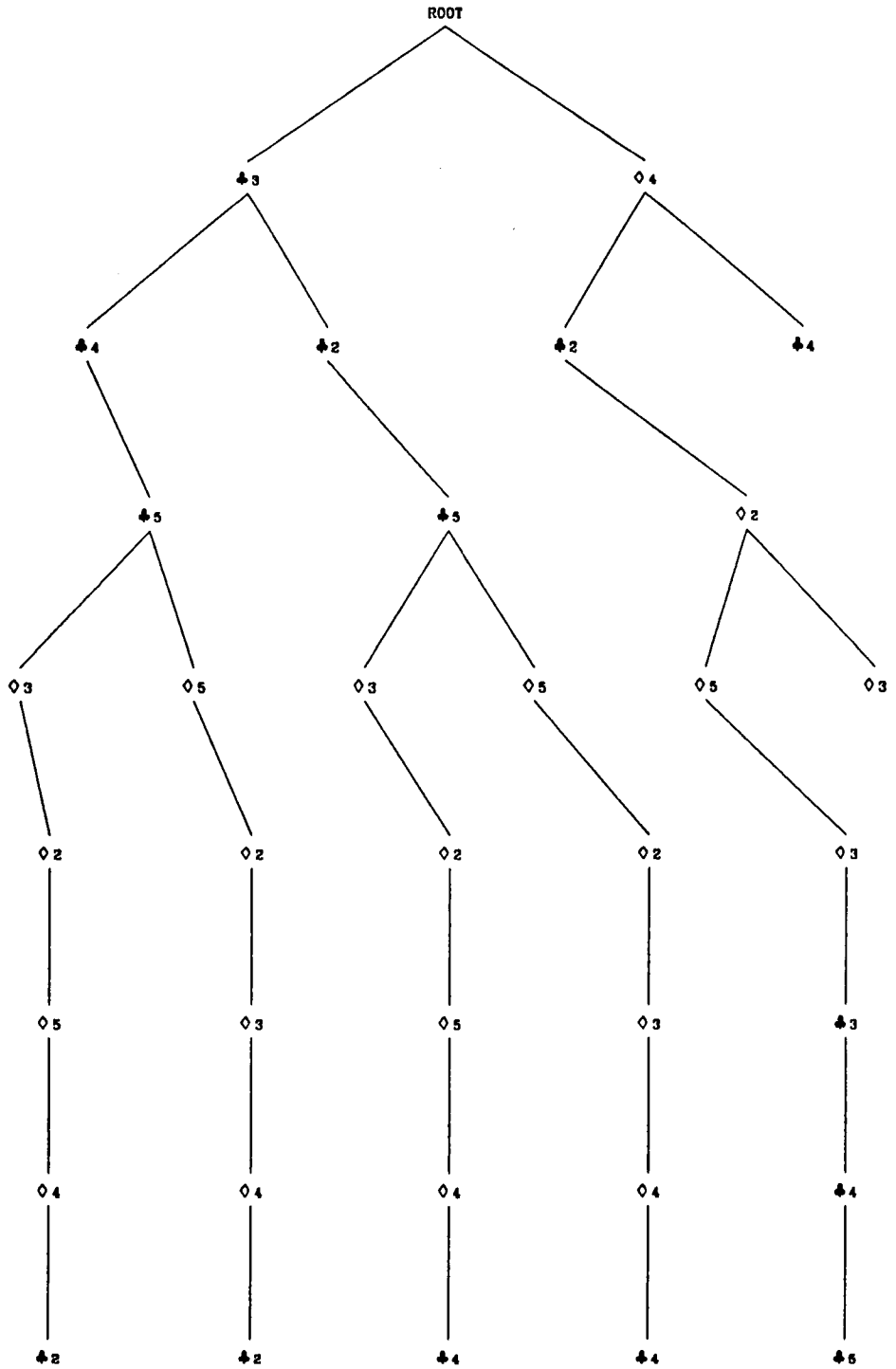


Figure 4.11: Example of Good Ordering Search Tree with Several Prunes

The ordering rules that Mutchler developed are of the the form "first hand low," and "second hand cover an honor with cheapest higher honor." The "first hand low" rule means that the leftmost child of the lead player of a trick should be the lowest card that the player has. The other children of the lead player are ordered according to further rules for the lead player of a trick. Similarly, rules were developed for the order of children for the second, third, and fourth player of the trick. The complete set of ordering rules and its relation to the game of bridge can be found in Appendix B.

The implementation of the good ordering technique consists of reordering the children of each node according to the rules developed by the expert. Initially, the children are a randomly ordered list of nodes. After reordering, the leftmost child is the best child according to the rules developed by the expert. Thus, the path with this child is traversed first since the search is a left to right depth first search. Implementation of the ordering rules consisted of adding a function which returned the children in their new "good" ordering. This function was implemented in approximately 300 lines of *Lisp* code. The function traverses the list of children, picks the next best child according to the rules, then appends this child onto the good ordered list. This is done until all children are in the good ordered list. The CPU time/node is increased due to this ordering. Note that it is possible that a different sorting technique could make this faster. The increase in CPU time at each node is a function of the number of children of that node and the number of times the rules require a pass through the original randomly ordered list.

We begin our analysis of the addition of good ordering to the alpha-beta technique by examining Table 4.4, the frontier development table for the alpha-beta technique with good ordering of nodes. The frontier is the same as the frontier for the alpha-beta search technique with random ordering of nodes, Table 4.2 on

Table 4.4: Frontier Development Table for Alpha-Beta with Good Ordering of Nodes Search Strategy

Number of Suits	Number of Cards						
	4	8	12	16	20	24	28
1	-	-	-	-	-		
2	-	-	-	-	-	-	
3			-			-	
4	-	-	-	-	-	-	

page 39. At first glance this is not encouraging. We must realize, however, that addition of this technique increases the CPU time/node. Thus we could expect our frontier to be worse for this technique than for the alpha-beta technique without good ordering.

Since our frontier is not in fact worse, we conclude that the addition of the good ordering allowed additional pruning that overcomes the increase in CPU time. Figure 4.12, Figure 4.13, Figure 4.14, and Figure 4.15 show the information for the average number of nodes expanded for the alpha-beta with good ordering of nodes technique as compared with the minimax technique and the alpha-beta with randomly ordered nodes technique. Table 4.5 summarizes the percentage decrease in the number of nodes expanded for the good ordering technique as compared to the alpha-beta strategy with random ordering of nodes.

Notice from Table 4.5 the percentage decrease in the number of nodes expanded ranges from 0% to 59%. Even with a 59% decrease in the number of nodes expanded the frontier cannot be expanded. This is due to the exponential increase in the size of the game tree as larger hands are attempted. Table 4.5 is evidence of the effect of good ordering. But, when the good ordering technique is compared to the basic minimax strategy in Table 4.6, the comparison is similar to Table 4.3 on page 43. Marsland and Popowich explain in [10] that it is difficult to develop

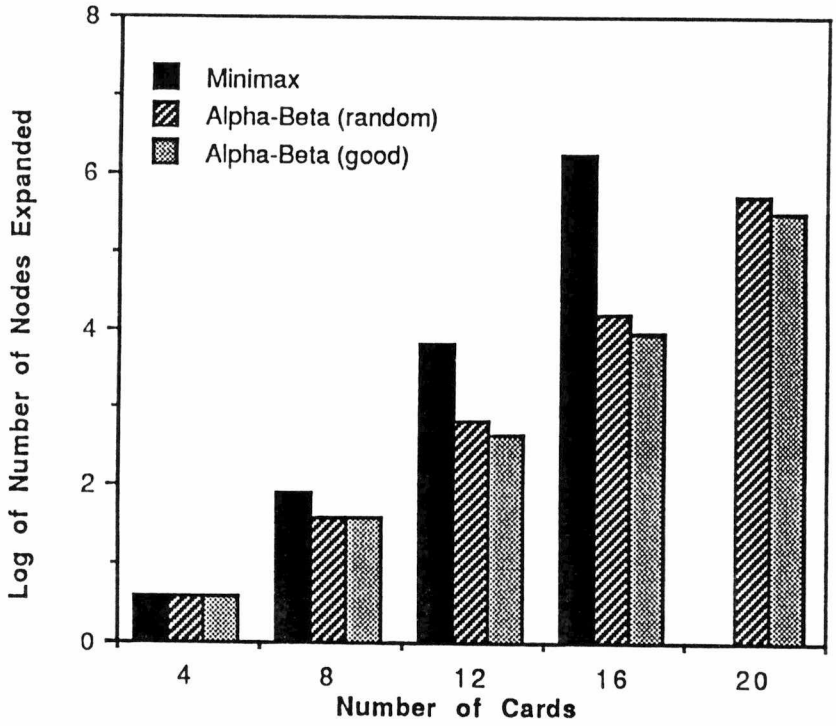


Figure 4.12: Alpha-Beta (good) vs. Previous Techniques, 1 suit games

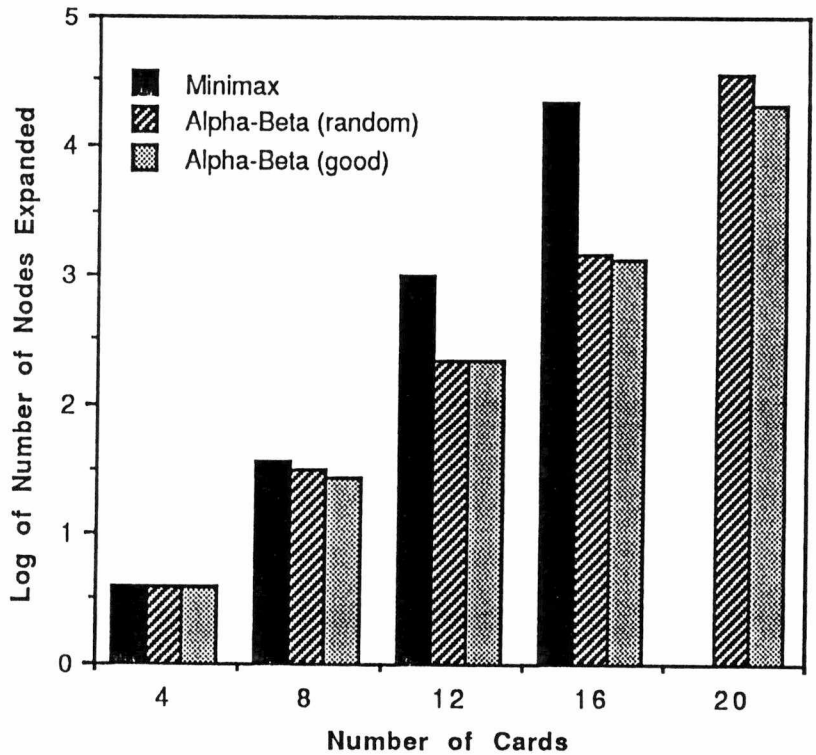


Figure 4.13: Alpha-Beta (good) vs. Previous Techniques, 2 suit games

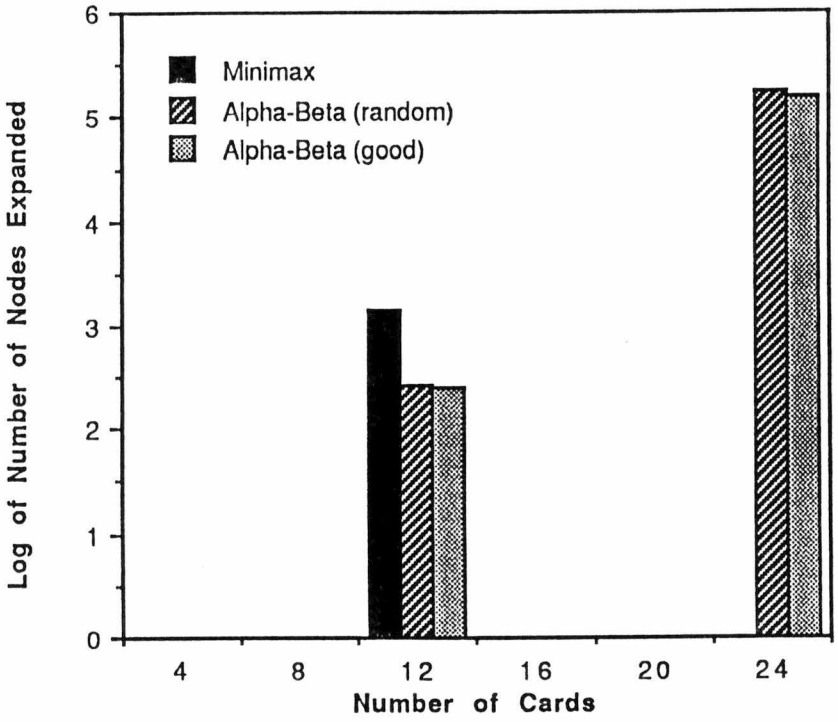


Figure 4.14: Alpha-Beta (good) vs. Previous Techniques, 3 suit games

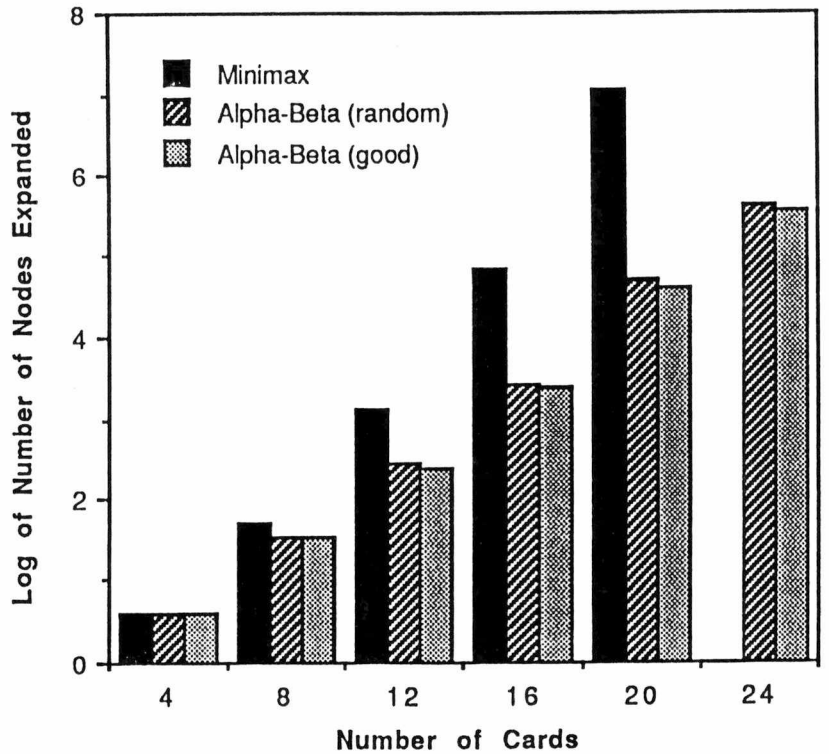


Figure 4.15: Alpha-Beta (good) vs. Previous Techniques, 4 suit games

Table 4.5: Game Tree Complexity Comparison for Alpha-Beta with Random Ordering of Nodes vs. Alpha-Beta with Good Order of Nodes

Number of Suits	Number of Cards In The Game	Number of Nodes Expanded
1	4	0%
1	8	5%
1	12	29%
1	16	43%
1	20	40%
2	4	0%
2	8	13%
2	12	2%
2	16	11%
2	20	43%
2	24	59%
3	12	13%
3	24	10%
4	4	0%
4	8	0%
4	12	18%
4	16	5%
4	20	20%
4	24	15%

Table 4.6: Game Tree Complexity Comparison for Minimax vs. Alpha-Beta With Good Ordering of Nodes Search Techniques

Number of Suits	Number of Cards In The Game	Number of Nodes Expanded
1	4	0%
1	8	51%
1	12	93%
1	16	99%
1	20	——
2	4	0%
2	8	25%
2	12	78%
2	16	94%
2	20	——
2	24	——
3	12	82%
3	24	——
4	4	0%
4	8	32%
4	12	83%
4	16	96%
4	20	99%
4	24	——

a mathematical model to estimate the complexity of search of a multibranch tree and that it is customary to use a uniform game tree as a model. With this is mind, we expected from [6] at best to search two times deeper and at worst to search no deeper than an exhaustive minimax search. Our results show that on average we were able to search 8/5 deeper than the exhaustive minimax search. This is only 1/10 deeper than we were able to search using alpha-beta with random ordering of nodes. The effect of good ordering of the nodes resulted in a smaller percentage decrease in the complexity of the game tree than was hoped for. An extension of this project that may prove valuable in decreasing the search is an analysis of the ordering rules of Appendix B. It is possible that better rules could be developed in order to realize more pruning in the game trees. Also, our expectations were based on results from studies of trees with a uniform branching factor, a category into which the trees for this project do not fall.

4.4 Strict Bucketing Technique

The strict bucketing technique uses facts specific to the game of bridge in order to reduce the search of the game tree. Many times a player will have several cards in his hand that are of equal value. For example, a player may have the ♣ 2 and the ♣ 3. Cards whose face value difference is one will be referred to as consecutive cards. Playing either of these cards will have the same effect on the game. Any trick that the player will lose by playing the ♣ 2 will also be lost by playing the ♣ 3. Similarly, any trick won by playing the ♣ 2 will also be won by playing the ♣ 3. After playing the ♣ 2, the player has the lowest ♣ of the ♣ cards not yet played in the game. This includes the ♣ cards in the hands of the other players. Similarly, if he plays the ♣ 3, he still has the lowest ♣ of the ♣ cards not yet played in the game. Thus, the two cards are of equal value to him.

A similar effect may occur when a player has two non-consecutive cards of a suit. But, the cards between the two non-consecutive cards must have already been played in a previous trick. For example, suppose a player has the ♣ 4 and the ♣ 6. Furthermore suppose that the ♣ 5 has been played in a previous trick of the game. Then, the ♣ 4 and the ♣ 6, though not consecutive cards, have consecutive ranks within the ♣ suit. Thus, playing either of the cards will have the same effect. Any trick that he will lose by playing the four will also be lost by playing the six. Similarly, any trick that he will win by playing the six will also be won by playing the four. If the five had been played in the current trick, then obviously the four and the six are not of equal value.

Since cards of this nature have the same effect on the game, only one branch needs to be taken in the search tree. This single branch accounts for the situation that occurs when either of the cards are played. We call groups of cards such as these a "bucket" of cards [4]. The previous minimax and alpha-beta techniques can be applied to any two-player zero-sum game. Strict Bucketing is different from the previous techniques because it is useful only to games where there exist moves that have the special "same effect" characteristic. Buckets may have any number of cards in them. A bucket is a group of cards with consecutive ranks. One branch will be taken for each bucket of cards.

Implementation of the strict bucketing technique is very simple. The `hands` array of the NODE structure described in Chapter 3 must be traversed to find the cards available for the current player. Instead of choosing every card that belongs to that player, we choose only a representative card from each bucket. Since all cards in a bucket have equal value, we choose the lowest card of the bucket since the lowest was easiest to get with our implementation. We look through the `hands` array for the first card of a suit that the current player has, add this card to the player's list of possible plays, then look for a card which a different player has.

All cards that the current player has whose value is between the first card found and the card belonging to a different player belong in the same bucket. We again look for the next card that the current player has. When found, this card is the representative card for the next bucket. If the player has cards which can be bucketed, we end up with a smaller list of possible cards to play since only one representative card from each bucket is considered.

Figure 4.16 is a game of two suits with eight cards in which the strict

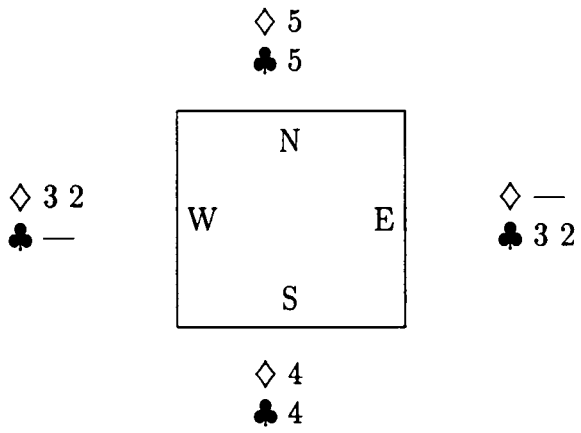


Figure 4.16: Game of Two Suits with Eight Cards Corresponding to the Game Trees in Figure 4.17 and Figure 4.18

bucketing provides excellent reduction of the game tree. The game in Figure 4.16 could be analyzed by a human bridge player in seconds. In particular, he would not consider the two cards available for West to lead as two different possibilities. It is obvious that the same result will come out of a game where West leads the ♢ 2 as a game where West leads the ♢ 3. This is the reduction that the strict bucketing technique implements. Figure 4.17 is the search tree for the game represented by Figure 4.16 when the alpha-beta with good ordering of nodes technique is used. There are 25 nodes expanded in this tree. Figure 4.18 is the search tree when

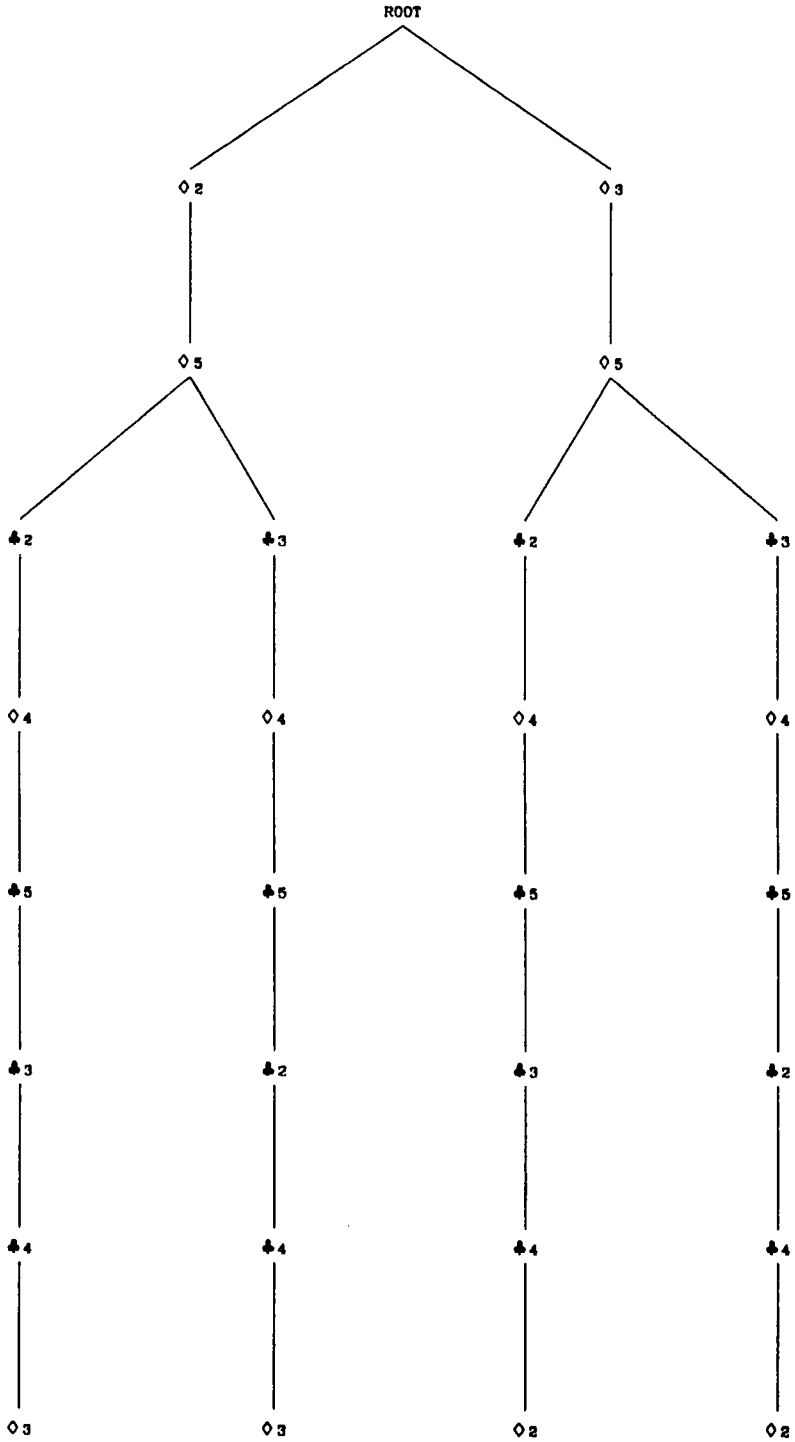


Figure 4.17: Good Ordering Search Tree of Figure 4.16

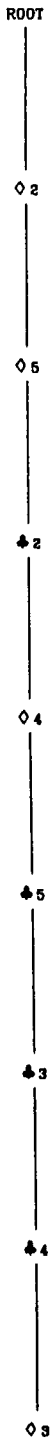


Figure 4.18: Strict Bucketing Search Tree of Figure 4.16

strict bucketing is added to the alpha-beta with good ordering of nodes technique. Notice that only eight nodes are expanded. Only one branch needs to be taken by West when he leads. Similarly, only one branch needs to be taken by East in the first trick since his ♣ 2 and ♣ 3 are in one bucket.

Addition of strict bucketing to the alpha-beta with good ordering of nodes technique was successful in reducing the search of game trees. From Table 4.7 we see that in a game of four suits with 24 cards an average of 97% less nodes were expanded. Figure 4.19, Figure 4.20, Figure 4.21, and Figure 4.22 graphically demonstrate the average decrease in the number of nodes expanded as compared to the alpha-beta with good ordering of nodes for one, two, three, and four suit hands respectively. We can see from Table 4.8, the frontier table for the strict bucketing technique, that we can get results for games of four suits with up to 28 cards in less than ten hours. Positions with *'s indicate expansion due to the addition of the strict bucketing technique. Previous techniques have only allowed us to get results for suits of four cards with up to only 24 cards in less than 10 hours.

4.5 Internal Cutoffs Technique

The internal cutoff technique allows for additional prunes of the search tree at depths of the game tree corresponding to the end of a trick. At the end of a trick, the number of tricks won so far by the N/S team on the current path is known. Also, the number of tricks left to play in the game is known. If we add these two numbers together, we have an upper bound on the number of tricks that the N/S team can possibly win on the current path. If this number is less than or equal to the number of tricks that the N/S team are assured of winning due to the value of ancestor nodes, there is no need to proceed with the search of the

Table 4.7: Game Tree Complexity Comparison for Strict Bucketing vs. Alpha-Beta With Good Ordering of Nodes Search Techniques

Number of Suits	Number of Cards In The Game	Number of Nodes Expanded
1	4	0%
1	8	13%
1	12	36%
1	16	57%
1	20	72%
2	4	0%
2	8	26%
2	12	48%
2	16	61%
2	20	81%
2	24	82%
3	12	45%
3	24	81%
4	4	0%
4	8	15%
4	12	34%
4	16	57%
4	20	75%
4	24	97%
4	28	——

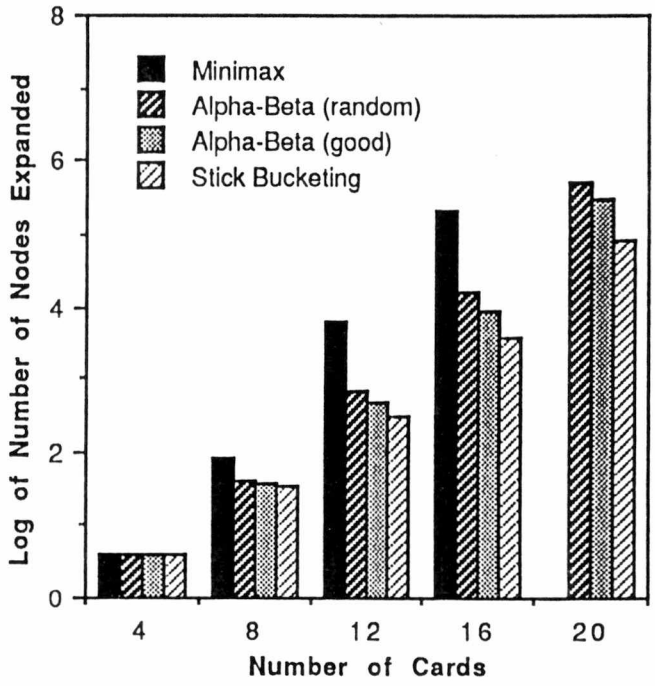


Figure 4.19: Strict Bucketing vs. Previous Techniques, 1 suit games

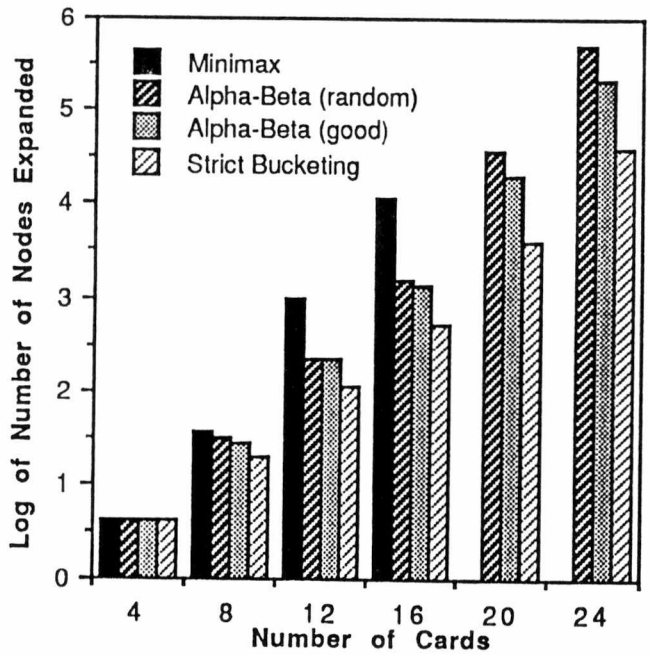


Figure 4.20: Strict Bucketing vs. Previous Techniques, 2 suit games

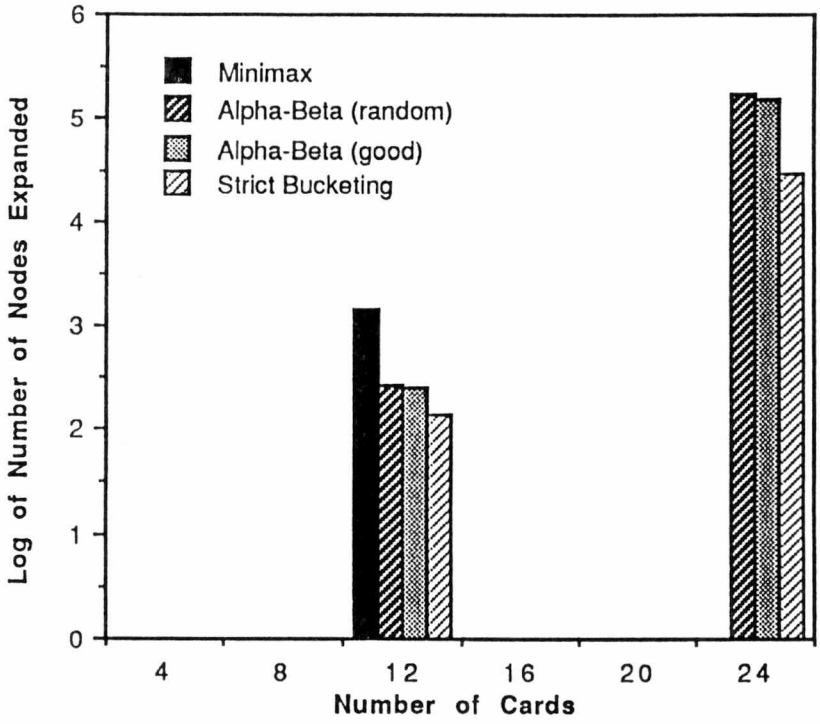


Figure 4.21: Strict Bucketing vs. Previous Techniques, 3 suit games

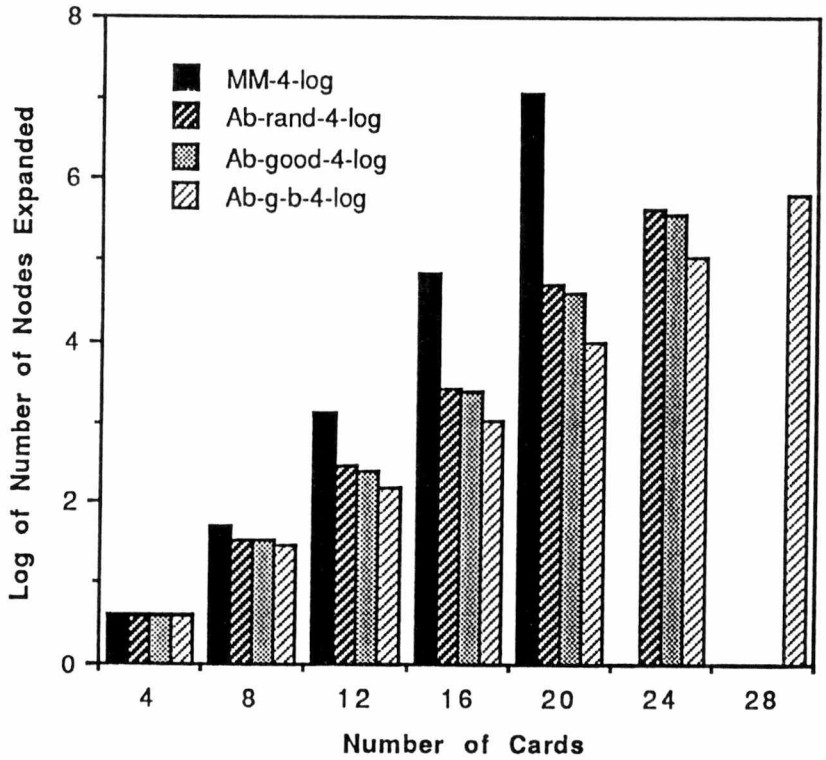


Figure 4.22: Strict Bucketing vs. Previous Techniques, 4 suit games

Table 4.8: Frontier Development Table for Strict Bucketing Search Strategy

Number of Suits	Number of Cards							
	4	8	12	16	20	24	28	32
1	-	-	-	-	-			
2	-	-	-	-	-	-		
3			-			-		
4	-	-	-	-	-	-	*	

children of the current node. Similarly, if the number of tricks won on this path so far is greater than or equal to the number of tricks that the opposing team can hold the N/S team to, there again is no need to proceed with the search of the children of the current node.

The values used in the comparison to determine if internal cutoffs are possible are the threshold values ALPHA and BETA that are kept to allow the alpha-beta prunes discussed in section 4.2 to take place. ALPHA is a lower bound on the number of tricks that the N/S team may take. BETA is the upper bound on the number of tricks that the N/S team may take. Pseudo code for the internal cutoffs algorithm is:

```
if  (num-tricks-won-so-far >= BETA)
    return (num-tricks-won-so-far)
else if  (num-tricks-possible-to-win  <= ALPHA)
         return (num-tricks-possible-to-win)
     else proceed as normal with search of the
                          children of this node
```

Only nodes at the end of a trick, every fourth level in the game tree, need to execute the code that checks if internal cutoffs are possible. This is because the

number of tricks won on a path does not change until the full trick has been played. Notice in Table 3.2 on page 19 that there is only a small increase in the CPU time per node upon addition of the internal cutoffs test.

Figure 4.23 is a game of four suits with 12 cards where the addition of

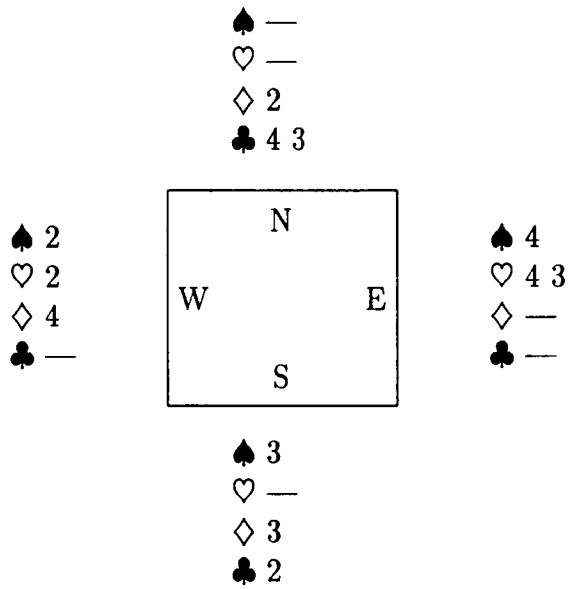


Figure 4.23: Game of Four Suits with Twelve Cards Corresponding to the Game Tree in Figure 4.24 and Figure 4.25

internal cutoffs reduced the complexity of the game tree search. Figure 4.24 is the search tree using the alpha-beta with good ordering of nodes and strict bucketing search strategies. Upon adding the internal cutoffs code, the search was reduced to the tree in Figure 4.25. With the internal cutoffs addition, only 26 nodes were expanded. Without the internal cutoffs code, 73 nodes had to be expanded. Compare the trees of Figure 4.24 and Figure 4.25 to see the 26 and 73 nodes that were expanded in the respective trees. Consider the leftmost internal cutoff that occurred at a depth of four in the search tree of Figure 4.25. The path to this point in the game consists of the ◇ 4, ◇ 2, ♠ 4, and the ◇ 3. The value of ALPHA

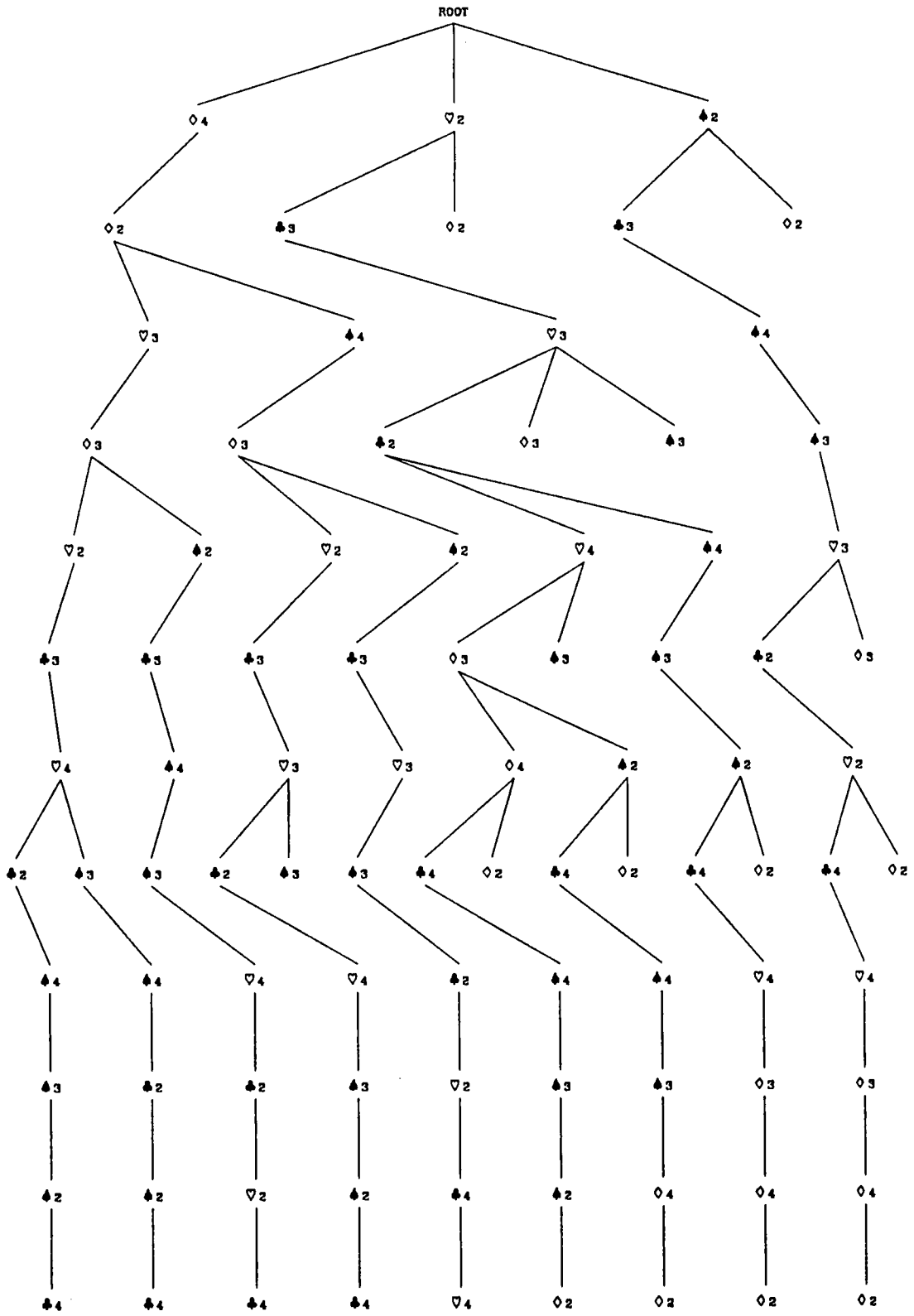


Figure 4.24: Search Tree of Figure 4.23 after Addition of Strict Bucketing Technique

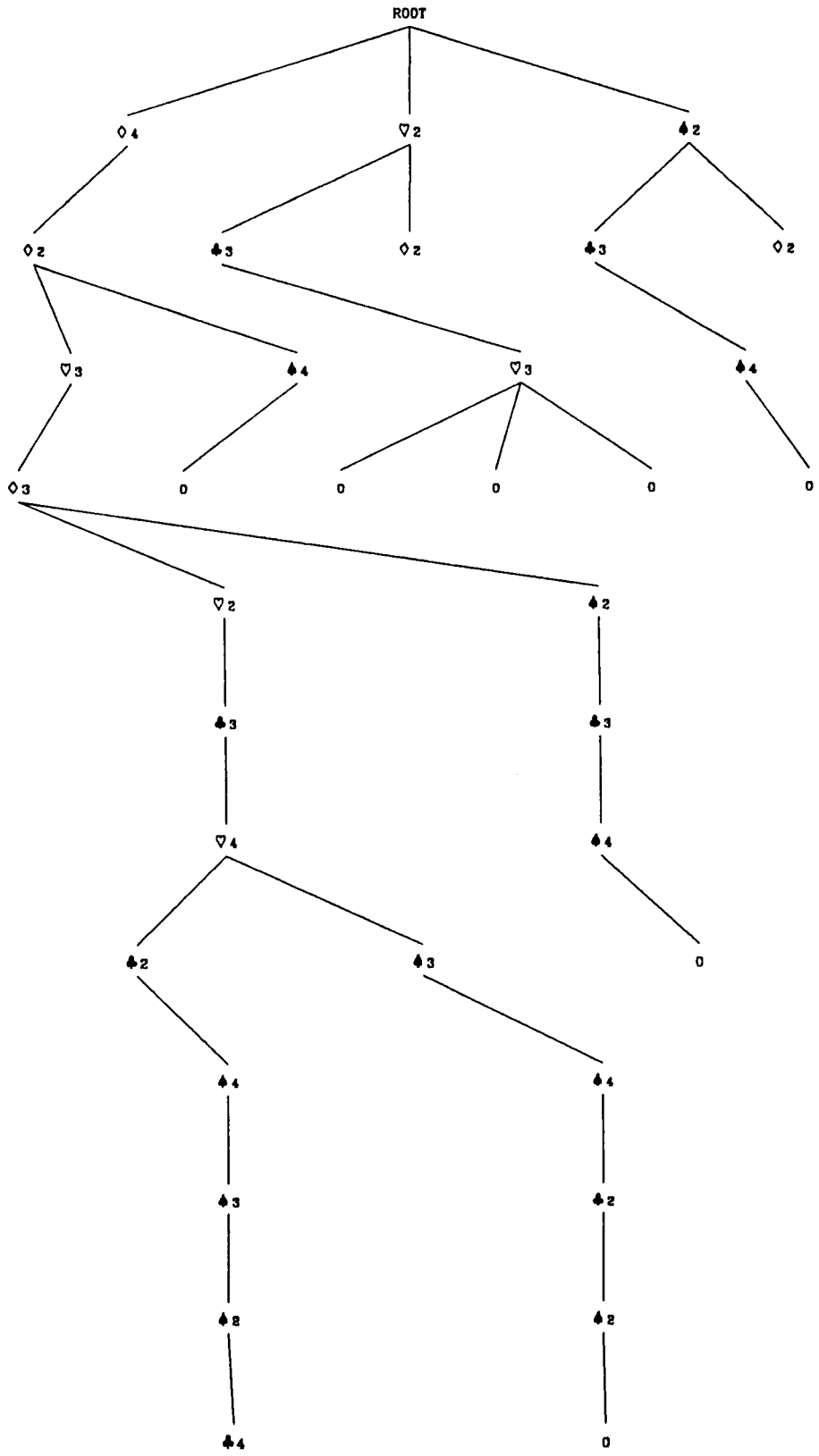


Figure 4.25: Search Tree of Figure 4.23 after Addition of Internal Cutoffs Technique

at the ♢ 3 card is -100 and the value of BETA is zero due to the previous paths searched. West wins the first trick on our current path with his ♢ 4. Therefore the number of tricks won so far by the N/S team on this path is zero. If we examine the above rules for internal cutoffs we see that if the number of tricks won so far is greater than or equal to BETA, then we do not need to search the subtree from this node. Since the number of tricks won so far on this path and BETA are both zero, we merely return zero as the bound on the minimax value of this node. There are several other positions in the search tree of Figure 4.25 that reflect internal cutoffs. Obviously internal cutoffs have greatly decreased the complexity of the search for the game of Figure 4.23.

Table 4.9 show us the average percentage decrease in the number of nodes expanded that occurs with the addition of internal cutoffs. In particular, note the 80% decrease in the number of nodes expanded in games of four suits with 28 cards. Though an 80% decrease is a lot, the frontier for the internal cutoffs technique remains at 28 cards for games of four suits. Table 4.10 shows the frontier when the internal cutoffs technique is added. We were not able to expand the frontier upon addition of the internal cutoffs technique. We must consider that as the number of cards in our games increases, there is an exponential explosion in the number of nodes that must be expanded in the game tree. Recall Figure 4.3 on page 34 that shows the explosion when only the minimax strategy with no pruning is used. Figure 4.26, Figure 4.27, Figure 4.28, and Figure 4.29 graphically demonstrate the average decrease in the number of nodes expanded with the addition of the internal cutoffs pruning technique. Though our frontier did not expand with the addition of internal cutoffs, Table 4.9 is evidence that internal cutoffs provide valuable pruning in the game trees.

Table 4.9: Game Tree Complexity Comparison for Internal Cutoffs vs. Strict Bucketing

Number of Suits	Number of Cards In The Game	Number of Nodes Expanded
1	4	0%
1	8	29%
1	12	46%
1	16	49%
1	20	57%
2	4	0%
2	8	25%
2	12	33%
2	16	49%
2	20	47%
2	24	73%
3	12	47%
3	24	60%
4	4	0%
4	8	24%
4	12	76%
4	16	53%
4	20	55%
4	24	58%
4	28	80%

Table 4.10: Frontier Development Table for Internal Cutoffs Pruning Strategy

Number of Suits	Number of Cards							
	4	8	12	16	20	24	28	32
1	-	-	-	-	-			
2	-	-	-	-	-	-		
3			-			-		
4	-	-	-	-	-	-	-	

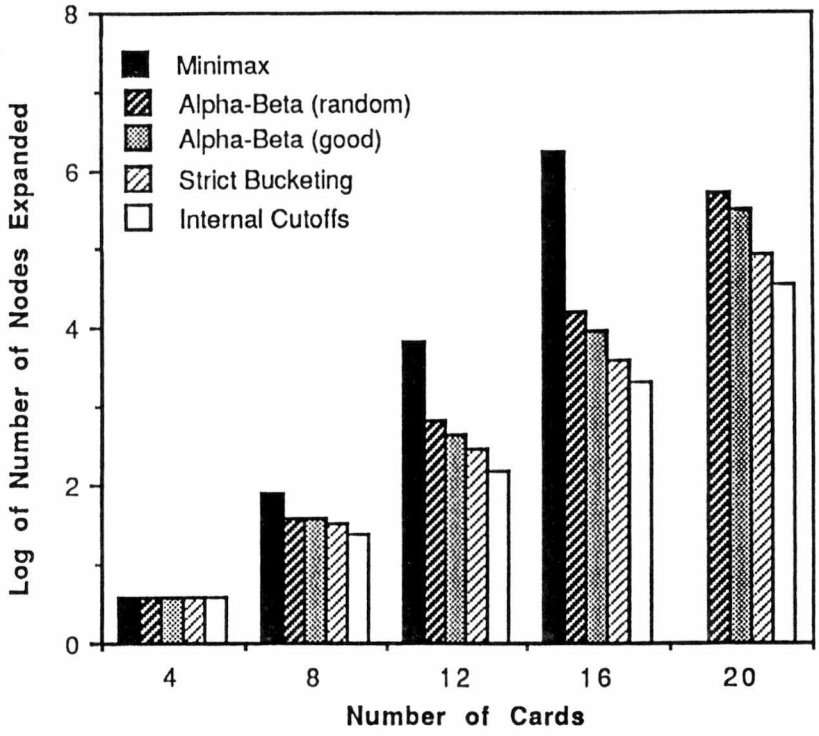


Figure 4.26: Internal Cutoffs vs. Previous Techniques, 1 suit games

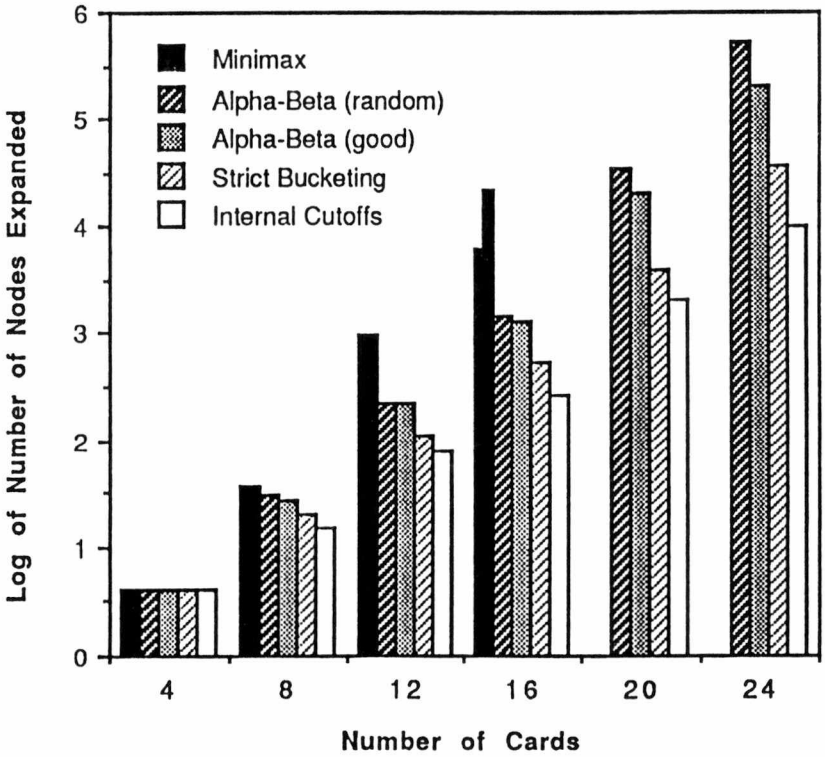


Figure 4.27: Internal Cutoffs vs. Previous Techniques, 2 suit games

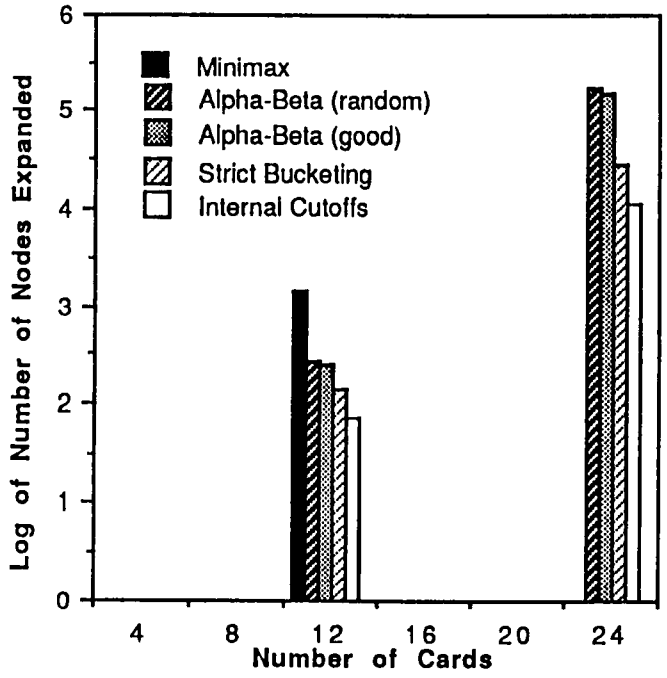


Figure 4.28: Internal Cutoffs vs. Previous Techniques, 3 suit games

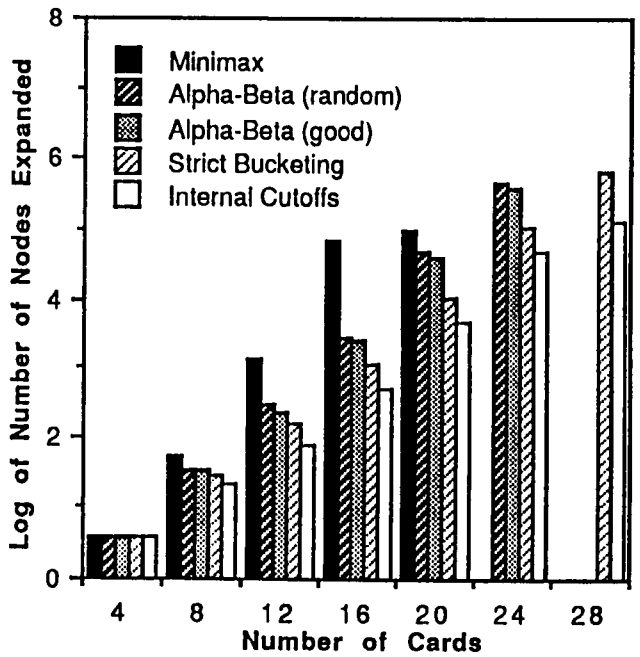


Figure 4.29: Internal Cutoffs vs. Previous Techniques, 4 suit games

4.6 Deferred Branching Technique

4.6.1 General Idea of Deferred Branching

The deferred branching technique is an extension of the strict bucketing technique. This new type of bucketing enables less branching near the top of the search tree. Instead of single cards being played, buckets of cards are played. In the worst case, each of our buckets would have a single card in them. Cards are kept grouped together in buckets until a game situation forces the distinction of cards in the buckets. We do not choose a representative card from the bucket to play as was done in the strict bucketing technique. We keep up with cards played by keeping up with the buckets that have been played, which cards belong to the played buckets, and how many times a card has been in a bucket that has been played. A new set of bucketing rules extends the rules of strict bucketing. An elaborate algorithm for keeping up with cards and their respective buckets was developed.

The goal of the new bucketing scheme is to maximize the bucketing at each node, thereby reducing the number of children and hence branches at each node. The following rule is followed:

> **Any group of a player's cards that will have the same effect on the outcome of the current trick are put into the same bucket.**

There is one exception to this rule that will be discussed later. With this bucketing rule, we can still determine the information we need in order to proceed with the game. In particular, we can determine the winner of the trick. In fact, combining any two buckets formed according to the above rule prevents the determination of information needed in order to proceed with the game. For example, suppose that South has the ♣ 4 and ♣ 6 and that East has the ♣ 5. Further suppose that

East led the trick and played a ♣ card other than the ♣ 5. Now, it is South's move. Under strict bucketing, South would not bucket his ♣ 4 and ♣ 6. Unlike strict bucketing, the extended bucketing takes cards played thus far in the trick into consideration. South's four and six can be bucketed because they will have the same effect on the outcome of the current trick.

The cards are bucketed according to three categories. First, any cards that will win the current trick are bucketed together into the definite winner's bucket denoted DEF-WIN. Second, any cards that definitely will not win the current trick are bucketed together into the non-winner's bucket denoted DEF-DONT-WIN. Third, any cards that do not fall into either of the first two categories are bucketed according to the rules of strict bucketing where cards that have already been played by other players in the current trick are taken into account. For each bucket created, one of the following statements is true: i) This bucket will win the trick; ii) This bucket will not win the trick; or iii) The fate of this bucket as winner of the trick is to be determined by the future cards played in this trick.

Suppose that West is about to make the first play of a new game of four suits with 12 cards. Furthermore suppose that he intends to play from his ♣ suit of which he has the Q, K, and Ace. Since any of his three cards will win the trick, all three cards are put into a single bucket. Thus instead of three branches in the search tree, there is only one branch corresponding to West leading his ♣ suit. As in the strict bucketing technique, the cards are marked in such a way as to denote that any of the three may have been played. Suppose the other three players each have three cards belonging to the ♣ suit. North is the next player but none of his cards are capable of altering the outcome of the trick. North's ♣ cards fall into the category of a DEF-DONT-WIN bucket regardless of how consecutive they are. Similarly, both East and South have only one bucket of ♣ cards, a DEF-DONT-WIN bucket.

For each of the players, we must mark the cards so that further in the game, we will know that one of their ♣ cards has been played. Since it has not mattered yet which of the ♣ cards has been played from each hand but may matter later, we merely store the information that one of them, not any particular one, has been played from each hand. Since West won the first trick, he must now lead the next trick. Suppose West again leads from his ♣ suit. The same sequence of events as in the first trick will take place. West will win the trick because the other players do not have a card larger than West's ♣ cards. We keep up with the information that West has played a second card from his ♣ suit and that each of the other three players have played a second card from their ♣ suit. Once more, West leads. This is the third time and he originally had only three cards from the ♣ suit. He will win the trick with bucketing occurring just as in the previous two tricks. The three cards in his ♣ suit must now be marked as played. He has played a bucket of three cards three times. The other players' ♣ cards must also be marked as played.

In the above example, three tricks which correspond to twelve levels in the game tree were played with the search tree containing only one path for the play of the ♣ suit. An exhaustive search would result in $(3!)^4$ paths for the play of the ♣ suit. The above example is a very simple case of the bucketing procedure. Many times it may occur that buckets overlap. That is, in one trick a bucket of three cards may be played. In the next trick, it is possible that only two of these cards are bucketed together. Then, we must have a system to keep up with the information that two of a group of three cards has been played and one of those two must be from a particular sub-group of the three cards. Similarly we may run into a situation where a large bucket is played that encompasses some previously played buckets. We must be able to tell when a card has been played due to the different buckets it has belonged to.

Thus far we have only discussed bucketing when all players have cards of the lead suit. When a player is forced to discard, a bucket is formed for each of the suits he has. Since any card he plays will lose, the bucketing rules indicate that all his cards be bucketed together regardless of suit. In order to simplify implementation of this technique, cards of different suits are put into different buckets. This is the only exception to the rule of putting all cards that have the same effect on the current trick into the same bucket. If a player has cards in three suits none of which are the lead suit, then three branches corresponding to each of the three suits the player may choose to discard from are formed in the search tree.

The following describes the bucketing algorithm:

```
if current player is not able to follow suit
   create discard buckets
else
  set x = max (cards that have been played thus far
               in the current trick, max card that future
               players in the trick could play)
  set y = max (minimum card of each of the previous players of the
               current trick, minimum card that future
               players in the trick could play)
  put all cards the player has that are less than y into
               the DEF-DONT-WIN bucket
  create buckets with extended strict bucketing rules for cards
               whose value is between the value of the y and x card
  put all cards the player has that are higher than x into
               the DEF-WIN bucket.
```

4.6.2 A Possible Problem with Deferred Branching

A question dealing with the deferred branching technique has been proposed for which no immediate and/or simple solution arises. To motivate the problem, we examine a portion of the search tree of the game of Figure 4.30. Figure 4.31 is a portion of the search tree produced using the Alpha-Beta with

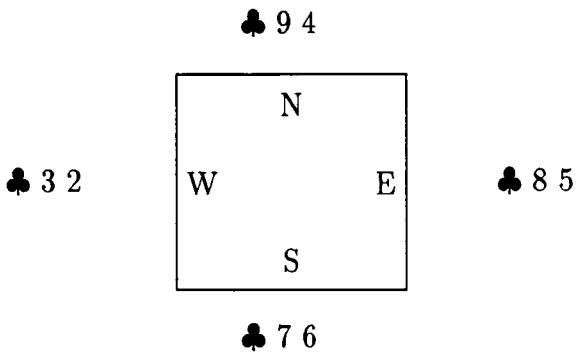


Figure 4.30: Game Used for Examination of the Possible Problem with Deferred Branching

good ordering and Strict Bucketing techniques. Figure 4.32 is the corresponding portion of the search tree produced using the Alpha-Beta with good ordering and the Deferred Branching Technique. At depth 1 of Figure 4.31 only the ♣ 2 is shown because with strict bucketing, the least card of the bucket is chosen for play. Compare this with the 2 and 3 of ♣ shown at depth 1 of Figure 4.32. At depth 3 of Figure 4.31 there are two branches corresponding to the 5 and 8 of ♣ being played. Notice that in Figure 4.32 the 5 and 8 are bucketed together. In particular, they are bucketed in a DEF-DONT-WIN bucket. Satisfying the alpha-beta cutoff criteria eliminated the need to search the subtree of the ♣ 8 at depth 3 in Figure 4.31. Notice also the "MAX" and "MIN" on the trees indicating the action at that level of the search tree. At depth 4, only the ♣ 6 is shown in

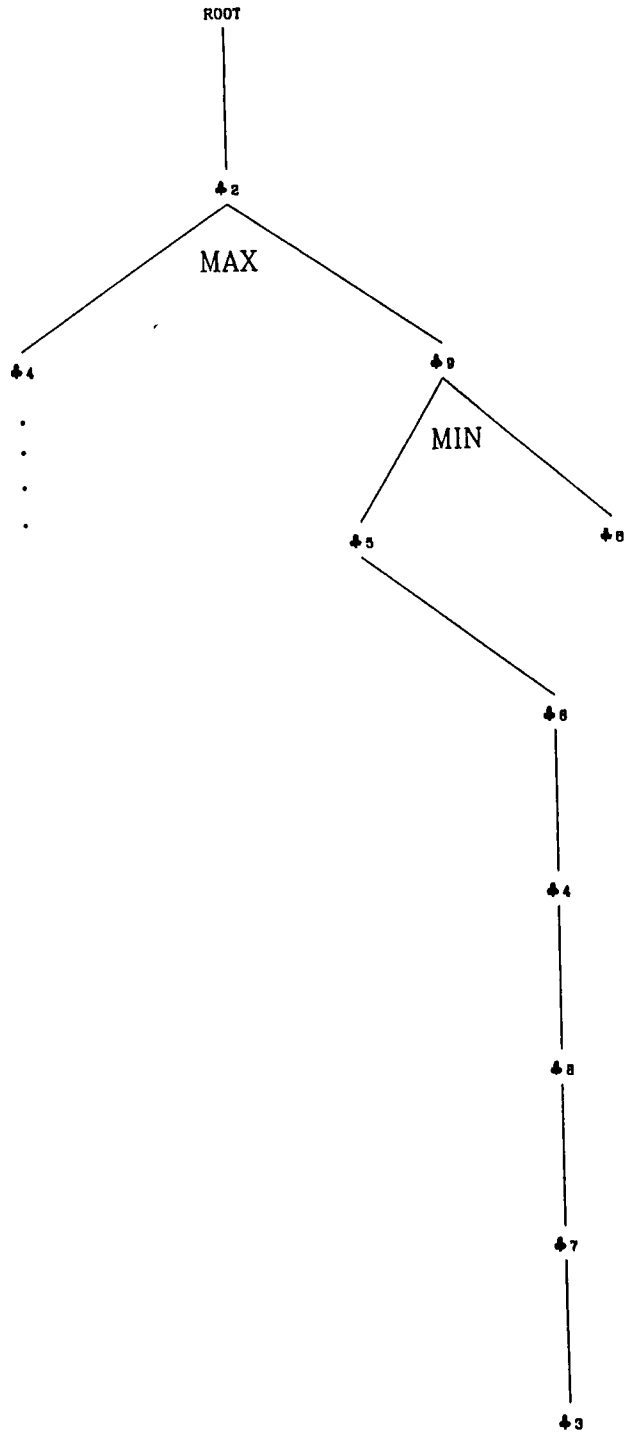


Figure 4.31: Search Tree of Figure 4.30 using Alpha-Beta (good) and Strict Bucketing Techniques

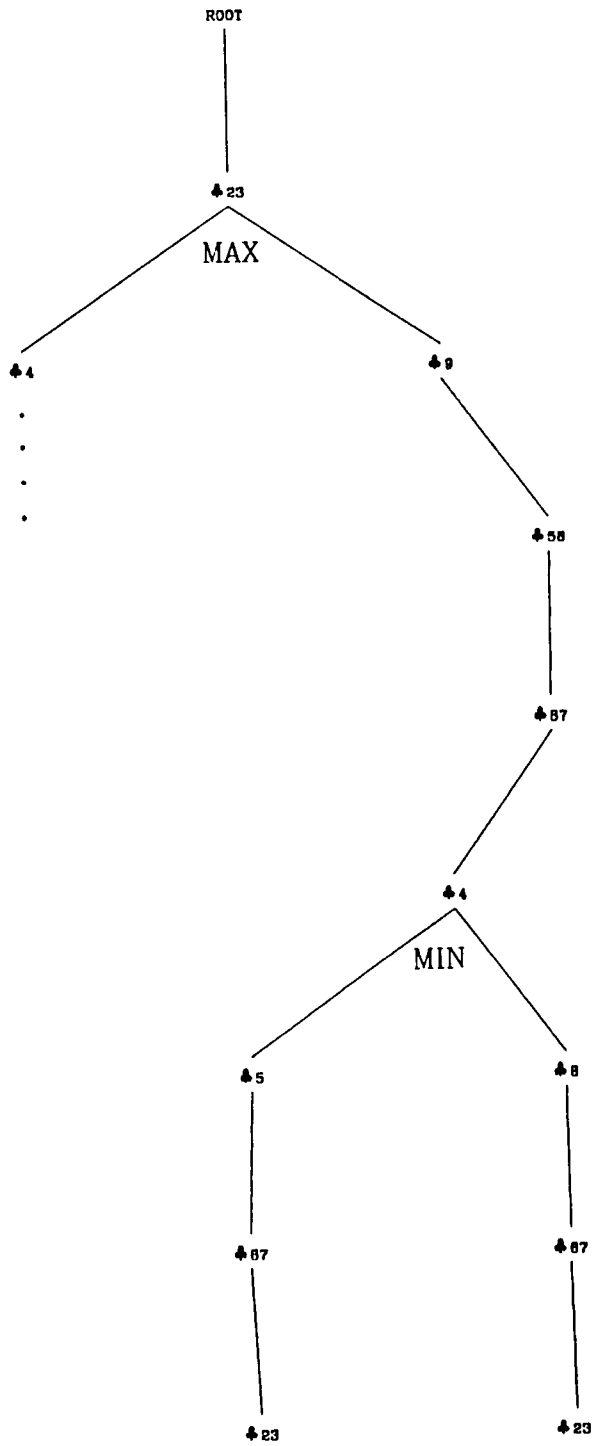


Figure 4.32: Search Tree of Figure 4.30 using Alpha-Beta (good) and Deferred Branching Techniques

Figure 4.31 unlike the 6 and 7 of ♣ in Figure 4.32. This is because of the strict bucketing choosing the least card in the bucket for play. At level 6 of Figure 4.32, the 5 and 8 of ♣ which had previously been bucketed together must now be separated to determine the outcome of the trick. Notice again the "MIN" at this level indicating the action being taken. The 6 and 7 of ♣ bucket on the left path at depth 7 of Figure 4.32 is a DEF-WIN bucket, but it is a DEF-DONT-WIN bucket on the rightmost path. Both buckets containing the 2 and 3 of ♣ at level 8 of Figure 4.32 are DEF-DONT-WIN buckets.

A simple eyeball comparison of the two trees shows us that deferred branching alters placement of the "MAX" and "MIN" areas of the search tree. In fact, the motivation for the technique is to defer branches to lower parts of the search tree. We can think of the search process as a function containing minimizing and maximizing subfunctions. The deferred branching technique alters the placement of the minimizing and maximizing subfunctions. For example, the minimization that occurs at depth two of Figure 4.31 has been deferred until depth five of Figure 4.32. The question posed is: "Could the shuffling of "MINs" and "MAXs" due to deferred branching cause the value of the function to be different than the value of the function before shuffling?" This question arises because we know that shuffling the order of "MINs" and "MAXs" within a function does not always preserve the function. For example, MIN (MAX a b) is not equal to MAX (MIN a b) for all a and b.

If the answer to the question in the previous paragraph is yes, then the deferred branching technique is a heuristic. If the answer is no, then a proof exists to show correctness. If, in fact, the deferred branching technique is a heuristic, our results show that it is a very good heuristic. No cases were found within the approximately 290 games tested where the deferred branching technique produced a result different than the exhaustive minimax search. Also, a bridge ex-

pert was unable, after many hours of searching, to find an example for which the deferred branching technique produces a different result than the exhaustive minimax search. Note that throughout this thesis, the deferred branching technique is not spoken of as a heuristic. Further analysis is needed to fully satisfy this problem but is beyond the scope of this thesis.

4.6.3 Implementation

The information about buckets is kept in an array called "buckets." This array was added to the NODE structure described in Chapter 3. The `buckets` array is the same size as the `hands` array. Each element of the `buckets` array directly corresponds to the element in the `hands` array with the same indices. An element of the `buckets` array contains two pieces of information. The first piece of information, LENGTH, is the number of cards in the bucket. For example, if the LENGTH were five, we would know that this card is in a bucket with a total of five cards. The second piece of information, COUNT, is also a number. COUNT is the number of cards that have been played from this bucket. For example, if the COUNT were three, then we would know that three of the five cards in the bucket have been played. We do not distinguish which of the three have been played because they are in the same bucket. Thus, playing any one of the cards has the same effect as playing any of the other cards. The two pieces of information about a particular bucket are kept in the element of the `buckets` array that corresponds to the largest card in the bucket.

Consider a game of two suits with 16 cards where West has the highest two cards of the ♣ suit. Assume the cards are dealt as described in Figure 4.33.

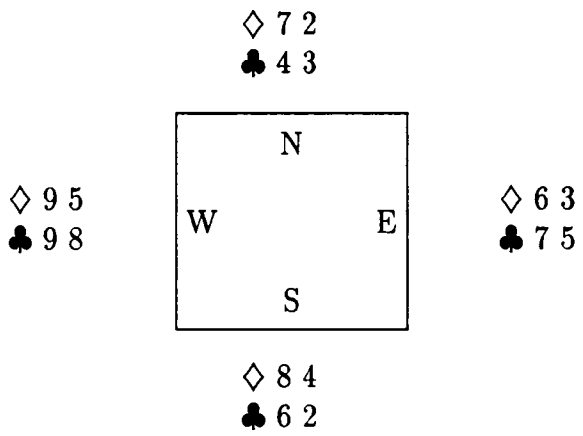


Figure 4.33: Game of Two Suits with Sixteen Cards

The following is the hands array of the NODE structure that corresponds to Figure 4.33.

suits	cards:	2	3	4	5	6	7	8	9
Clubs		S	N	N	E	S	E	W	W
Diamonds		N	E	S	W	E	N	S	W

Before search begins, the buckets array of the NODE structure is initialized to all zeros as below.

suits	cards:	2	3	4	5	6	7	8	9
Clubs		0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0
Diamonds		0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0

West has the choice of leading from the ♣ suit or the ♢ suit. Let's consider the subtree formed when West leads from the ♣ suit. Since West will win no matter which ♣ card he plays, a bucket is played containing the two ♣ cards. North buckets both of his ♣ cards since no matter which he plays, he will not win the trick. Similarly, East and South each play a bucket containing their respective two cards in the ♣ suit. Since we are now at the end of the trick, the `buckets` array must be marked to indicate which buckets were played. The updated `buckets` array looks as follows:

```
suits    cards:   2     3     4     5     6     7     8     9
-----          ---------------------------------------------
Clubs          | 0 0 | 0 0 | 2 1 | 0 0 | 2 1 | 2 1 | 0 0 | 2 1 |
Diamonds       | 0 0 | 0 0 | 0 0 | 0 0 | 0 0 | 0 0 | 0 0 | 0 0 |
                ---------------------------------------------
```

The two numbers found at Clubs 9 in the `buckets` array implies that one card of a bucket of length two has been played with the largest in the bucket being the nine of Clubs. The other card in this bucket is the next card of lower value than nine that belongs to the same player as the nine of Clubs. This is the eight of Clubs. Similarly, the other markings in this `buckets` array indicate the buckets played by the other players.

West again has the lead with a choice of playing from the ♣ suit and the ♢ suit. Again, let's look at the path on which West leads from his ♣ suit. West again has a bucket containing the eight and nine of ♣ available for play. The other players also play the same buckets as in the first trick since none of their cards are able to affect the outcome of the trick. Since each player has played a bucket of two cards twice, all cards in the buckets must now be marked as played. The played cards are marked with P's in the `buckets` array. The `hands` array is also

updated to denote played cards. After the second trick on this path, the **buckets** array looks as follows:

suits	cards:	2	3	4	5	6	7	8	9
Clubs		P P	P P	P P	P P	P P	P P	P P	P P
Diamonds		0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0

The **hands** array is also updated as follows:

suits	cards:	2	3	4	5	6	7	8	9
Clubs		S (P)	N (P)	N (P)	E (P)	S (P)	E (P)	W (P)	W (P)
Diamonds		N	E	S	W	E	N	S	W

Marking the **buckets** array is particularly complex when buckets are played that overlap into previously played buckets. One type of overlap occurs when the same bucket is played a second time. When this occurs, we remark the largest card of the bucket to indicate that two cards instead of one have now been played from this bucket. Also, we mark the card in the bucket that has the second highest value. This card is marked to indicate that one card from a bucket whose size is one smaller than the actual bucket played has been played. For example, suppose a bucket containing the four, five, and six of ♣ is played. The element in the **buckets** array corresponding to the six of ♣ is marked with LENGTH = 3 and COUNT = 1 to indicate that one of three cards has been played. Suppose that the same bucket is played again. The **buckets** array reflects this by remarking

the ♣ 6 with LENGTH = 3 and COUNT = 2 to indicate that two of three cards has been played. Also, the element in the **buckets** array corresponding to the ♣ 5 is marked with LENGTH = 2 and COUNT = 1 to indicate that one of two cards has been played.

4.6.4 Analysis

With the extended bucketing rules and a new procedure for keeping up with cards that have been played and cards that could have been played, we are able in most cases to reduce greatly the branching in the game tree. Cards are kept bucketed until it is necessary to single out particular cards in the play of the game. This causes larger buckets near the top of the game tree resulting in less branching. As we get deeper in the game tree, particular cards must be singled out. Thus, we have more buckets with fewer cards in each bucket. This is the reason the name deferred branching is given to this technique. With less branching near the top of the tree, the complexity of the game tree is decreased.

The algorithm for marking the **buckets** array is complex. Approximately 230 lines of *Lisp* code were needed. A detailed description of this algorithm can be found in Appendix C.

The goal of the deferred branching technique was to optimize the amount of bucketing in order to realize a minimal number of expanded nodes in the game tree. Using the new bucketing rule, we conjecture that the deferred branching technique will not increase the number of expanded nodes. This proves to be a false conjecture. Upon analysis of approximately 250 games where the deferred branching search strategy was used, three games resulted in more nodes being expanded than with the previous search strategies. The increase in the number of nodes was not significant, but it was interesting to determine the cause since we conjectured that this would not happen. A further analysis was done on

these three particular games to determine why the deferred branching strategy caused more nodes to be expanded. Since the deferred branching strategy is added on top of several other strategies, interactions of the various strategies unfortunately caused this result. In one of the three games, the "good" ordering of the nodes caused the increase in the number of nodes expanded. We expect that the results for the deferred branching technique could be better if the Good Ordering Technique were fine tuned. The performance of each of the techniques may be altered by interactions such as these.

Figure 4.34 is a game of four suits with twelve cards in which the deferred

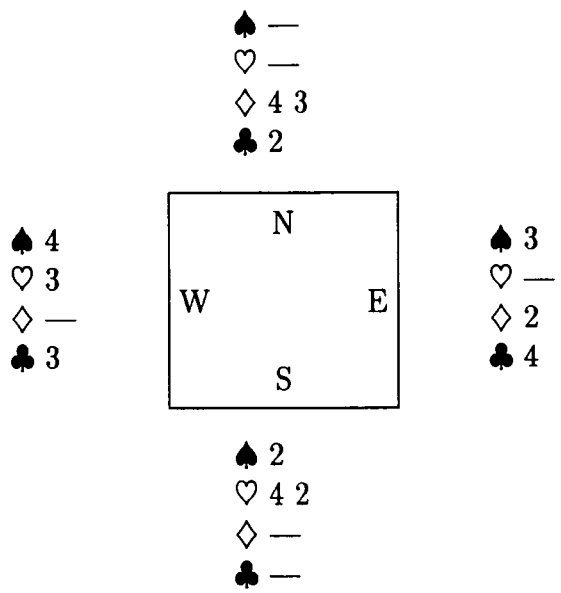


Figure 4.34: Game of Four Suits with Twelve Cards Corresponding to the Search Tree in Figure 4.35 and Figure 4.36

branching technique decreased the number of nodes expanded from 84 to 62. Figure 4.35 is the search tree corresponding to the game in Figure 4.34 created by the techniques incorporated before the deferred branching technique. Figure 4.36 is the search tree corresponding to the game in Figure 4.34 with the deferred branching strategy added. The tree in Figure 4.36 exhibits the buckets played by

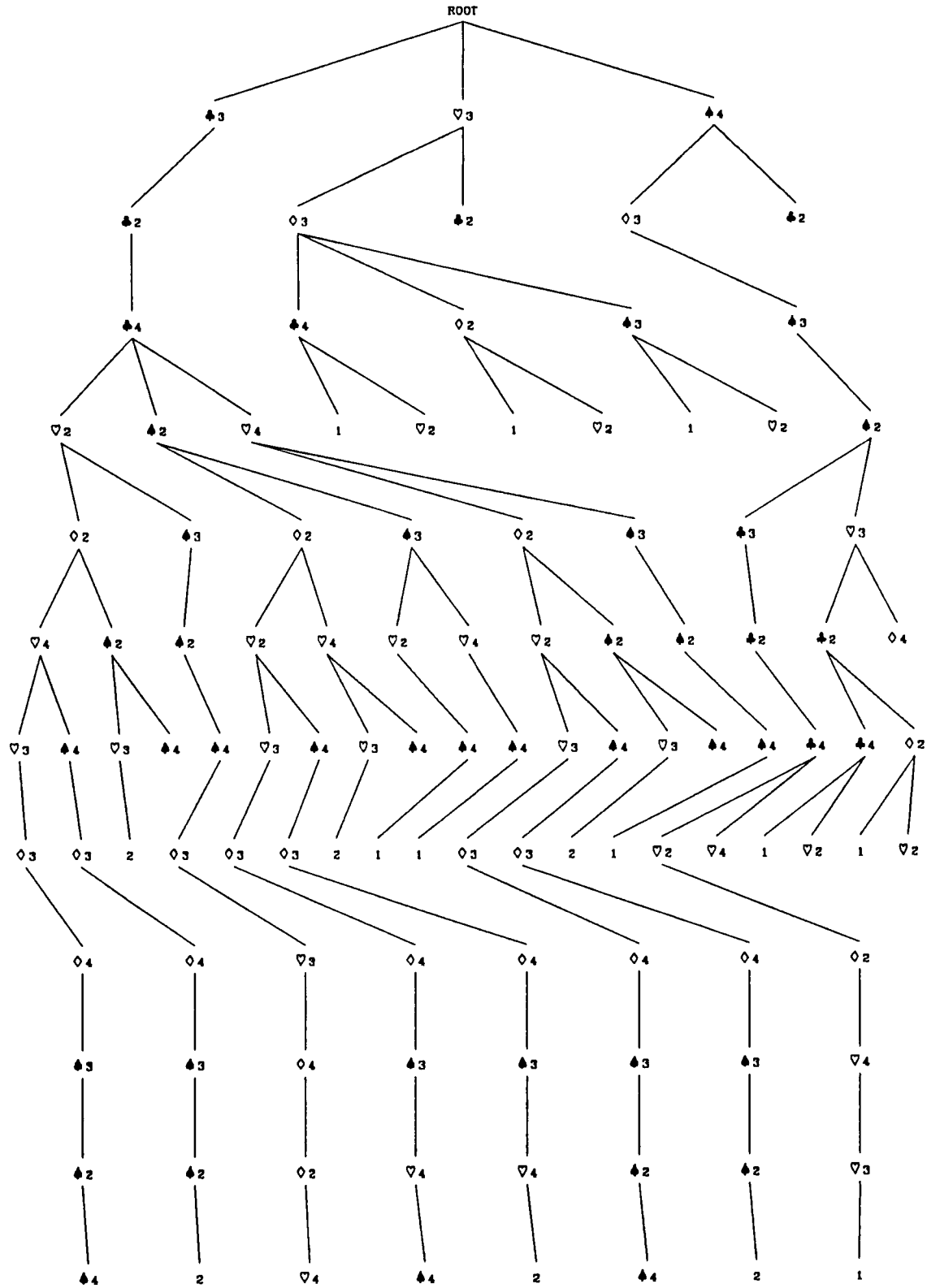


Figure 4.35: Search Tree Corresponding to Figure 4.34 before Addition of Deferred Branching Technique

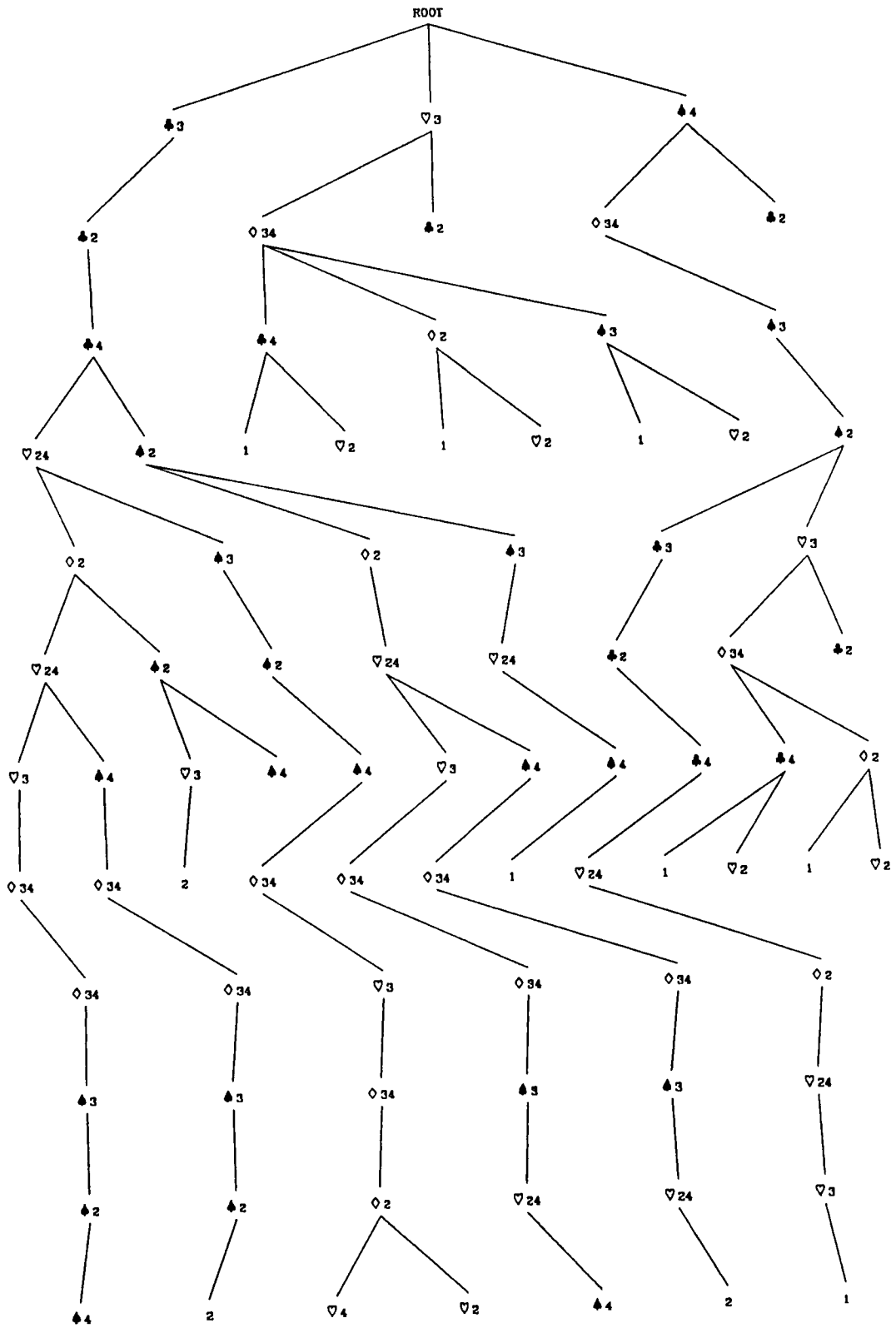


Figure 4.36: Search Tree Corresponding to Figure 4.34 after Addition of Deferred Branching Technique

showing all cards in the buckets. For example, if we look at the leftmost subtree in Figure 4.36 we see that the South hand played a bucket of ♡ cards, the two and the four, on the first trick. With the previous strategies as shown in Figure 4.35, these two cards resulted in two branches in the game tree. If we compare the subtrees of the two and four of ♡ in Figure 4.35, we see that the subtrees are similar. With the deferred branching technique, this similarity is realized and only one path is needed in the search tree instead of two.

Table 3.2 on page 19 shows the complexity of the implementation of this technique. The amount of CPU time needed per node in the search tree has increased by about a of factor of four. But, if we observe the frontier, Table 4.11, after the addition of the deferred branching technique we see that the frontier has

Table 4.11: Frontier Development Table for Deferred Branching Pruning Strategy

Number of Suits	Number of Cards								
	4	8	12	16	20	24	28	32	36
1	-	-	-	-	-				
2	-	-	-	-	-	-			
3			-			-			
4	-	-	-	-	-	-	-	*	

expanded in the case of games with four suits. We are now able to on average get results for games of four suits with 32 cards. The deferred branching technique successfully prunes many paths of the game tree which are not needed to determine the minimax result of the root node. Table 4.12 shows the average decrease in the number of nodes expanded as compared to the previous strategies. In particular, in games of four suits with 28 cards, a 79% decrease in the number of nodes expanded is realized.

Though the addition of the previous technique, internal cutoffs, did not

Table 4.12: Game Tree Complexity Comparison for Deferred Branching vs. Internal Cutoffs

Number of Suits	Number of Cards In The Game	Number of Nodes Expanded
1	4	0%
1	8	25%
1	12	56%
1	16	69%
1	20	83%
2	4	0%
2	8	27%
2	12	32%
2	16	62%
2	20	68%
2	24	74%
3	12	26%
3	24	69%
4	4	0%
4	8	0%
4	12	9%
4	16	34%
4	20	65%
4	24	78%
4	28	79%
4	32	——

allow the frontier to expand, addition of the deferred branching technique on top of the other strategies allows the frontier to expand in the case of games with four suits. Again, the interaction of the strategies is noticed as we may not have been able to expand the frontier at this point without the previous addition of the internal cutoffs. Figure 4.37, Figure 4.38, Figure 4.39, and Figure 4.40 graphically show the average number of nodes expanded as compared to the previous accumulation of techniques for games of one, two, three, and four suits, respectively. Deferred Branching has proven to be a successful step toward our goal of games of four suits with 52 cards. Though the implementation described here is specific to the game of double-dummy bridge, the idea of delaying decisions in the search tree could be applied to other games.

4.7 Hashing Technique

Many times in a game of bridge, the same point in the game can be reached by several different plays of the cards. For example, suppose two tricks of a game have been played with West leading both tricks. Imagine a sequence of the same cards being played where the cards in the first trick are played in the second trick and the cards of the second trick are played in the first trick. We reach a point in the imagined game equivalent to the original game. The same cards have been played and the same team has won both tricks thus far. Also, since West won both tricks, he is now the lead player of the third trick just as in the original game.

This phenomenon may occur many times as our search techniques exhaustively look at every possible play of the cards. If our search leads us to an equivalent point in the game which we have already reached through another path, we could reduce the search by looking up the minimax value that was obtained the first time we were at this point in the game. Thus we do not have to search

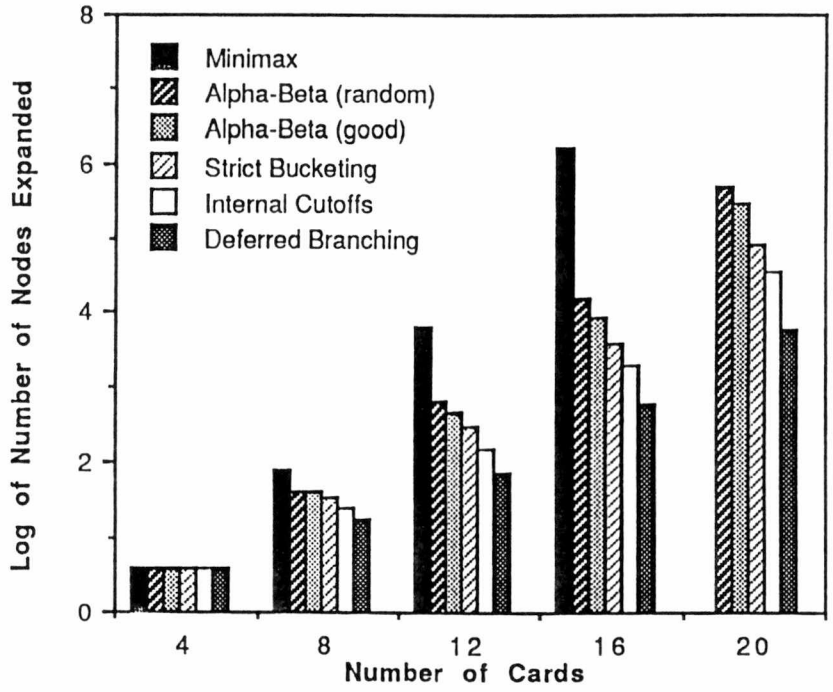


Figure 4.37: Deferred Branching vs. Previous Techniques, 1 suit games

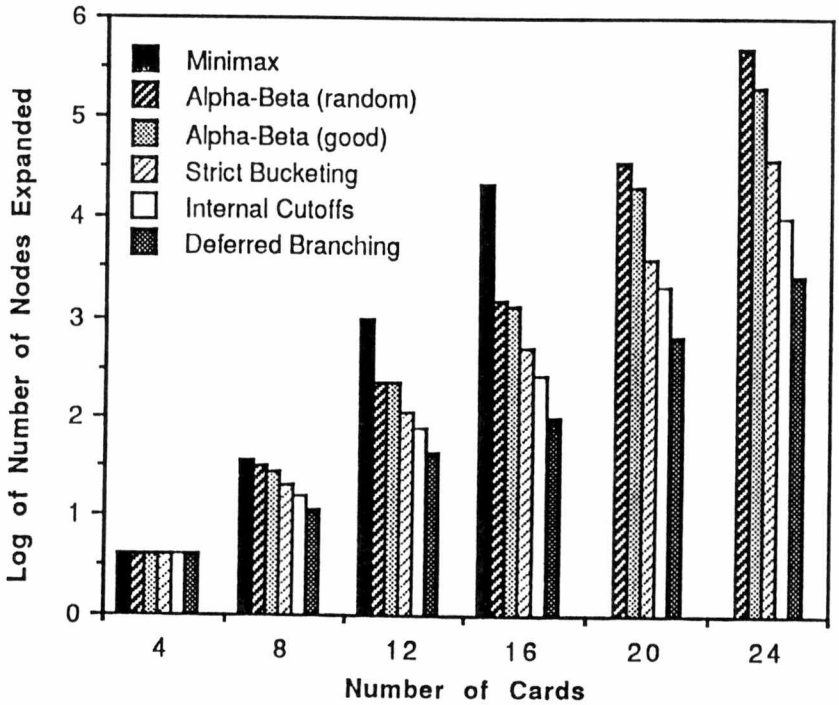


Figure 4.38: Deferred Branching vs. Previous Techniques, 2 suit games

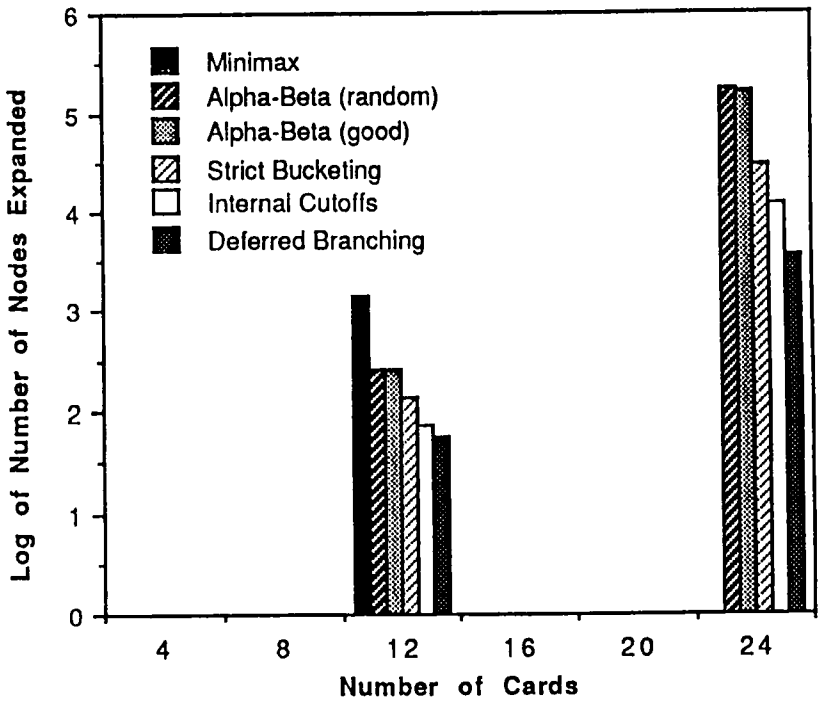


Figure 4.39: Deferred Branching vs. Previous Techniques, 3 suit games

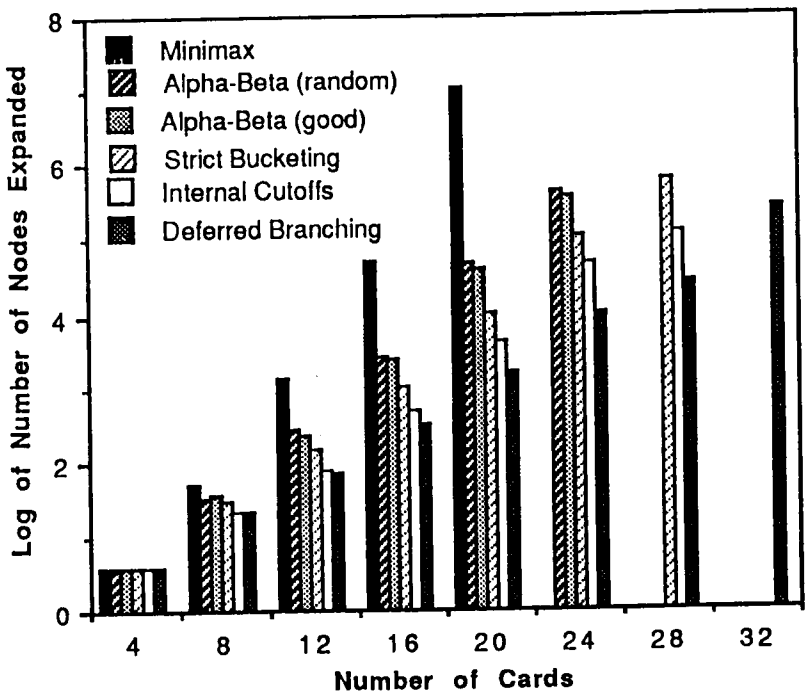


Figure 4.40: Deferred Branching vs. Previous Techniques, 4 suit games

the subtree from the current node. Use of the hashing techniques prevents the reevaluation of some subtrees.

A hashing technique was implemented to store minimax values of nodes at depths of the game tree corresponding to the end of tricks. Nodes at other levels were not stored in order to simplify the implementation of hashing and because limited storage prevents storage of all nodes. Since many types of pruning occur due to the various strategies, bound values are being passed up the search tree. Therefore we store bound information on the minimax value of the node in the hash table. The hash key includes information about the cards that have been played, the buckets that have been played, the number of tricks that have been won thus far, and the hand from which the next trick will be led.

The bound information that we store in the hash table about the node is determined by the value that is backed up from the search of the node's subtrees and the current value of ALPHA and BETA. Since prunes may have occurred in the subtrees, the value backed up is a bound on the true minimax value of the node. If the backed up value is precisely between the values of ALPHA and BETA, then the backed up value is the actual minimax value of the node. In this case, we store this value in the hash table as the actual minimax value of the node. If the backed up value is less than or equal to ALPHA, the true minimax value is less than or equal to this value. Thus, we store this value in the hash table as an upper bound on the actual minimax value of this node. If the backed up value is greater than or equal to BETA, the true minimax value is greater than or equal to this backed up value. Thus, we store this value in the hash table as a lower bound on the minimax value of this node. Assume that x is the backed-up value, then the information stored in the hash table about the minimax value of the current node can be summarized as follows:

```
if ALPHA < x < BETA
then store "eq x" in the hash table

if x <= ALPHA
then store "le x" in the hash table

if x >= BETA
then store "ge x" in the hash table
```

A paper by Campbell and Marsland entitled "A Comparison of Minimax Tree Search Algorithms" [9] was helpful in determining these rules.

We look up the nodes in the tree that are at depths corresponding to the end of tricks since these are the nodes for which we store information. If the key for the current node matches a key in the hash table, we obtain information about the minimax value of the current node in the form of either "eq x", "le x", or "ge x". Suppose the hash table lookup resulted in "eq x" information for the current node. Then the actual minimax value of the current node is x. Suppose the hash table lookup resulted in "le x" information for the current node. Then we must analyze this information further for the current node. If x is less than or equal to the ALPHA value at the current node, then x can be returned as the backed-up value for this node. If x is greater than ALPHA, then BETA must be set to x and the search continued from this node. Suppose the hash table lookup resulted in "ge x" information for the current node. Then we again must analyze this information further. If x is greater than or equal to BETA, then x can be returned as the backed-up value for this node. If x is less than BETA, then ALPHA must be set to x and the search continued from this node. Thus the bound information in the hash table may give us immediate information about the minimax value of the current node. Or, it may provide us with information

to tune the values of ALPHA and/or BETA before search of the subtree from the current node proceeds.

As we do not have infinite storage and time resources, it is not only interesting but necessary to determine the most efficient use of a hash table. Campbell and Marsland's paper, "Parallel Search of Strongly Ordered Game Trees" [8] indicates that hash techniques used in game trees in the past for games such as chess have been most successful in middle game and end game situations. With this in mind, we collected an additional set of data for the hashing technique in order to determine which depth, d, of the search trees for the game of double-dummy bridge the most successful lookups occurred. Figure 4.41 summarizes the information. In computing the savings for the graph of Figure 4.41, we consider

- $f(d)$: the number of nodes a successful lookup prevents us from traversing
- $c1$: the time to process one nodes
- $succ(d)$: the number of successful lookups at each depth of the search tree
- $c2$: the time to unsuccessfully look up then insert into the hash table
- $insert(d)$: the number of nodes inserted into the hash table at each depth
- $c3$: the time it takes to use the values returned from a successful lookup.

The values of $c1$, $c2$, and $c3$ were approximated. We set $c3$ to one. Then $c2$ must equal $c3$ + the time for an insertion into the table, which we approximate at being twice as time consuming as $c3$. Thus, let $c2 = c3 + 2$. The ratio $c1/avg(c2, c3)$ should be similar to the ratio of the *time per node before hashing/speedup per node due to hashing.* These values can be found from Table 3.2 on 19. Thus, $c1/2 \sim .127/.028$ implies $c1 \approx 9$. The (x, y) coordinates are calculated as follows:

$$x = d$$

$$y = c1 * succ(d) * f(d) - c2 * insert(d) - c3 * succ(d).$$

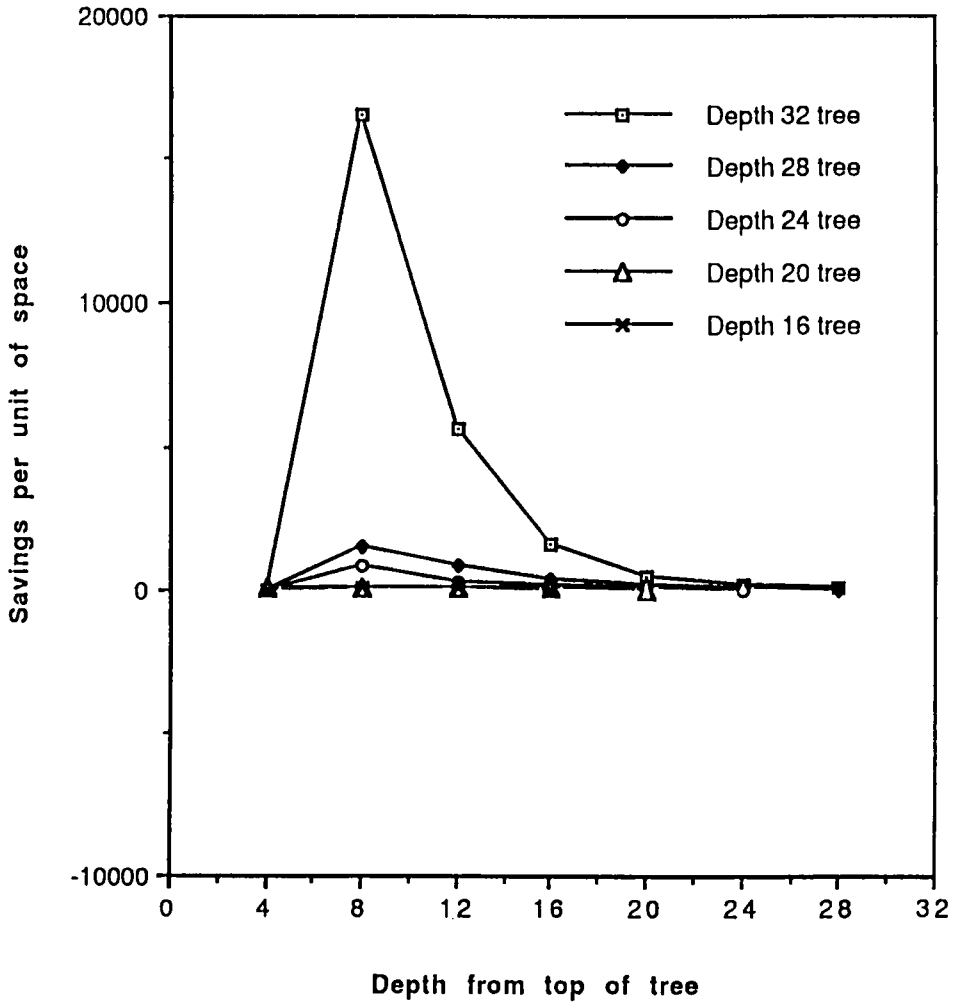


Figure 4.41: Hashing Payoffs

Figure 4.41 indicates that the best payoff comes at a depth of eight for the double-dummy game. Also an interesting observation is that the payoff increases as the size of the game increases. This is important as we pursue the goal of completing the full 52 card double-dummy game. This data shows that, unlike the game trees mentioned in [8], the hash table is most successful near the top of the tree.

Figure 4.42 is a game in which the addition of the hashing technique de-

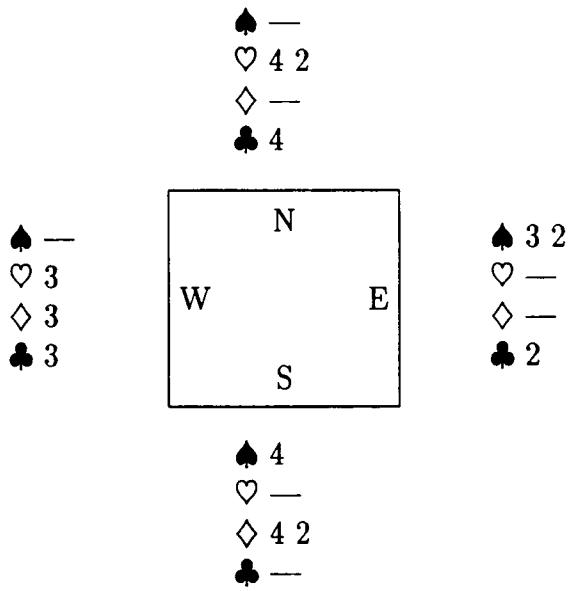


Figure 4.42: Game of Four Suits with Twelve Cards Corresponding to the Search Tree in Figure 4.43 and Figure 4.44

creased the number of nodes expanded from 91 to 75 nodes. Figure 4.43 is the search tree produced before the addition of the hash table. Figure 4.44 is the search tree produced after the addition of the hashing technique. The minimax value of the four right-most nodes at level eight in Figure 4.44 was retrieved from the hash table, preventing the search of their subtrees.

Table 4.13 is the frontier table after the addition of the hashing technique. The hashing technique does not provide enough of a decrease in the number of nodes expanded to expand the frontier. We must remember the exponential

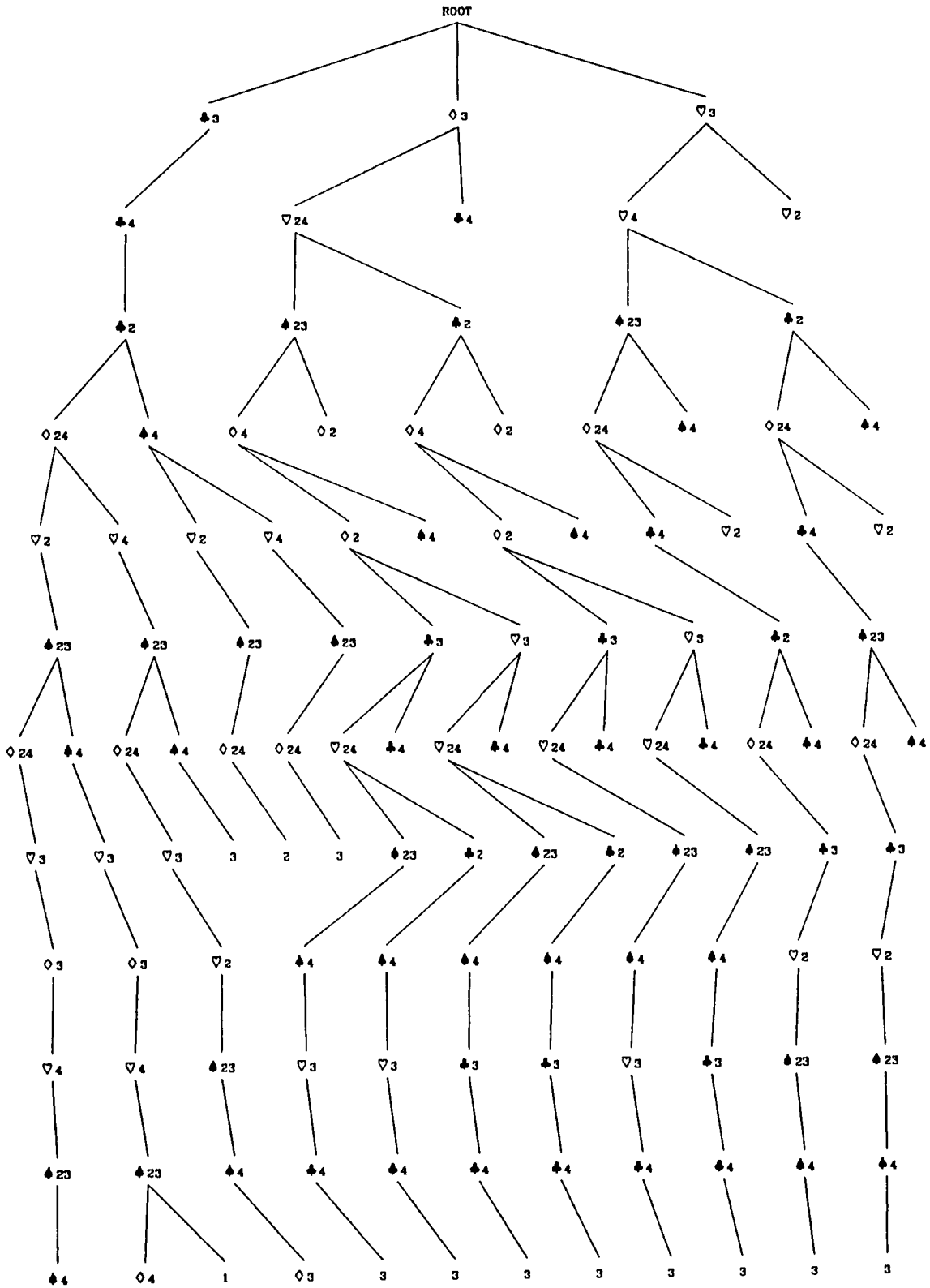


Figure 4.43: Search Tree for Figure 4.42 before the Addition of Hashing Technique

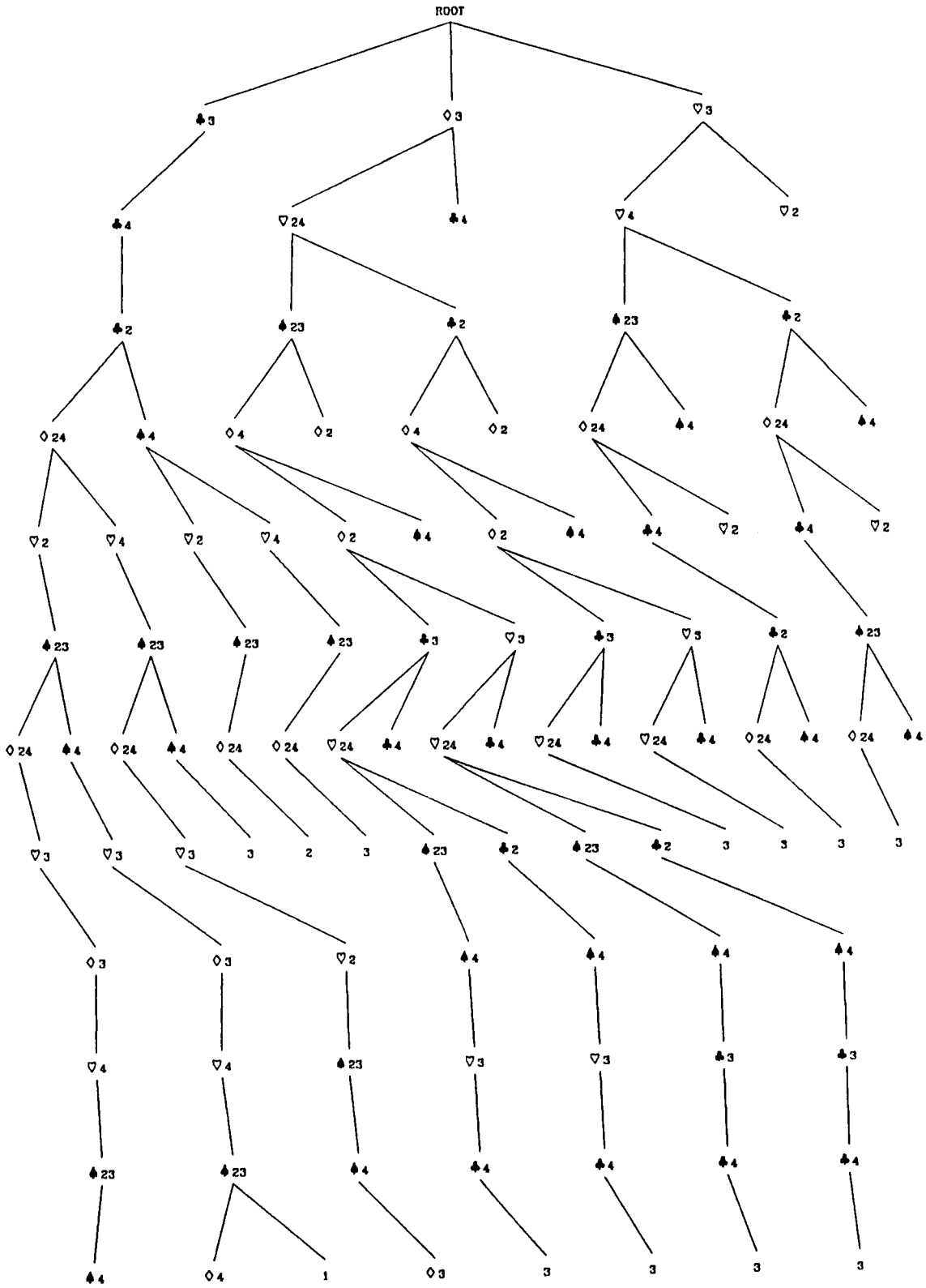


Figure 4.44: Search Tree for Figure 4.42 after the Addition of Hashing Technique

Table 4.13: Frontier Development Table for Hashing Technique

Number of Suits	Number of Cards								
	4	8	12	16	20	24	28	32	36
1	-	-	-	-	-				
2	-	-	-	-	-	-			
3			-			-			
4	-	-	-	-	-	-	-	-	

explosion that is occurring in complexity as we increase the size of our game trees. Figure 4.45, Figure 4.46, Figure 4.47, and Figure 4.48 show the decrease in the number of nodes expanded after the addition of the hashing technique. Table 4.14 shows the percentage decrease in the number of nodes expanded due to the addition of the hashing technique for games of all sizes. Though we are not able to expand our frontier, Table 4.14 shows the effectiveness of the hashing technique as a method for reducing the complexity of the game tree.

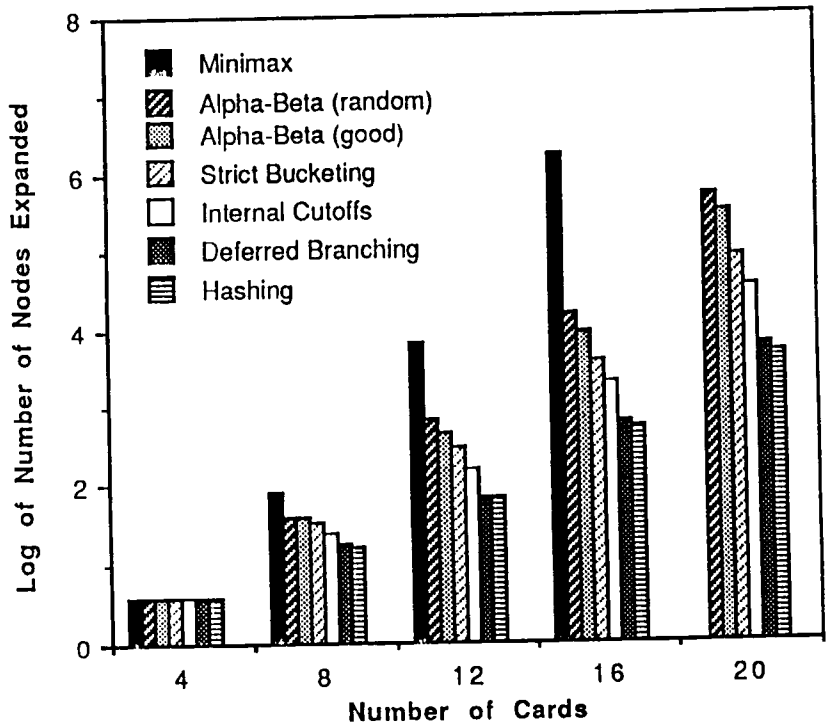


Figure 4.45: Hashing vs. Previous Techniques, 1 suit games

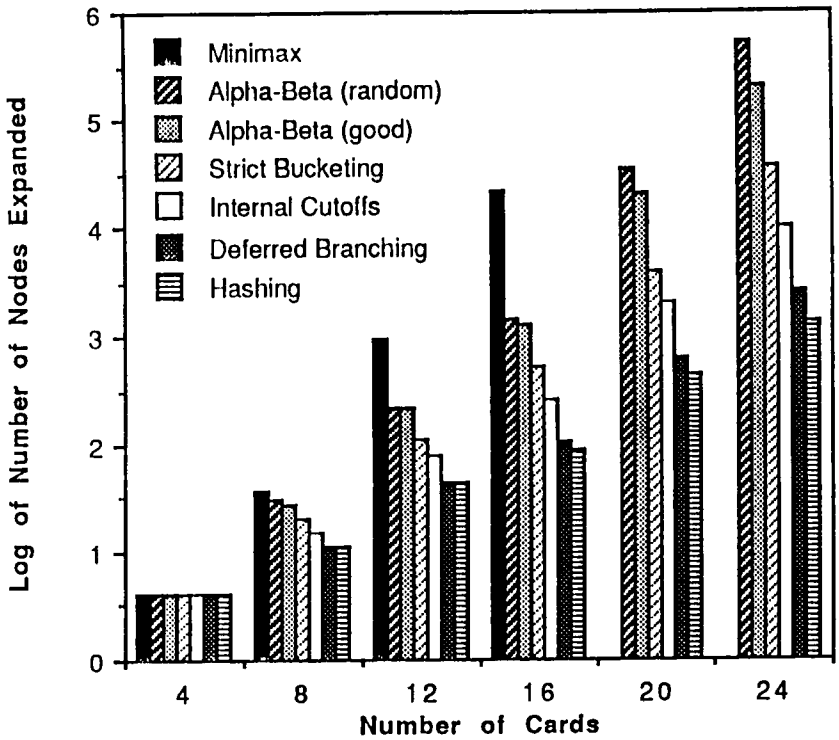


Figure 4.46: Hashing vs. Previous Techniques, 2 suit games

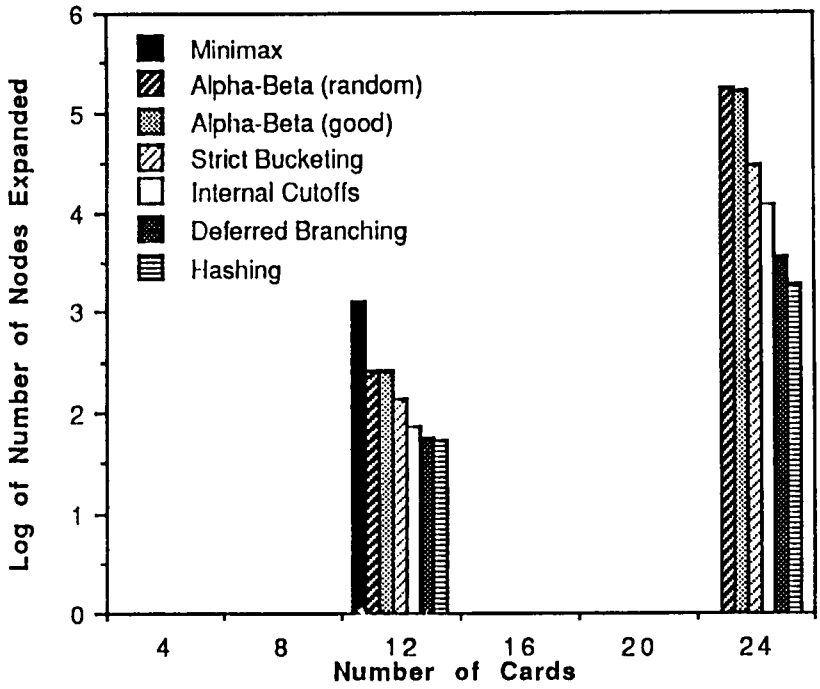


Figure 4.47: Hashing vs. Previous Techniques, 3 suit games

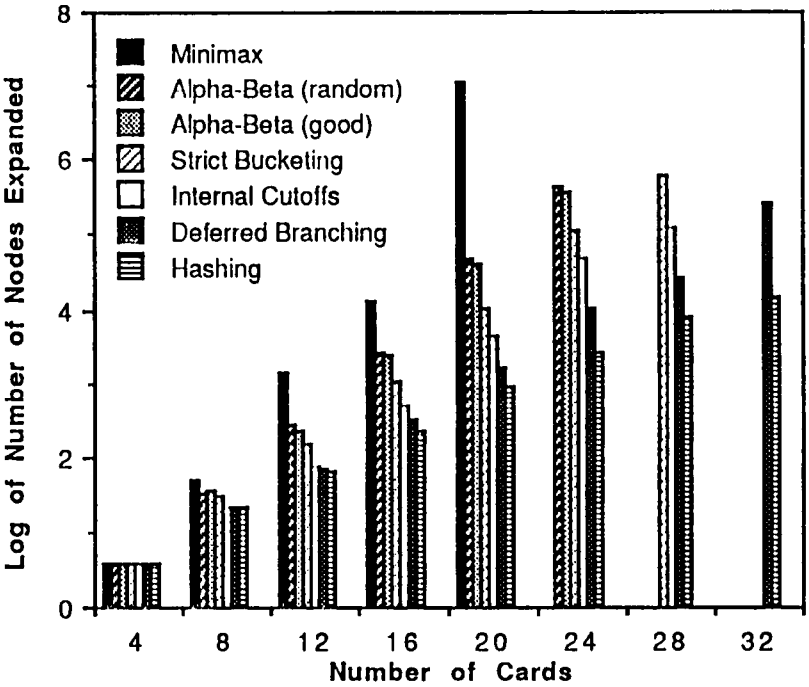


Figure 4.48: Hashing vs. Previous Techniques, 4 suit games

Table 4.14: Game Tree Complexity Comparison for Hashing Technique vs. Deferred Branching

Number of Suits	Number of Cards In The Game	Number of Nodes Expanded
1	4	0%
1	8	5%
1	12	0%
1	16	17%
1	20	19%
2	4	0%
2	8	0%
2	12	5%
2	16	12%
2	20	30%
2	24	47%
3	12	4%
3	24	46%
4	4	0%
4	8	0%
4	12	7%
4	16	30%
4	20	44%
4	24	74%
4	28	70%
4	32	95%

Chapter 5

SUMMARY

In order to produce an interesting comparison of the performances of the techniques, data was collected for 200 randomly dealt games of four suits with 16 cards after the addition of each of the techniques. Games of this size were chosen because games of four suits are particularly interesting to the double-dummy bridge problem and 16 cards is the largest number of cards all techniques are able to produce results for in less than 10 hours. After the addition of each of the techniques, the 99% confidence interval was calculated using the number of nodes expanded in the search trees as data. The 99% confidence interval indicates the interval within which the average of the number of nodes expanded will lie 99% of the time. Table 5.1 shows the 99% confidence interval information for the techniques.

We can see from the table that the intervals do not overlap except in the case of Alpha-Beta with random ordering and Alpha-Beta with good ordering. Note that with the addition of each technique, the confidence intervals lie below the previous intervals. We see from Table 5.1 that the addition of each technique proved to be a valuable tool in reducing the complexity of search.

Table 5.2 shows the average percentage decrease in the number of nodes expanded realized with the accumulation of the various pruning techniques. This

Table 5.1: Comparison of Confidence Intervals for the Number of Nodes Expanded for Each of the Techniques

Techniques (additive)	99% Confidence Interval
Minimax	[56157, 80920]
Alpha-Beta with random ordering	[2326, 2954]
Alpha-Beta with good ordering	[2195, 2825]
Strict Bucketing	[919, 1216]
Internal Cutoffs	[430, 581]
Deferred Branching	[290, 375]
Hashing	[203, 259]

Table 5.2: Game Tree Complexity Comparison for Accumulation of all Techniques vs. Basis Minimax Search Technique

Number of Suits	Number of Cards In The Game	Number of Nodes Expanded
1	4	0%
1	8	78%
1	12	99%
1	16	99.6%
1	20	—
2	4	0%
2	8	69%
2	12	96%
2	16	96%
2	20	—
2	24	—
3	12	96%
3	24	—
4	4	0%
4	8	56%
4	12	95%
4	16	99.7%
4	20	99.8%
4	24	—
4	28	—
4	32	—

decrease is based on the average number of nodes expanded with the basic minimax search technique. From Table 5.2 we see that the search reducing techniques incorporated into the minimax search strategy were successful at reducing the search space for the double-dummy bridge problem. Note that the amount of savings increases with the size of the game. Table 5.3 provides additional interesting information for games with 16 cards.

Table 5.3: Summary Information for the Number of Nodes Expanded in Double-Dummy Bridge Games of 16 Cards

Technique	MIN	MAX	AVG	STD. DEV.
Minimax	233	425,012	68,538	67,868
Alpha-Beta (random)	183	11,422	2,640	1,723
Alpha-Beta (good)	158	10,485	2,510	1,727
Strict Bucketing	136	5,265	1,067	812
Internal Cutoffs	46	2,488	505	413
Deferred Branching	34	1,356	332	233
Hashing	30	851	231	151

Though the various search reducing techniques were successful, we were not able to expand our frontier to enable a full double-dummy bridge game to be completed in less than 10 hours. We are however able to successfully produce results for hands of up to 32 cards within a reasonable amount of time. These results tell us the maximum number of tricks the N/S team can take against a perfect defense for randomly dealt hands. A large amount of data was compiled from which many interesting characterizations of the various techniques come. The techniques have proven to be valuable steps toward the realization of a perfect double-dummy bridge player.

Chapter 6

FUTURE IDEAS AND EXTENSIONS OF THIS WORK

With little alteration to the code implemented within this thesis, several questions relevant to a bridge game could be answered. In many phases of a Contract Bridge game various types of information are needed by the players in order to make decisions about the game. For example, answers to the following questions would be valuable information to a bridge player within the course of a game. Fortunately, the code implemented for this thesis would require few changes to provide this information. Some of the questions include:

- How many winners do I have in each suit?
- Can I fulfill my contract depending just upon high cards?
- Given a particular plan of action for the N/S team, what is the maximum number of tricks the N/S team can take with this plan?
- Given a particular plan of action for the E/W team, what is the maximum number of tricks the N/S team can take?

The result from the search of the game trees provided for this thesis is the maximum number of tricks the N/S team can take against a perfect defense. This could be extended to determine the particular play of the cards required to obtain this maximum number of tricks by the N/S team. This information would need to be represented as a tree in order to represent the plays of the N/S team for all possible plays by the E/W players. There was no need for this information within this thesis. However, this would be a valuable extension to provide information about the path that must be taken to realize the payoff.

Since the search of the full double-dummy game cannot be realized through the implemented techniques, a heuristic estimate could be developed after the search has proceeded to a particular depth. Thus, we would be able to provide an estimate on the maximum number of tricks the N/S team could take for the full double-dummy game. The advantage of this over purely heuristic methods for the estimation is that the underlying program is a perfect program. Thus, only after we have searched exhaustively as long as our time allows do we use heuristics to create an estimate. We would expect the use of the exhaustive search as an underlying program before heuristics are used would permit the estimate to be better than an estimate derived purely from heuristics. However, the reverse is possible [12].

A further idea, that would require more changes than the previous extensions, involves an alteration of the Deferred Branching Technique. When creating the buckets for each of the players, the deferred branching technique puts cards in the same bucket if they have the same effect on the current trick. An extension to this is to consider the team effect. For example, put all cards that will result in the N/S team winning the trick into a single bucket. Also, put all cards that will result in the E/W team into a single bucket. This extension would require much study and preparation before implementation as a careful process of keeping up

with the play of the game would be needed.

Several extensions could be made to the code in order to realize speedup. For example, exploration of parallel alpha-beta search algorithms could prove successful in the speedup effort. This extension is currently being investigated by Elizabeth Jones at the University of Tennessee as part of her research project for the Master of Science degree. Also, code optimization could be investigated to determine if speedup is possible and if the payoff is worth the effort.

BIBLIOGRAPHY

[1] Elwyn R. Berlekamp. Program for double-dummy bridge programs — a new strategy for mechanical game playing. *Journal of the ACM*, 10(4):357–364, 1963.

[2] G. L. Carley. A program to play contract bridge. Master's thesis, Massachusetts Institute of Technology, Cambridge, 1962.

[3] Shelly DeSatnick. *Bridge for Everyone.* First Avon Printing, New York, Nov 1982.

[4] Robert J. Fabrizio Jr. A two suit bridge player utilizing a combinatorial algorithm. Master's thesis, Duke University, Jun 1982.

[5] J. Kahn, J. Lagarias, and H. Witsenhausen. Single-suit two-person card play. *International Journal of Game Theory*, 16(4):291–320, 1987.

[6] Donald Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6:293–326, 1975.

[7] David N. L. Levy, editor. *Computer Games.* Springer-Verlag, New York, 1988.

[8] T. A. Marsland and Murray Campbell. Parallel search of strongly ordered game trees. *Computing Surveys*, 14(4):533–551, Dec 1982.

[9] T. A. Marsland and Murray Campbell. A comparison of minimax tree search algorithms. *Artificial Intelligence*, 20(4):347–367, Jul 1983.

[10] T. A. Marsland and Fred Popowich. Parallel game-tree search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 7(4):442–452, Jul 1985.

[11] Chris N. Napjus. Declarer: A learning bridge-playing program which generalizes and infers. *Computers in Behavioral Science*, 16(4):404–410, 1971.

[12] Dana S. Nau. Decision quality as a function of search depth on game trees. *Journal of the ACM*, 30(4):687–708, Oct 1983.

[13] Judea Pearl. The solution for the branching factor of the alpha-beta pruning algorithm and its optimality. *Communications of the ACM*, 25(8):559–564, Aug 1982.

[14] J. R. Quinlan. A knowledge-based system for locating missing high cards in bridge. In *Proceedings of the 4th IJCAI, Tokyo*, 1979.

[15] Elaine Rich. *Artificial Intelligence.* McGraw-Hill Book Company, New York, 1983.

[16] J. S. Riley and T. Throop. An interactive bridge playing program. In *Proceedings of the National Gaming Council, 8th Symposium*, Kansas City, 1969.

[17] J. R. Slagle and J. K. Dixon. Experiments with some programs that search game trees. *Journal of the ACM*, 16:189–207, 1969.

[18] A. Stanier. *BRIPP—A bridge playing program.* PhD thesis, Essex University, Colchester, 1974.

[19] A. Stanier. A bridge bidding program. In *Proceedings of the 4th IJCAI, Tbilisi*, 1975.

[20] A. Stanier. Planning to make tricks at bridge. In *Proceedings of the AISB Summer Conference*, 1976.

[21] Leon Sterling and Yossi Nygate. Python : An expert squeezer. In *Proceedings of the 4th International Conference on Logic Programming*, 1987.

[22] T. A. Throop. The univac plays bridge. *Computers and Automation*, 11:3B–5B, Mar 1962.

[23] A. I. Wasserman. *Achievement of skill and generality in an artificial intelligence program.* PhD thesis, University of Wisconsin, Madison, 1970.

APPENDIXES

Appendix A

Calculation of Lower Bound on Time It Takes to Solve a Full 52 Card Double-Dummy Bridge Game

In order to calculate a lower bound, consider a deck of 48 cards such that each of the four hands has three cards from each of the four suits. A deck of 48 cards is chosen because a full 52 card game will take at least as long as a 48 card game, and since discarding could only cause more branches we choose this evenly dispersed hand. The root of the tree will have 12 children corresponding to the lead player's choices of lead plays. At depth four, which corresponds to the lead play of the second trick, there are 11 children. At depth eight, there are 10 children, etc. The first time the three non-leading hands must follow suit, they each have three choices of cards. The second time they play from this same suit, they have two choices. The third time, they have only one choice of which card to play. But since there are four suits, we must consider $(3!^4)^3$ branches for non-lead plays. Therefore, the lower bound on the number of paths in a game of double-dummy bridge is $12! * 3!^{12}$ which is approximately equal to 1.04×10^{18}.

Assuming the search can process 1000 paths of the search tree per second, the exhaustive search would take over three million years.

Appendix B

Rules for Intelligently Ordering the Nodes used by the Alpha-Beta with Good Ordering of Nodes Search Technique

This ordering was developed by an expert bridge player, Dr. David Mutchler of the University of Tennessee Computer Science Department, in order to realize the effects of a good ordering of nodes in the game tree for double-dummy bridge. A different set of rules is developed for each of the hands of the trick. For example, the rules for the lead hand of the trick are different than the rules for the other hands of the trick. An honor card is defined as any of the top five cards of a suit that have not been played. When a hand does not contain any cards of the lead suit, a discard order must be followed.

The rules for the order of children when a discard situation occurs are as follows:

1. Lowest card from each suit that satisfies the following rule: the number of cards in the hand is more than the maximum of the number of cards any of

the other hands have of this suit,

2. Then lowest from suits that did not satisfy the above condition,

3. Then merged highest to lowest cards of each suit (i.e. highest from suit 1, highest from suit 2, highest from suit 3... , lowest from suit 1, lowest from suit 2, lowest from suit 3).

The rules for the order of children for the first hand in the trick:

1. Lowest card from each suit,

2. Then highest card from any suit down to lowest card from any suit.

The rules for the order of children for the second hand in the trick:

1. Cover an honor with cheapest highest honor,

2. Then lowest,

3. Then lowest that will definitely win the trick,

4. Then highest down to lowest.

The rules for the order of children for the third hand in the trick:

1. Lowest that will definitely win,

2. Then lowest,

3. Then highest down to lowest.

The rules for the order of children for the fourth hand in the trick:

1. Lowest that will definitely win,

2. Then lowest,

3. Then highest down to lowest.

Appendix C

Pseudo Code for Marking Buckets in the Deferred Branching Search Technique

```
For each BUCKET of the four buckets played in the last trick:

    set CURRENT to the element of the buckets array corresponding
        to the largest card in BUCKET
    set LEN = the number of cards in BUCKET
    set CNT = 1  (one of LEN cards played in the bucket)

Step 1:
   For each element of the buckets array greater than
       CURRENT that belongs to the same player as CURRENT

       Increment COUNT of any element whose LENGTH indicates that
           it encompasses BUCKET
       If the COUNT for this element equals the LENGTH for this element
```

```
        Then
            set save-CURRENT to this element
            set save-BUCKET to correspond to the bucket whose
                largest card is CURRENT
        set CURRENT to save-CURRENT if save-CURRENT was set
        set BUCKET to save-BUCKET if save-BUCKET was set
        If CURRENT's COUNT = CURRENT's LENGTH
            Then skip STEP 2 and STEP 3

Step 2:
    Inc CNT for each existing bucket that BUCKET encompasses

Step 3:
    If there is no existing mark on CURRENT in the buckets array
        Then
            mark CURRENT with LEN and CNT
            set COND1-FLAG to TRUE
        Else
            If existing LENGTH on CURRENT >= LEN
              Then increment the existing COUNT on CURRENT
            If LEN >= existing LENGTH on CURRENT
              Then increment CNT

Step 4:
    DO until either CONDITION 1 or CONDITION 2 is met

        COND 1: COND1-FLAG is TRUE thus no more marking is necessary
```

```
        EXIT DO LOOP

COND 2: CURRENT has no existing marks so mark CURRENT
            with LEN and CNT
        EXIT DO LOOP

COND 3:
  If LEN = CNT (cards must be marked as played)
     Step 5:
     Foreach element of the buckets array greater than
        CURRENT that belongs to the same player as CURRENT

       Alter LENGTH and COUNT of the element if the cards
        being marked as played effect the bucket whose
        largest card is this element

        If COUNT = LENGTH after this alteration
           Then
              set CURRENT to this element
              set BUCKET to correspond to the entire group of
                  cards to be marked as played

     Mark the elements of BUCKET as played
     Mark corresponding elements of the HANDS array as played
     If the smallest card marked as played has an existing
        LENGTH and COUNT mark
       Then
```

```
        set LEN to LENGTH-1 where LENGTH corresponds to the
            smallest card played
        set CNT to COUNT-1 where COUNT corresponds to the
            smallest card played
        set CURRENT to the largest card smaller than the
            smallest card played
        set BUCKET to correspond to the bucket whose largest
            card is CURRENT

Else (cards do not need to be marked as played but
     CURRENT has existing LENGTH and COUNT marks)
  set x to max (LEN, existing LENGTH mark of CURRENT)
  set y to max (CNT, existing COUNT mark of CURRENT)
  If LEN < existing LENGTH mark of CURRENT
     Then mark CURRENT with LEN and CNT
  set LEN to x-1
  set CNT to y-1
  If   1) card just smaller than CURRENT has existing
          LENGTH and COUNT marks and
       2) LEN <= the LENGTH mark of the card just smaller
          than CURRENT
     Then increment the COUNT of the card just
          smaller than CURRENT
  set CURRENT to the card just smaller than CURRENT
  set BUCKET to correspond to the bucket whose largest
      card is CURRENT
```

VITA

P. Susan McFaddin Williams graduated from the University of Tennessee, Knoxville in June of 1987 with a Bachelors of Science Degree in Computer Science and Mathematics. She was then employed by Honeywell Space and Strategic Avionics Division in Clearwater, Florida until July of 1988. She reentered the University of Tennessee, Knoxville in August of 1988 and graduated with a Master's of Science in Computer Science in December of 1989. She now works for Tennessee Eastman Company, Kingsport, Tennessee and resides in Blountville, Tennessee.