To the Graduate Council:

I am submitting herewith a thesis written by Elizabeth McFarlin Jones entitled "A Graphics Interface and an Evaluation of Parallel Alpha-Beta Search Algorithms for a Mechanical Bridge Player." 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.

David Mutchler

Dr. David Mutchler, Major Professor

We have read this thesis
and recommend its acceptance:

D J Sorin

Joan R. S. Blair

Accepted for the Council:

Lewinkel

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 the 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 Elizabeth McFarlin Jones

Date April 6, 1990

A Graphics Interface and an Evaluation of Parallel Alpha-Beta Search Algorithms for a Mechanical Bridge Player

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Elizabeth McFarlin Jones

May 1990

To my grandfather, Park G. Vestal (1906-1990).

ACKNOWLEDGEMENTS

I want to thank my major professor, Dr. David Mutchler, for all the guidance that I have received throughout this project. His positive attitude and criticism were essential to the completion of this work. I would also like to thank the other members of the committee, Dr. Jean Blair and Dr. David Straight, for their suggestions and participation.

Finally, I would like to thank my husband, Fami, for his unwavering support and understanding. As my colleague, he offered me valuable insights and criticism. As my husband, he encouraged me and frequently played housekeeper, cook, secretary and therapist to assist me.

ABSTRACT

As one of a series of Master's theses that will hopefully lead to the development of a world class contract bridge playing program, this work focuses on two areas: the design and implementation of the graphics interface CBRIDGE and an introductory exploration of the potential benefits of three parallel alpha-beta search algorithms. First, CBRIDGE was designed and implemented in order to provide an environment in which human and machine players could interact. This interface will facilitate the testing of machine players and eventually enable machine players to be entered into tournaments. Second, we conducted experiments with three parallel alpha-beta search algorithms -- aspiration, principal variation alpha-beta and principal variation splitting -- that were developed with chess game trees in mind to see whether the reported benefits in speedup applied to the search of bridge game trees. We anticipated these results would indicate what magnitude of improvements would be possible by parallelizing the alpha-beta search currently used in our double-dummy machine player.

TABLE OF CONTENTS

	Page
1 INTRODUCTION	1
2 CBRIDGE: A GRAPHICS INTERFACE	5
2.1 Introduction	5
2.2 Game Options and Parameters	8
2.2.1 Player Options	10
2.2.2 Display Options	12
2.2.3 Bidding Options	12
2.2.4 Hand Options	17
2.2.5 Deck Options	23
2.2.6 End Game Options	26
2.3 Machine Players and Communication Issues	27
2.4 Error Handling and Backing Up	32
2.4.1 Error Trapping	32
2.4.2 Backing Up	33
2.5 Defaults and Automatic Play	37
2.5.1 Default Bids and Card Selections	39
2.5.2 An Automatic Bidding and Playing Mode	41
2.6 Summary	43
3 PARALLEL ALPHA-BETA ALGORITHM STUDY	44
3.1 Introduction	44
3.2 Aspiration Algorithm	49
3.2.1 Experimental Design	51
3.2.2 Performance Expectations	52
3.2.3 Experimental Results	53
3.3 Principal Variation Alpha-Beta Algorithm	62
3.3.1 Experimental Design	62
3.3.2 Performance Expectations	66
3.3.3 Experimental Results	68
3.4 Principal Variation Splitting Algorithm	73
3.4.1 Experimental Design	74
3.4.2 Performance Expectations	74
3.4.3 Experimental Results	78
3.5 Summary	82
4 CONCLUSIONS	87
5 FUTURE WORK	90

Page

BIBLIOGRAPHY 93

APPENDIX: An Introduction to the Game of Bridge 96

VITA 98

LIST OF FIGURES

	Page
Figure 2.1 Option Setting Menu	9
Figure 2.2 Player Options	11
Figure 2.3 Display Options	13
Figure 2.4 "Hide Hand" and "Show Hand" Samples	14
Figure 2.5 Interactive Bidding Option	15
Figure 2.6 Fixed Auction Option	16
Figure 2.7 Random Hand Option	18
Figure 2.8 Fixed Hand Option	19
Figure 2.9 Sample Legal Input File	20
Figure 2.10 Sample Template File	21
Figure 2.11 Interesting Hand Option	22
Figure 2.12 Deck Options	25
Figure 2.13 Machine Explanation Option	29
Figure 2.14 Machine Explanation Displayed	30
Figure 2.15 Sample Error Detection and Diagnostic	34
Figure 2.16 Back Up Auction Option	35
Figure 2.17 Proposed Back Up Trick Option	38
Figure 3.1 Sample Alpha-Beta and Aspiration Searches	50
Figure 3.2 Aspiration Search Run Times, 1 Suited Hands	54
Figure 3.3 Aspiration Search Run Times, 2 Suited Hands	54
Figure 3.4 Aspiration Search Run Times, 3 Suited Hands	55
Figure 3.5 Aspiration Search Run Times, 4 Suited Hands	55
Figure 3.6 Comparison of Alpha-Beta and Serial Aspiration Search	57

Page

Figure 3.7 Comparison of Serial and Parallel Aspiration Search 59

Figure 3.8 1-Suit 4-Card Minimax Tree 61

Figure 3.9 1-Suit 8-Card Minimax Tree 61

Figure 3.10 Sample Processor Tree 63

Figure 3.11 Sample Lookahead Tree 63

Figure 3.12 Sample Application of PValphabeta Algorithm 64

Figure 3.13 PValphabeta Version 2 Processor Tree 65

Figure 3.14 PValphabeta Version 3 Processor Tree 67

Figure 3.15 PValphabeta Search Run Times, 1 Suited Hands 69

Figure 3.16 PValphabeta Search Run Times, 2 Suited Hands 69

Figure 3.17 PValphabeta Search Run Times, 3 Suited Hands 70

Figure 3.18 PValphabeta Search Run Times, 4 Suited Hands 70

Figure 3.19 Comparison of Alpha-Beta and PValphabeta Search 71

Figure 3.20 PVSalphabeta Algorithm 75

Figure 3.21 Sample Processor Tree 76

Figure 3.22 Sample Lookahead Tree 76

Figure 3.23 Sample Application of PVSalphabeta Algorithm 77

Figure 3.24 Percentage of Best Children by Position 79

Figure 3.25 PVSalphabeta Search Run Times, 1 Suited Hands 80

Figure 3.26 PVSalphabeta Search Run Times, 2 Suited Hands 80

Figure 3.27 PVSalphabeta Search Run Times, 3 Suited Hands 81

Figure 3.28 PVSalphabeta Search Run Times, 4 Suited Hands 81

Page

Figure 3.29 Comparison of Alpha-Beta and PVSalphabeta Search 83

Figure 3.30 Comparison of Aspiration, PValphabeta and PVSalphabeta Searches 85

CHAPTER 1

INTRODUCTION

Computer game playing is an area of interest to artificial intelligence researchers because games of skill provide a limited but well-defined environment within which computers can be programmed to "reason" and "imitate" the thought processes. Much work has been done studying chess, a perfect information game, and several programs have been developed to play at the master and expert level [Newborn 1989]. Contract bridge (see Appendix for a description of the game), however, is a game of imperfect information and as such it offers a suitable environment to broaden the understanding of computer game playing and computer "reasoning." The level of hidden information (i.e. the information one player does not have about other players' hands) in contract bridge is somewhat unique for a card game. Unlike poker, for example, the combination of all the players' hands gives total information. It is this property that allows humans (and therefore computers) to make deductions about what other players do as opposed to taking a probabalistic approach as in a game like poker [Stanier 1975]. Expert human players fill in some of the hidden information in contract bridge based on an evaluation of such things as what a player did or did not bid and what card was or was not lead [Quinlan 1979]. The problem addressed here is to enable machine players to reason in a similar manner.

Although contract bridge is a very popular card game, relatively little work [Berlekamp 1963, Stanier 1975 and 1976, Quinlan 1979] has been published on programming it. The commercial bridge programs available typically do not play well. As the editor of *The Bridge World* magazine says, "Efforts to get a computer to play bridge have been abysmal failures....Writing a program to play well, even to play beyond a novice's level, seems to be a fiendishly difficult problem." [Kaplan 1989, p. 24]

Berlekamp [1963] published an article about a heuristic, non-bidding program capable of solving no-trump double-dummy bridge problems, a variation of contract bridge in which there is no hidden information (see Appendix for a more complete description of double-dummy bridge), and providing proofs about the solution. This machine player succeeded in solving some problems with a 32-card deck in minutes. Unfortunately, it required hints to solve the harder hands (assuming a full deck) within a reasonable amount of time.

Stanier developed two programs, one bidder [1979] and one strategic planner to make tricks [1976], for the game of contract bridge. The bidder program makes bids on par with an average club player and derives inferences from the bidding sequence. It does not, however, make indirect inferences based on bids not made. The planner program develops strategies for making a given contract that includes the concepts of finessing and ruffing as well as other bridge "schema." Unfortunately, there remains no secondary program to implement this strategy in terms of a sequence of cards.

Quinlan [1979] developed an expert system to locate missing bridge honor cards in closed hands by analyzing the bidding sequence and the opening lead. Although it is successful approximately 80 percent of the time, it can not resolve conflicting derived inferences.

The majority of these machine players, developed during the 1960s and 1970s, used heuristics to deal with the hidden information, and to our knowledge none of them played at a consistently expert level. Mutchler believes the development of a hybrid technique involving both heuristics and search of the game tree to be important in the creation of an expert bridge player, and he is working along with others to demonstrate this by developing a world class bridge player. In pursuit of this goal, machine players for the bidding and play phase of the game are in the process of being implemented. The bidding program, PRECISION, is an expert system consisting of rules primarily taken from the book *Precision and SuperPrecision Bidding* by Belladonna and Gorozzo and from Mutchler, our resident bridge expert. The basic form of this bidder was developed by Cirelli, Kolwyck and Spencer in conjunction with Mutchler [Cirelli 1989]. Currently more rules are being added to PRECISION, and the ability to infer the strength of a partner's hand is being added by Wilson and Mutchler.

As an interim goal in the development of a program for the play phase of bridge, a machine player for the game of double-dummy contract bridge has been developed by Williams and Mutchler [Williams 1989]. This program plays "perfect" (in the minimax sense) double-dummy bridge. The search techniques implemented to date are primarily additional pruning rules to the standard alpha-beta algorithm [Slagle 1969]. These techniques allow the machine player to complete on average 32-card games in under 10 hours. This work reflects Mutchler's belief that it is more feasible to speed up a slow but perfect bridge player than to make a fast, weak player play better bridge. Although the current time costs of the double-dummy player may appear discouraging, we believe that this approach will prove advantageous in the future. In the overall scheme, speeding up the double dummy player so that it can play a full 52-card game in a reasonable amount

of time and then incorporating a method of dealing with hidden information will perhaps be the topic of future Master's theses.

Now that we have looked at the current state of the larger project whose goal is to engineer the world's best contract bridge player, let us consider the two contributions of this thesis. First, a graphics interface was designed and implemented in order to provide an environment in which human and machine players could interact. This interface will facilitate the testing of machine players and eventually enable machine players to be entered into tournaments. Second, we conducted experiments with three parallel alpha-beta search algorithms that were developed with chess game trees in mind to see whether the reported benefits in speedup applied to the search of bridge game trees. We anticipated these results would indicate what magnitude of improvements would be possible by parallelizing the alpha-beta search in the double-dummy player. These two contributions are discussed in more detail in the next chapters.

CHAPTER 2

CBRIDGE: A GRAPHICS INTERFACE

2.1 Introduction

The first of the two goals for this research project was to design and implement a graphics interface for the game of contract bridge. Our original design called for the production of a user friendly and entertaining contract bridge game capable of challenging human players at a variety of skill levels, as well as providing an environment in which machine players could be tested and eventually entered into tournaments. As you can probably imagine, this is a very tall order, one which we have not fully met given our time constraints. What we have implemented, however, is a solid first version of the contract bridge graphics interface named CBRIDGE. CBRIDGE runs on Sun 3/60 workstations and is approximately 7500 lines of C and XWindows graphics code; it supports both human and machine players and provides a variety of instructive, useful and fun features.

Although CBRIDGE was developed on a Sun 3/60, it should be capable of running on other compatible machines (e.g. Sun 4s). The basic software and hardware capabilities required for CBRIDGE to function properly are (1) access to XWindows Xlib graphics functions, (2) access to a standard Unix C compiler, (3) a monitor with a resolution of at least 920 x 720 pixels, and (4) a three button mouse. While the two software requirements

are seemingly straight forward, unexpected problems can arise. For example, when we tried to test CBRIDGE on a Sun 4 Sparc station that runs XWindows, portions of CBRIDGE failed to compile because the XWindows include files could not be found; they were not located in the "standard" subdirectory. The reasons for the monitor and mouse hardware requirements are as follows. The minimum resolution is the fixed size of the initial CBRIDGE window; any smaller sized screen would cause the graphics to be distorted. A mouse with more than three buttons would be acceptable to CBRIDGE, but the help messages that refer to the "right" button, for example, intending to reference button three, would be inaccurate. Similar inconsistencies with the help messages would arise when a two button mouse was in use; on these devices a button three event is generated by simultaneously pressing both buttons one and two. A single button mouse, however, would be unacceptable and cause CBRIDGE to "hang" interminably in the play phase of the game. In addition to these requirements, the user must be using the default display device of a machine. This typically means that a user must be sitting at the machine on which he/she wishes to run CBRIDGE. Otherwise, the DISPLAY environment variable must be set to the appropriate local machine's monitor, and the machine on which CBRIDGE is running must be added to the local machine's access control list via the *xhost* command.

Before examining some of CBRIDGE's specific features, let us consider some general issues, artistic and technical, that will affect the quality and user-friendliness of a graphics interface. Granted that a beautiful presentation can not make up for a lack of technical merit, a neat and attractive design can greatly enhance a user's overall impression of the interface. The more technical issues are consistency, menu format, feedback and help messages, error handling and response time [Hearn 1986].

Consistency is very important and is expressed in many ways. For example, status messages appear at fixed places on the display surface, and menu items appearing in several different menus appear in the same relative position within a menu. Another example of consistency is the function of the mouse: the left mouse button always selects an option while the right button always executes or "okays" an option.

Menu formats are simple, uncluttered and easy to understand. A multi-level menu structure organizes the initial game options and parameters into logical submenus; this structure supports menus which have only a few options apiece, so that the user is not overwhelmed with all the choices at once. "In general, menus with few options...reduce the amount of searching time needed to find a particular option, and they take up less screen space." [Hearn 1986, p. 335] Menus are displayed only when needed and when choices are valid.

Feedback messages are always displayed in the same area so the user knows where to look for errors, and HELP is always available for options and functions that are currently accessible. To help the user learn the system, there are facilities for both computer initiated prompting as well as user initiated requests for help messages and information regarding the use of the program. Aside from the HELP function, there is always a display indicating whose turn it is, as well as an added prompt if a human attempts to play out of turn.

Errors resulting from illegal human "moves" are handled by displaying the cause of the error with a suggested remedy. These diagnostics can help the user understand what went wrong and how to correct the situation. Backing up features also support confident user exploration by allowing users to correct mistakes and cancel selected commands before

they begin executing. Not only are there features to back up the entire game, but a user's selection usually does not execute until he or she gives the "okay" or "execute" command.

All user input is via menu selections (except where the user is prompted and must type in a file name), and the system makes some response to each input instantaneously. When processing time for a request or input is lengthy, as it is when waiting for a machine player's bid or play, the arrow cursor is converted into a clock to let the user know that his/her input has been received and is executing. In this way, the system maintains a continual interactive dialogue with the user.

Having seen examples of how some of the more general principles of interface design have been incorporated into CBRIDGE, we will next discuss some of the most interesting specific features implemented as well as some ideas for feature enhancement. Ideally, some of these feature enhancements would be implemented as the overall project of developing a world class contract bridge player continues.

2.2 Game Options and Parameters

Game flexibility is the key principal behind the game options and parameters. CBRIDGE allows the user to tailor the way in which the game will be played to his/her own specifications or to play a default game type. Most options and parameters may be adjusted prior to the start of the game, some may be modified during the game, and still others after the completion of a game. Initially, if the user elects to modify the game option settings, he/she must do so interactively via a hierarchical menu structure (see Figure 2.1).

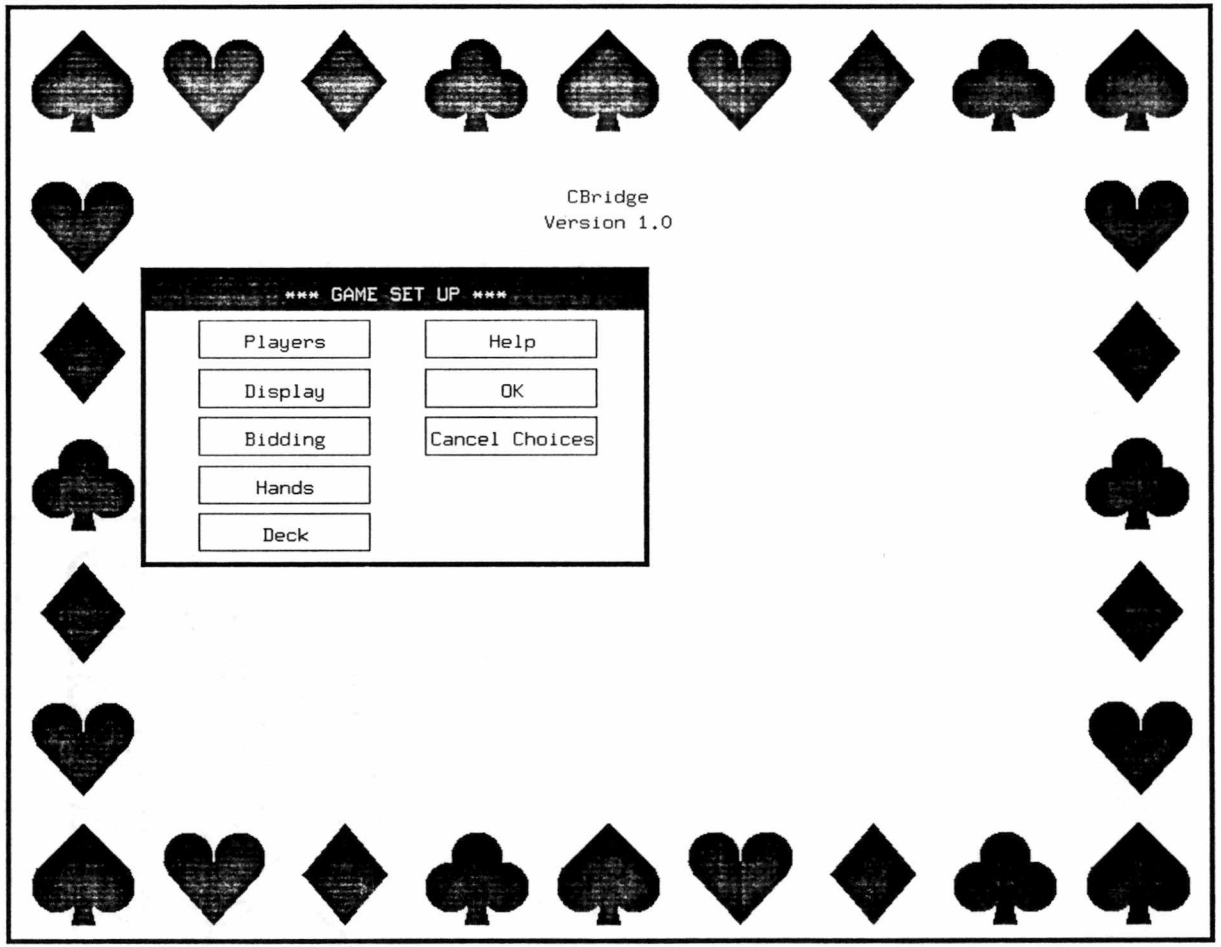


Figure 2.1 Option Setting Menu

Although it is easy to adjust the parameters this way, it can become tedious to do so every time you initiate a series of games. To alleviate this repetition, one enhancement to the option set up design would be to allow parameter settings to be read from a file. CBRIDGE could, for example, check for a default file containing the user's preferred initial parameter settings. If the default file exists, the initial default options are modified. Either way, the user would still have the choice of going through and changing the game options via menu selections.

Game options can be divided into six categories: player, display, bidding, hand, deck, and end game options. A brief discussion of each of these categories and some design alternatives follow.

2.2.1 Player Options

CBRIDGE was designed to play with any combination of human and machine players (see Figure 2.2), although we can envision more uses for certain configurations than others. Testing might be the most valuable function of this parameter, allowing machine players to play against themselves, humans or other machine players. The current default setting (North and South are human players while East and West are machine players) reflects this belief. In addition, this feature could be conveniently used to enter machine players in tournaments. Currently, since we have no fully operational machine player, the only real option is to have all seats "occupied" by human players.

Regarding the players, there is one additional variable, the dealer, that is under the user's control. By convention, the default dealer is initially South. The player to the left of the current dealer becomes the dealer in the subsequent game.

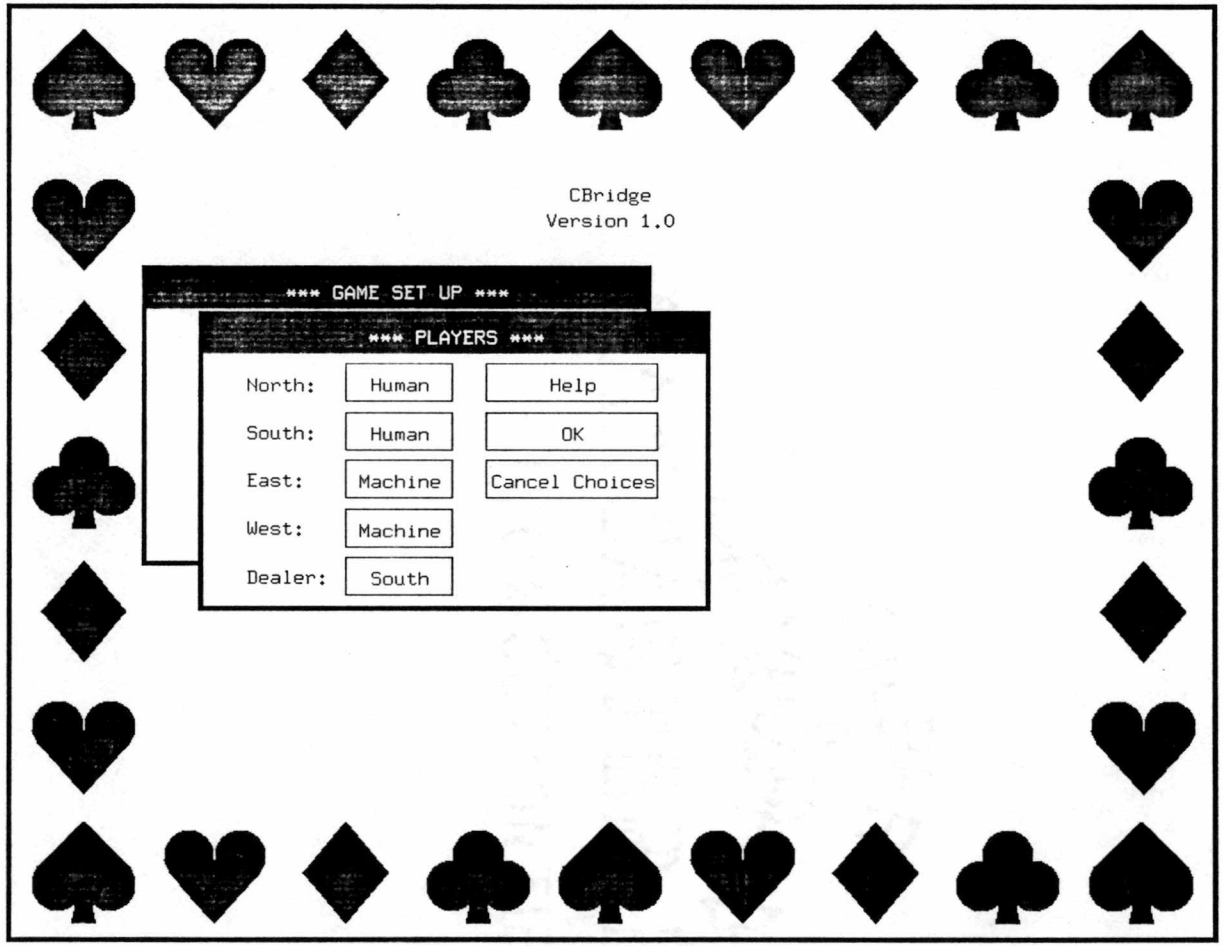


Figure 2.2 Player Options

2.2.2 Display Options

To accommodate all the possible player combinations and to allow for more challenging play, another parameter determines whether each individual hand will initially be hidden or displayed on the screen (see Figure 2.3). The viewing status of a hand may be changed by the user, however, at any time. Picking on a hidden card icon (see East or West's hands in Figure 2.4) opens or displays the hand associated with that icon. To cover or hide a displayed hand, the user simply picks any one of the spade, heart, diamond or club icons (see North or South's hands in Figure 2.4) shown with that hand.

2.2.3 Bidding Options

The auction phase of the game may be allowed to proceed interactively (see Figure 2.5) or it may be fixed. Given the fixed auction option (see Figure 2.6), a complete series of bids comprising an auction are read from a specified file, and then the game begins with the play.

Currently, the file format for a fixed auction requires all bids to be in the order made beginning with West. Although this seems a natural and innocent specification, it also means that South must be the dealer for the auction to be evaluated as a legal one. (The player who initiates the bidding is always the player to the left of the dealer.) A more flexible alternative to this file specification would be as follows. An initial integer or flag value would indicate which player bid first. The actual bids would then follow in the order made. An additional feature could be included to handle "one-sided" auctions in which one team always passed. A second integer or flag could be used to indicate that the bid list would comprise only the first bidder and his/her partner's bids; all other players passed. Otherwise, the bid list would contain all players' bids.

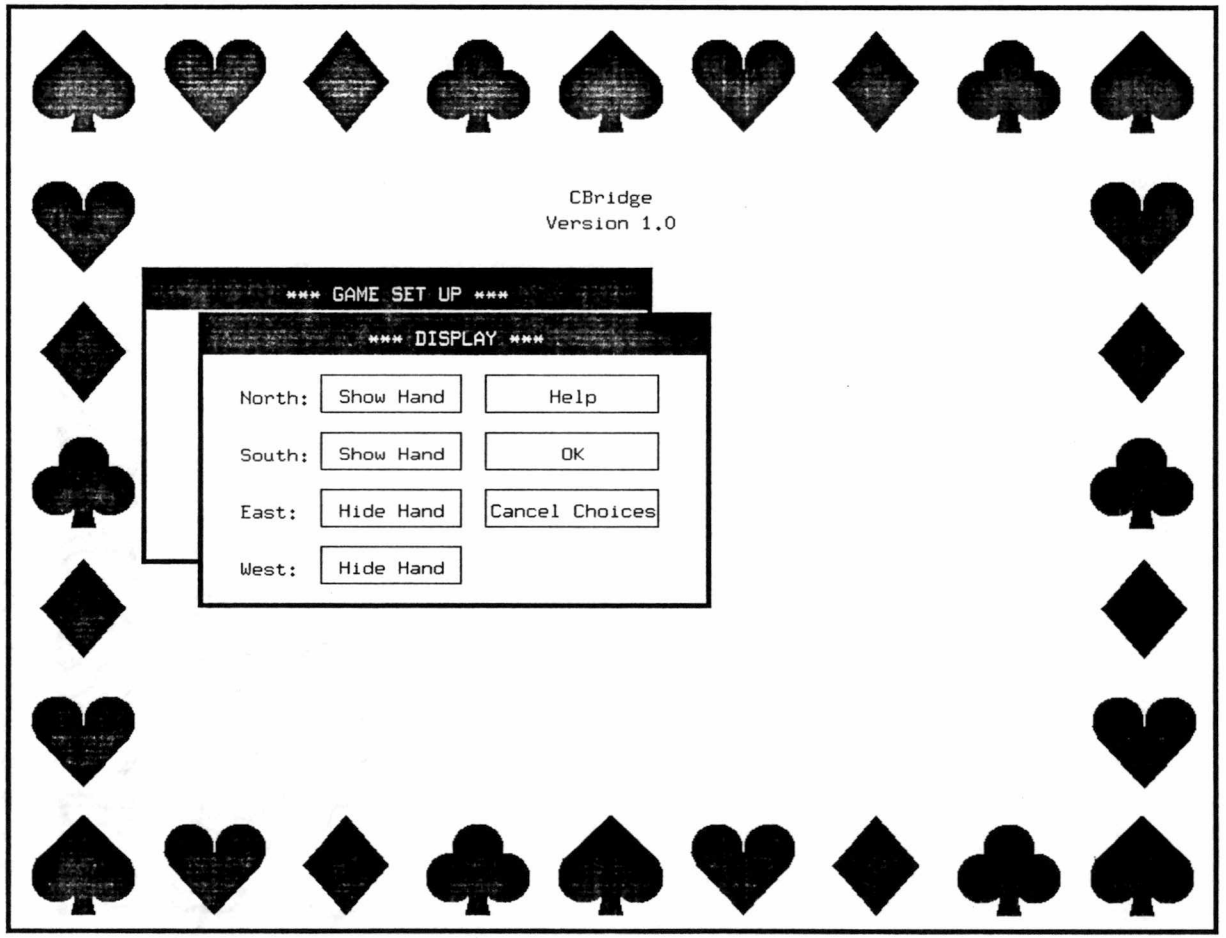


Figure 2.3 Display Options

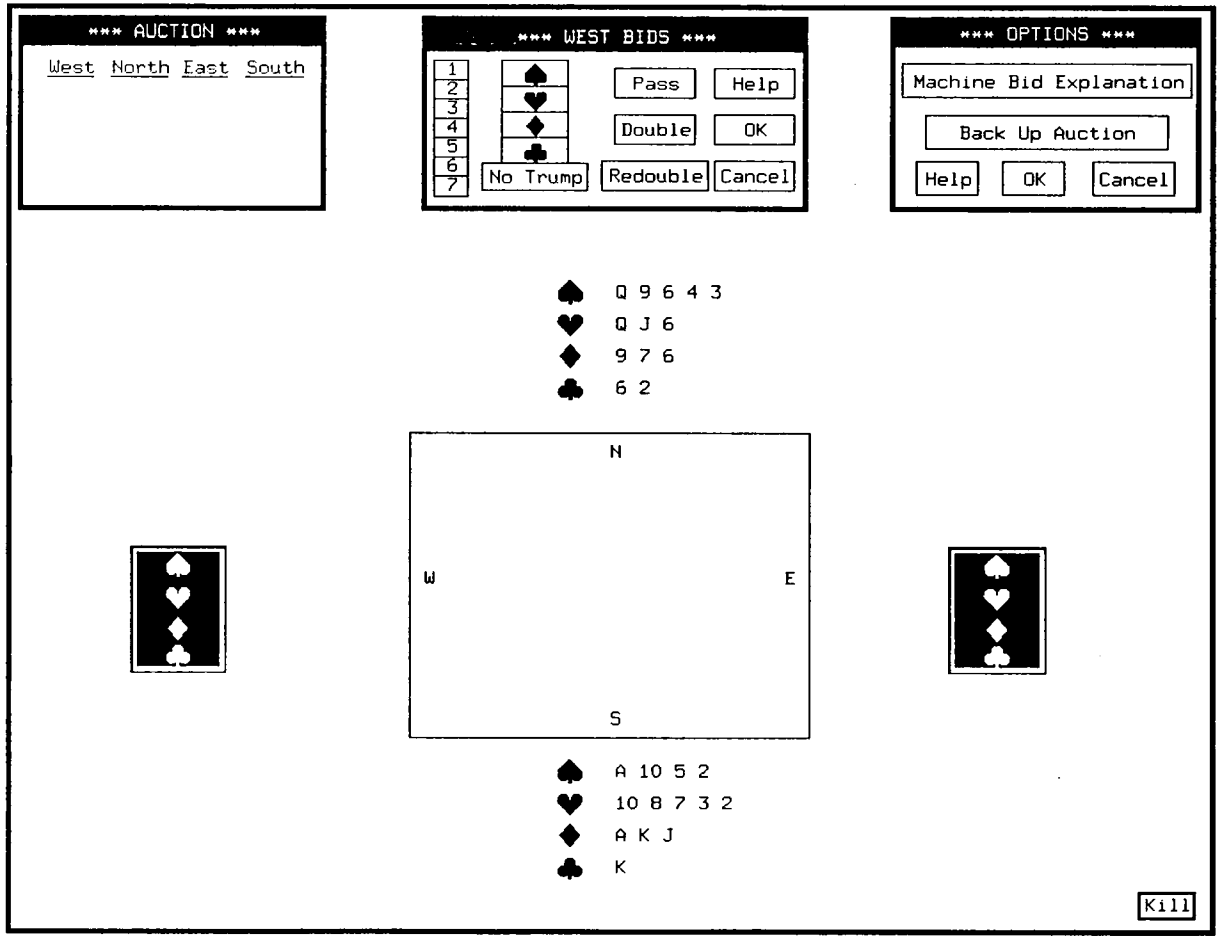


Figure 2.4 "Hide Hand" and "Show Hand" Samples

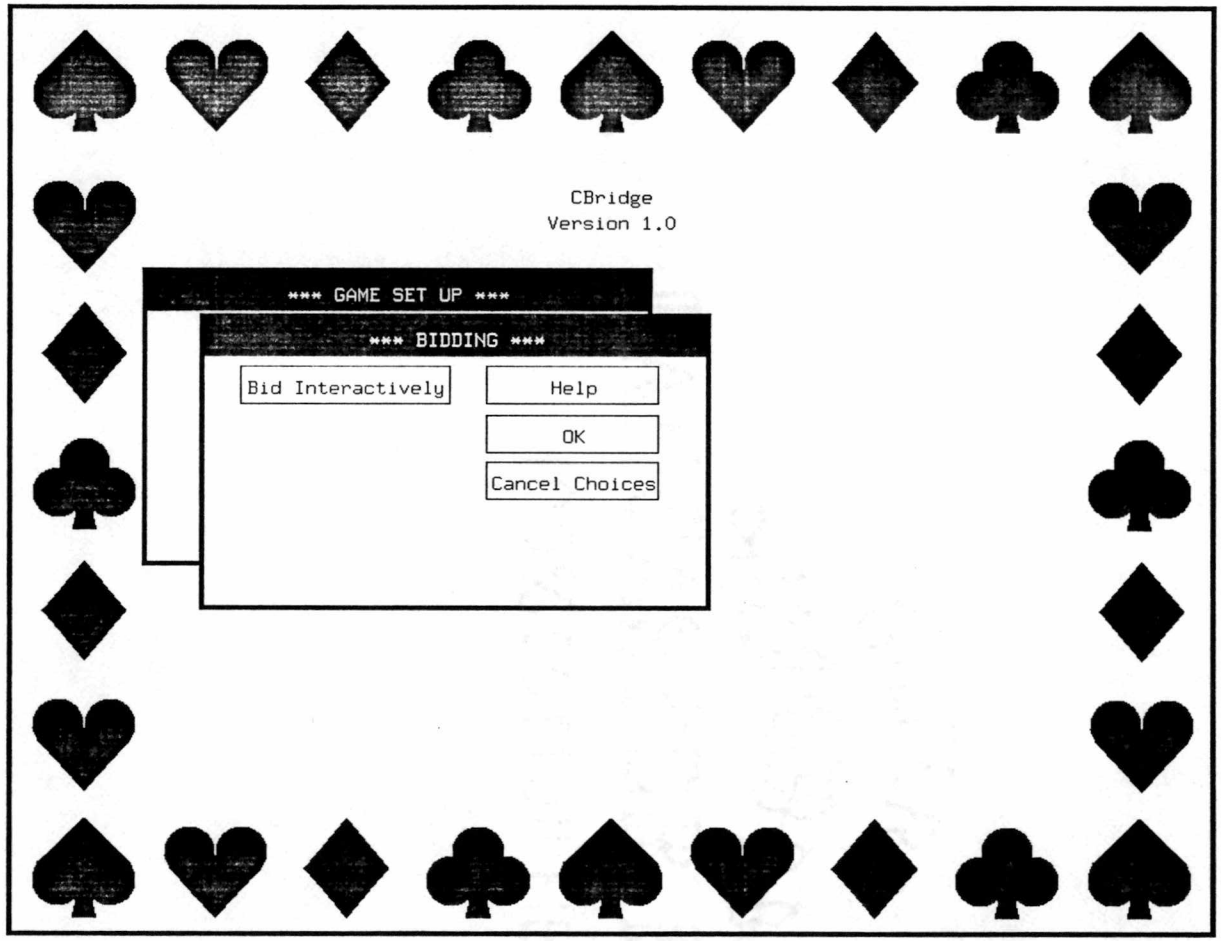


Figure 2.5 Interactive Bidding Option

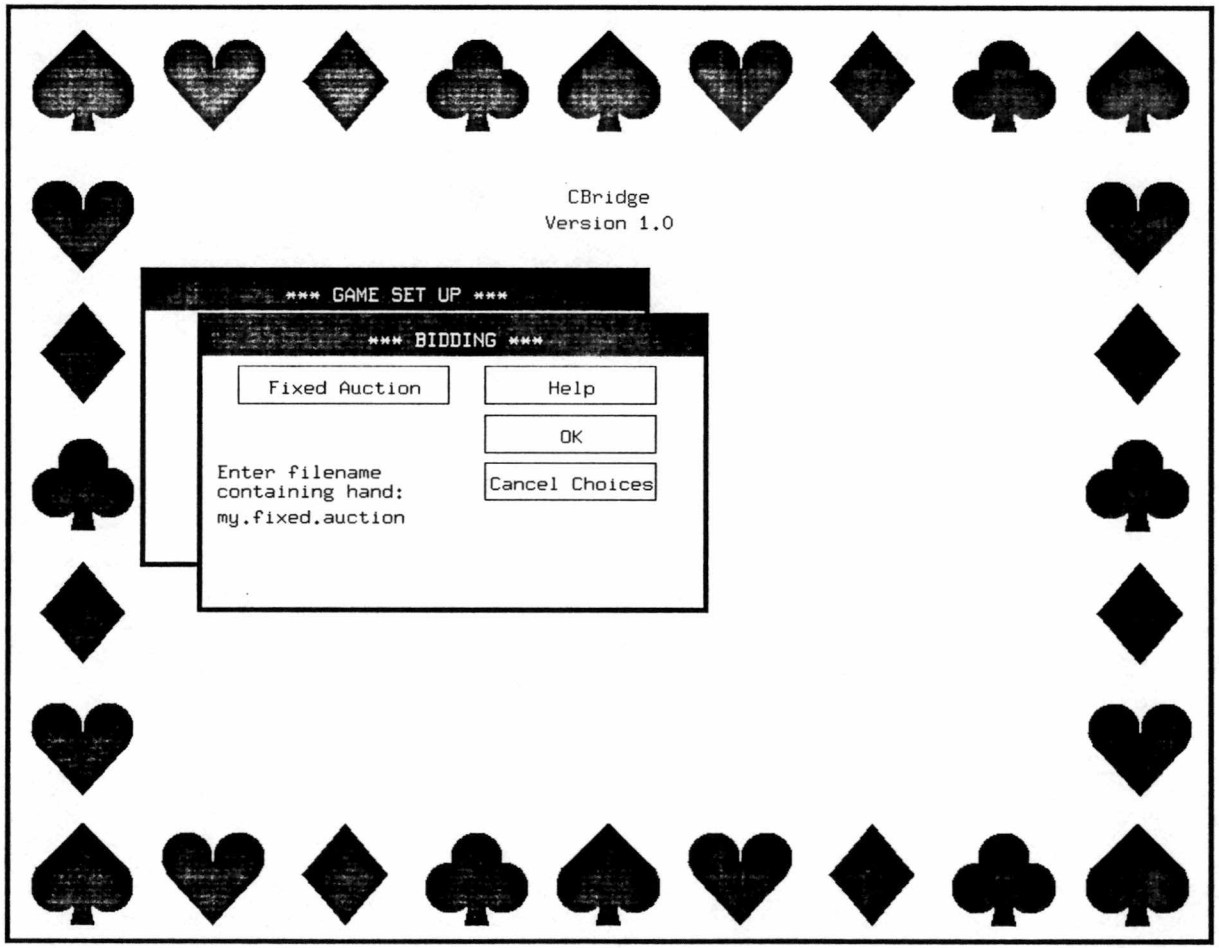


Figure 2.6 Fixed Auction Option

2.2.4 Hand Options

Currently, hands "magically" appear before the players, and there is no visual analogy to dealing. An animated dealing sequence, however, might be an entertaining addition.

Although the default hand type is a randomly dealt hand (see Figure 2.7), a user may choose to play a "Fixed" hand or an "Interesting" hand. Fixed hands are read from a specified file (see Figure 2.8) which currently must be formatted in a manner more convenient for the programmer and computer than the user.

We can envision an improvement in the user-friendliness by changing this format to one more familiar to bridge players (see Figure 2.9). Should the fixed hand file fail to parse, a window might be opened to a template file in the *vi* editor. This template would include instructions regarding formatting rules as well as the basic suit indicators for each hand (see Figure 2.10).

"Interesting" hands have been selected from *Watson's Classic Book on the Play of the Hand at Bridge* [Watson 1969] and stored in a file; users selecting this hand type (see Figure 2.11) play a randomly selected game from this file. The number of entries in this file is limited and should be expanded to include other challenging hands perhaps taken from *The Bridge World*'s [Kaplan 1989] monthly puzzles. Additionally, more information about the games (for example, how many tricks N/S and E/W can win or even hints regarding the optimal line of play) might be stored in this file and optionally made available to the user. The ability to add other games to this file currently would require a recompilation of CBRIDGE as the maximum number of entries in the interesting hand file is "hard-coded". This number defines the range of entries within which a hand may be randomly selected. Were this value stored, for example, as the first number in the interesting hand file, additions

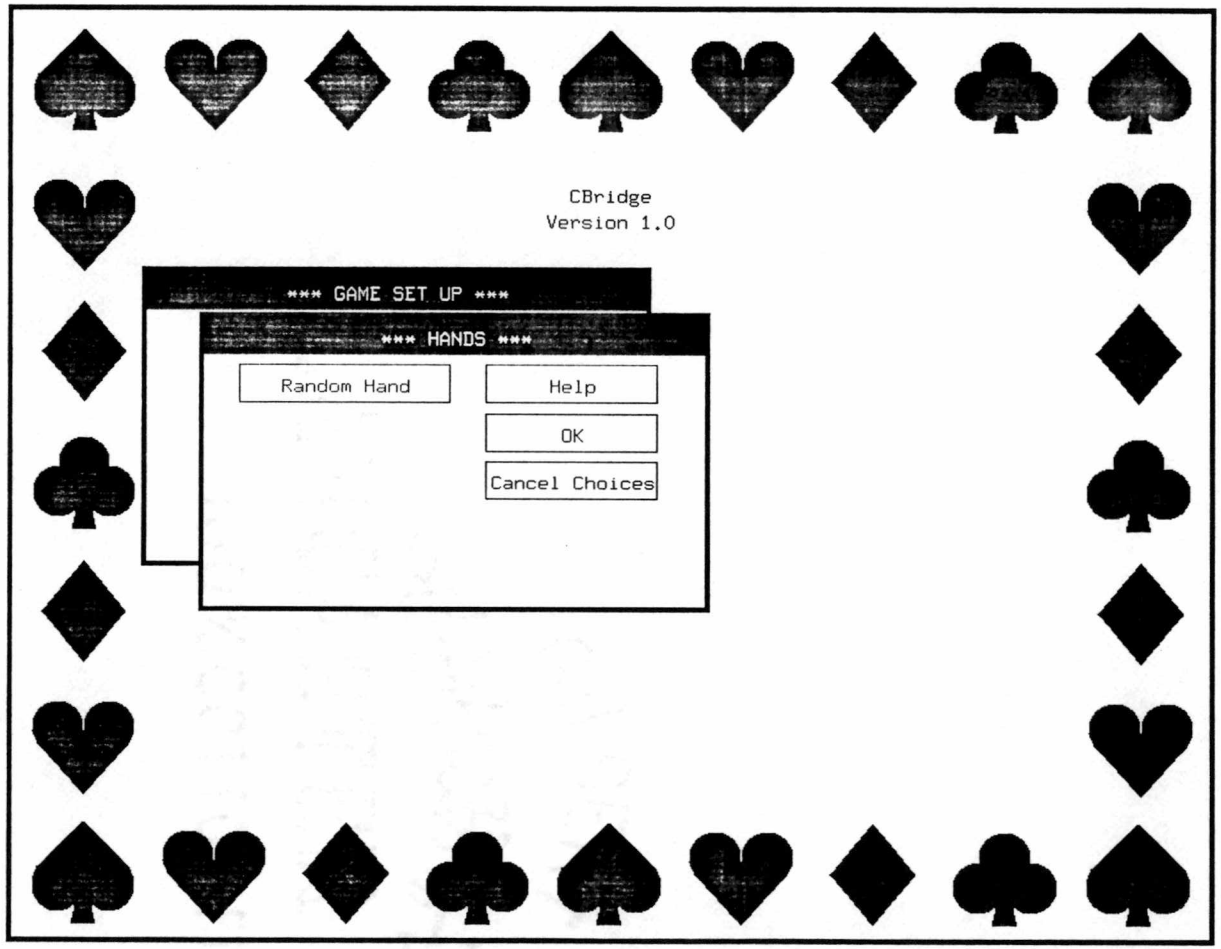


Figure 2.7 Random Hand Option

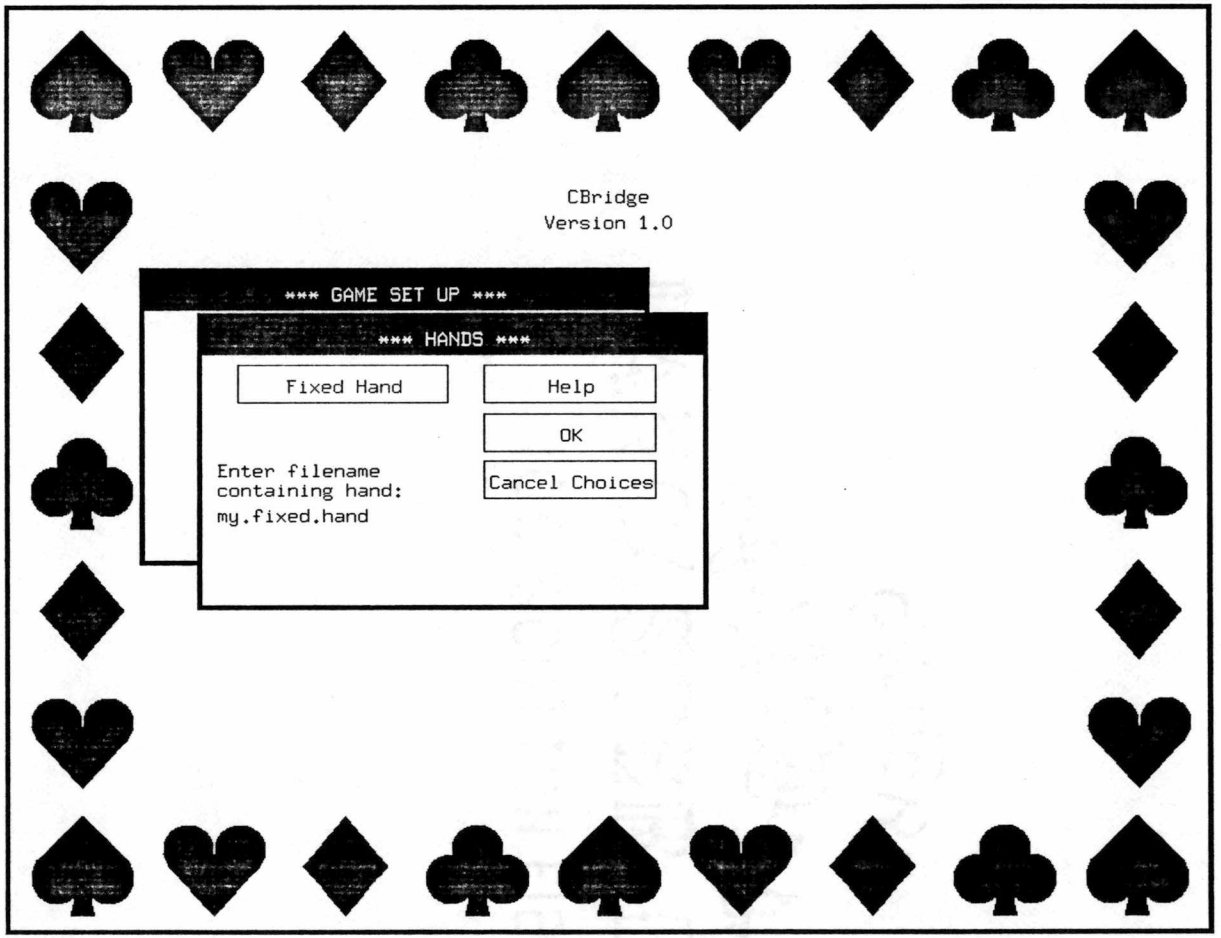


Figure 2.8 Fixed Hand Option

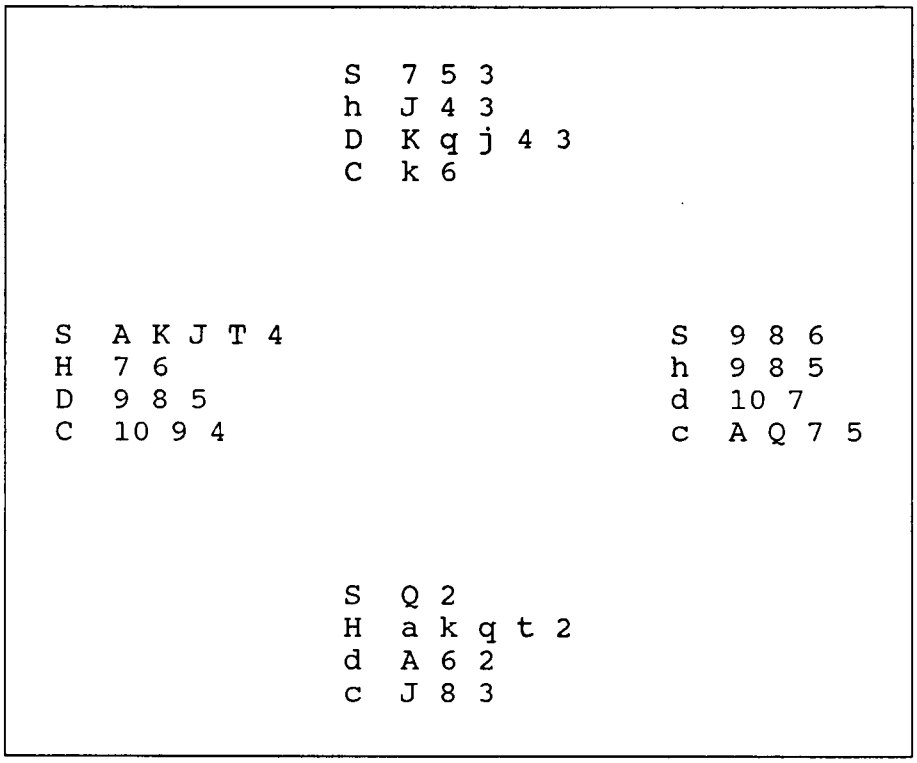


Figure 2.9 Sample Legal File Input

```
* Fixed Hand Template
* Enter all players cards separated by a space.
* (A or a for ace, T or t or 10 for ten,
*  Q or q for queen, K or k for king.)
* S represents spades, h hearts, etc.


                    North
                    s
                    h
                    d
                    c

West                                    East
s                                       s
h                                       h
d                                       d
c                                       c


                    South
                    s
                    h
                    d
                    c
```

Figure 2.10 Sample Template File

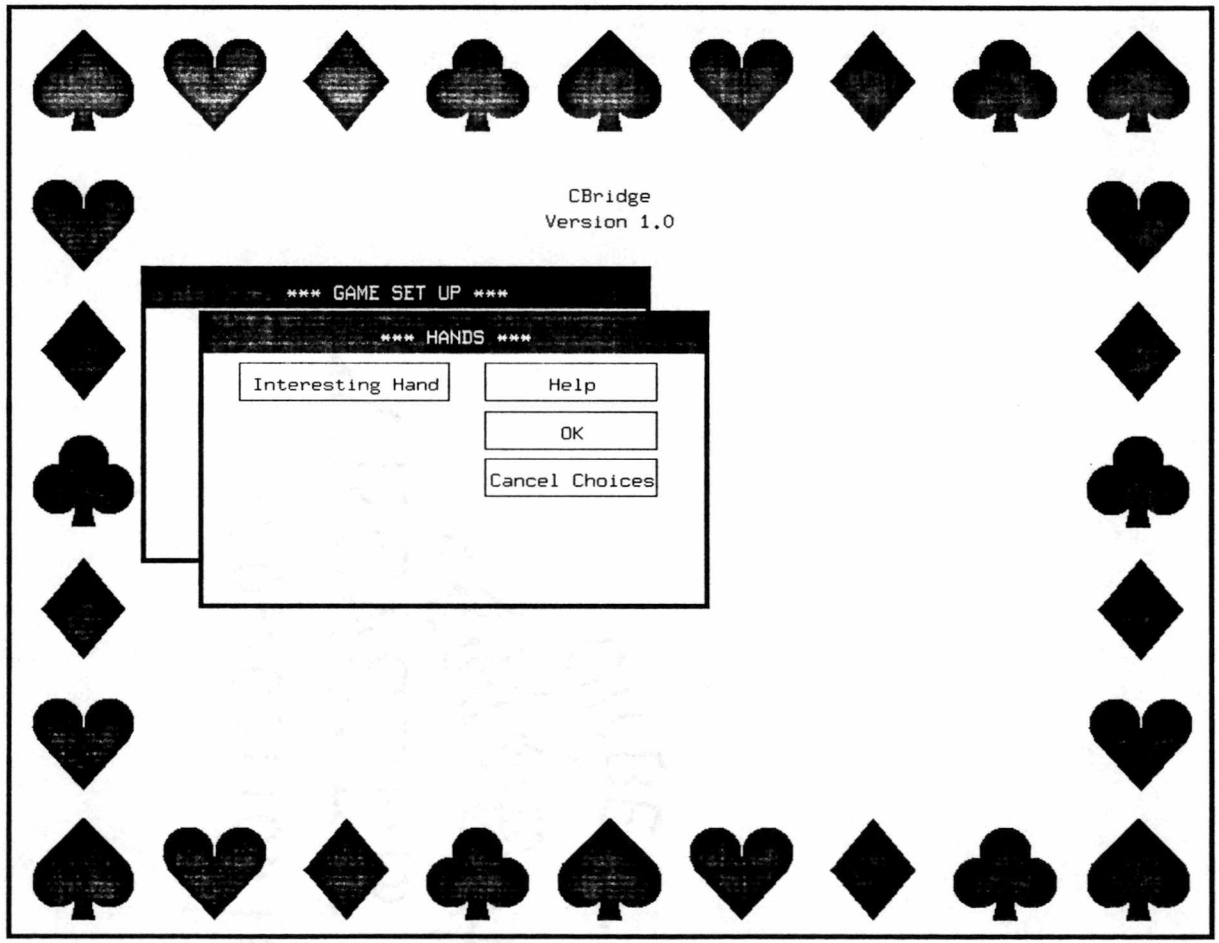


Figure 2.11 Interesting Hand Option

could easily be made manually without requiring a recompilation. Automatic additions might also be made after playing some fixed or random hand upon a user's request.

2.2.5 Deck Options

In our initial design, we wanted the deck size to be adjustable in order to allow interaction and testing of early versions of machine players that would not be capable of playing with a full deck of cards given a limited amount of response time. This option might allow us to speed up the development of machine players in general and more specifically to develop and expand the double-dummy machine player developed by Williams and Mutchler [Williams 1989]. However, allowing a variable number of suits and cards in the deck requires that major changes be made in the rules and flow of the game.

The biggest problem is with the auction. In a standard 4-suit, 52-card deck, the lowest bid of 1 club contracts a team to win seven tricks with clubs as trumps. This same bid of 1 club in a 2-suit, 16-card deck, however, does not make sense. As there are only four tricks possible with 16 cards, the contracting team certainly can not take seven of them. In addition, if neither of the two suits in the deck are clubs, then the notion of clubs as trumps is lost as well. To accommodate varying deck sizes, the rules of the auction would need to be changed.

As the bidding rules would have to change so too would the scoring rules. What, for example, would constitute a slam? If a grand slam is defined as taking all the tricks in the deck, is that still applicable in the case of a 1-suit, 4-card deck in which there will only be one trick? Would a game still be 100 points? Still other complications arise with the hand options. The random hand generator would of course need to be modified, but should there also be interesting hand files for all legal deck variations? What would you

call the 14th card in a 1-suit, 52-card deck? Ace plus one? In light of these considerations and difficulties, the decision was made to require a standard 4-suit, 52-card deck in CBRIDGE version 1.0, but to provide the basic option structure (see Figure 2.12) to easily incorporate variable decks in a later version.

Possible methods of dealing with the variable deck problem would involve making the following rule changes and requirements whenever a non-standard deck was specified. Both human and machine players could be permitted, but whether or not a particular machine player could handle variable decks would be questionable. It would be reasonable to allow only random or fixed hands. The auction and scoring process could be skipped over entirely or interpreted in a new way. Bid levels would be interpreted as a contract to win only that number of tricks (not the usual six plus the bid level). In addition, suits could only be legally bid if the deck contained those suits with No Trump always being a legal suit to bid. The maximum legal bid level would be the number of tricks in the deck, and the highest bid suit would be No Trump as usual.

To score non-standard games, there are several possibilities of varying complexity. Most simply, scoring could be omitted as a non-applicable function. In another scheme, one hand played would be a game, and points for both sides calculated as usual except for the omission of honor and slam points. As mentioned earlier, the concept of a slam does not seem applicable to very small hands, and in the same way, honor points are incongruous. Finally, an adjustable scoring and point scheme might be derived. Either the number of points needed to win a game or the point value of each trick could be modified in such a way as to maintain a relative relationship to standard scoring.

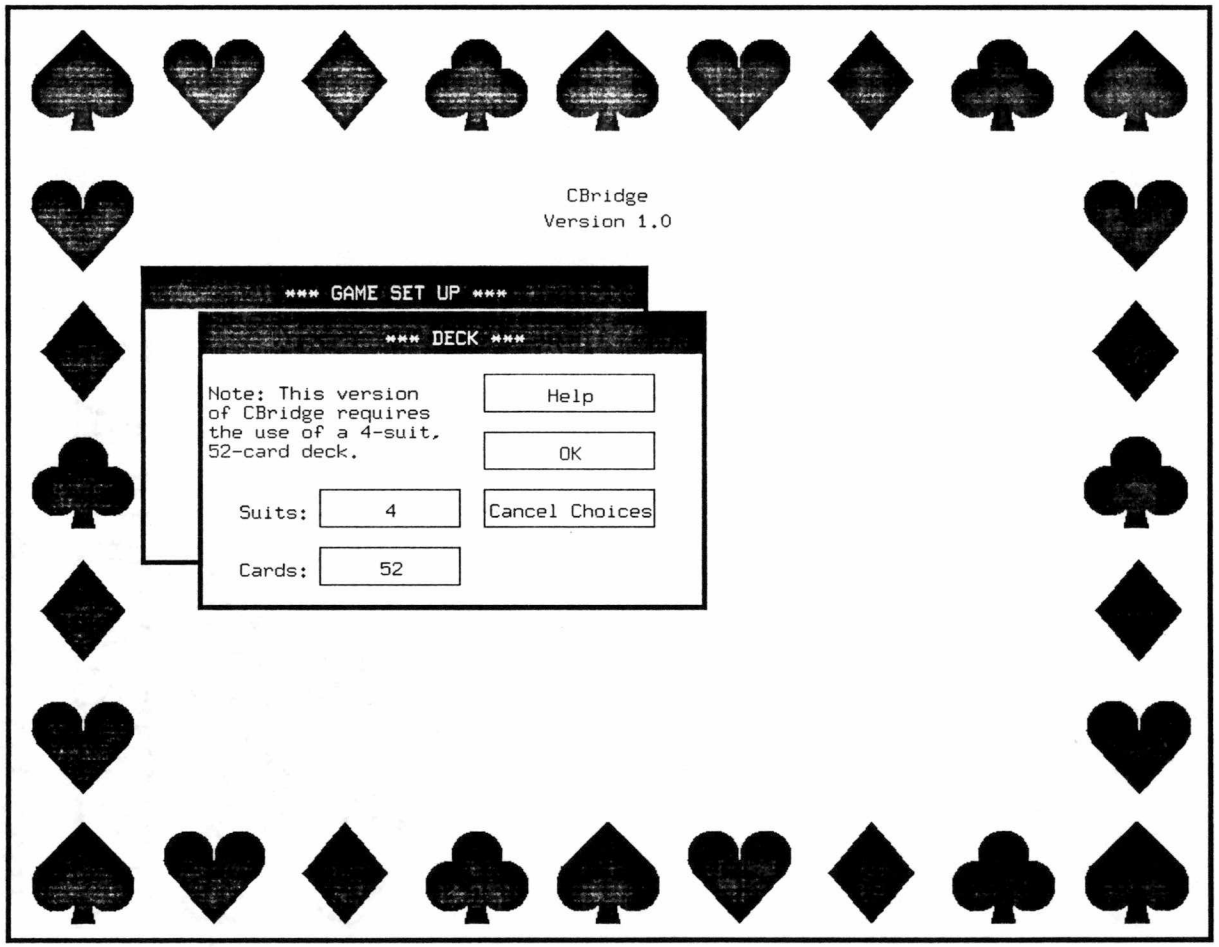


Figure 2.12 Deck Options

For any variable deck scenario, not all possible decks would be legal; legal decks would be constrained by the following requirements. First, the number of cards must be evenly divisible by four so that all players would have an equal number of cards. Second, the number of cards must also be evenly divisible by the number of suits to ensure that there are the same number of cards in each suit. Finally, the maximum number of cards in a suit would be 13, and the maximum number of suits would be 4. This would ensure that no new card bitmaps would need to be created for the play phase of the game.

2.2.6 End Game Options

At the completion of a game, the user currently has only two choices: to play another game or to quit. If another game is played, the player to the left of the current dealer becomes the new dealer, and the user again has the choice of re-setting any of the initial options.

In the development and testing process of this program, however, we have come upon some other features that would be useful if implemented at this point. It would be nice to be able to replay the same hand to try out different strategies. This could easily be done given the current method of marking and unmarking cards as played. The replay might entail setting new parameters or beginning either with the auction or given no change in the previous auction, with the play of the cards. As previously mentioned, a user may want to save a played hand in the interesting hand file, or save the current option and parameter settings in the appropriate default file. All the above end game options would greatly increase the flexibility and usefulness of CBRIDGE to bridge players.

2.3 Machine Players and Communication Issues

One of the primary goals of this project was to facilitate interaction between human and machine players. Initially, the creation of such an environment would be useful in testing machine players, and later, it might enable machine players to be entered into tournaments. At this point, however, it is impossible to fully meet this goal, because the "machine player" is still an unknown entity. Until the development of a true machine player capable of generating legal bids and plays is well under way, we will be missing an essential ingredient necessary for the fine tuning of the machine interface portion of this project. Communication with the machine player developer will be needed to determine exactly what information should be made available to the machine player and how. Despite these obstacles that will have to be addressed at some later time, we have developed what we believe to be the basic features of a machine player interface.

Before getting into more detail about these interface features, let us consider what kinds of machine players are currently available to us. The PRECISION program developed by Wilson, Mutchler and others [Cirelli 1989], is a semi-functional machine bidder; given a hand and a list of all bids made so far in the auction, PRECISION is capable of generating a bid for some hands. As to a machine play or card selector, the closest working model we have now is Williams' and Mutchler's double-dummy player [Williams 1989]. Although it supplies the maximum number of tricks North and South can take against a perfect defense, it could be easily modified to provide the card that should be next played to achieve the maximum payoff. The double-dummy player, however, is very slow (the largest game it can complete on average in under ten hours is with 32 cards), and it requires knowledge of all players' hands. Given that this is the only model of a card selector that we have so far, it will probably be the basis for the future play generators. Based on this

assumption, we will also use this player as a model in the development of the machine player interface. Given these two machine player models, PRECISION and the double-dummy player, now let us examine the machine player interface.

All communication between the interface and the machine players is via files. The two primary advantages of this are language independence and security. First, file communication allows machine players to be written in any language regardless of its compatibility to C, the interface language. Secondly, files provide more security than the alternative, linking programs. Were machine players and CBRIDGE to be linked together into one program, the machine players would have access rights to all game information and could easily cheat. The disadvantage of file communication is obvious; it takes longer.

Whenever it is a machine player's turn during the auction, a query is written to a pre-determined file in the publicly accessible */tmp* directory of the local Sun workstation. The machine bidder is expected to detect and process this file's information and generate a response or bid. In turn, CBRIDGE waits to read the player's response from another file, */tmp/mach_bid_result*, according to a pre-defined format. Machine players have the option to include a short explanation for the bid (or play) they generate in their response file. This explanation or rule will be provided to the user upon his request (see Figures 2.13 and 2.14).

To avoid inaccurately reading a partially written file, both CBRIDGE and machine players are required to write to intermediate files and then execute the Unix *mv* command to rename the intermediate files to the pre-determined file names. Once the intermediate file is completely written to disk, the "atomic" *mv* command simply modifies and renames the links to the file. In this way the message is guaranteed to be completely written to disk

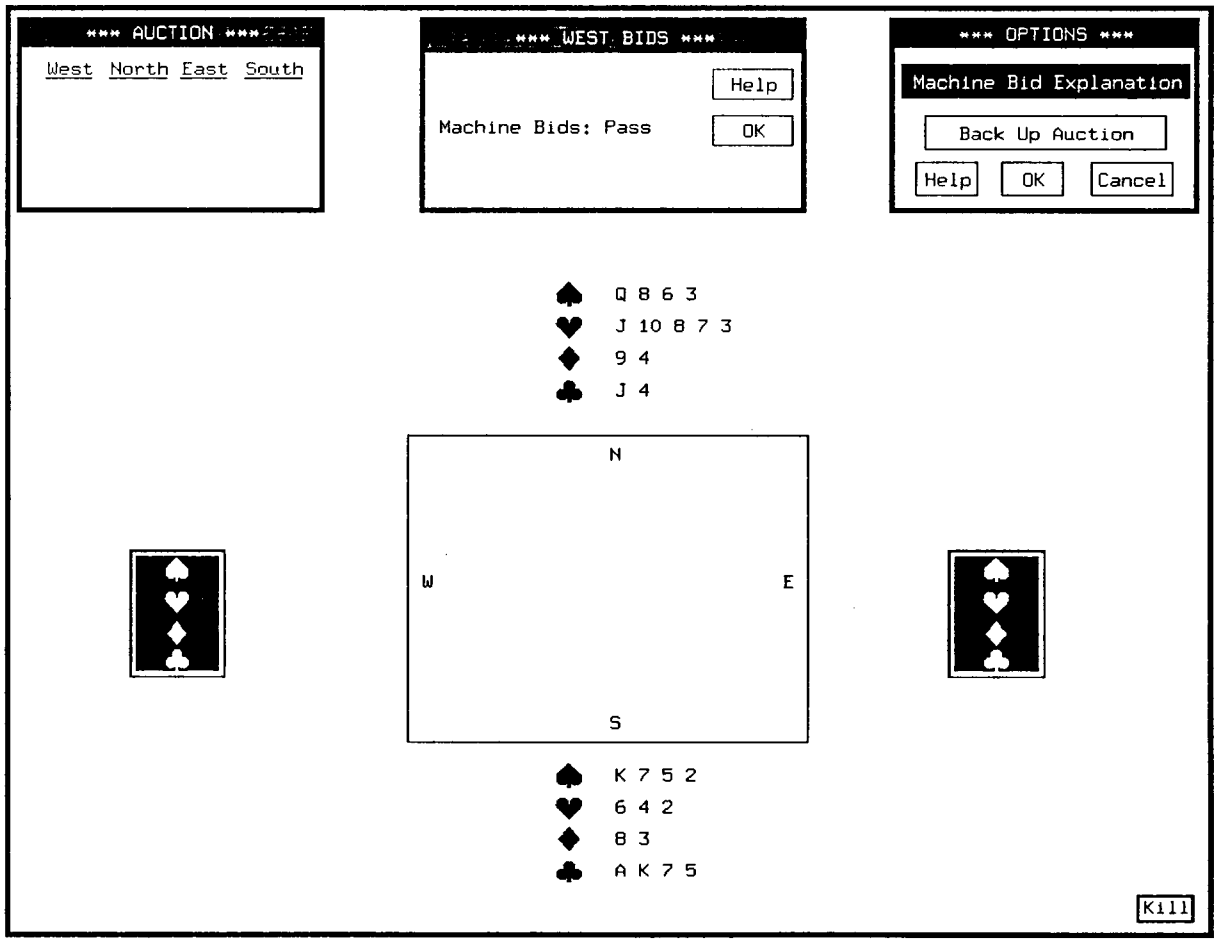


Figure 2.13 Machine Explanation Option

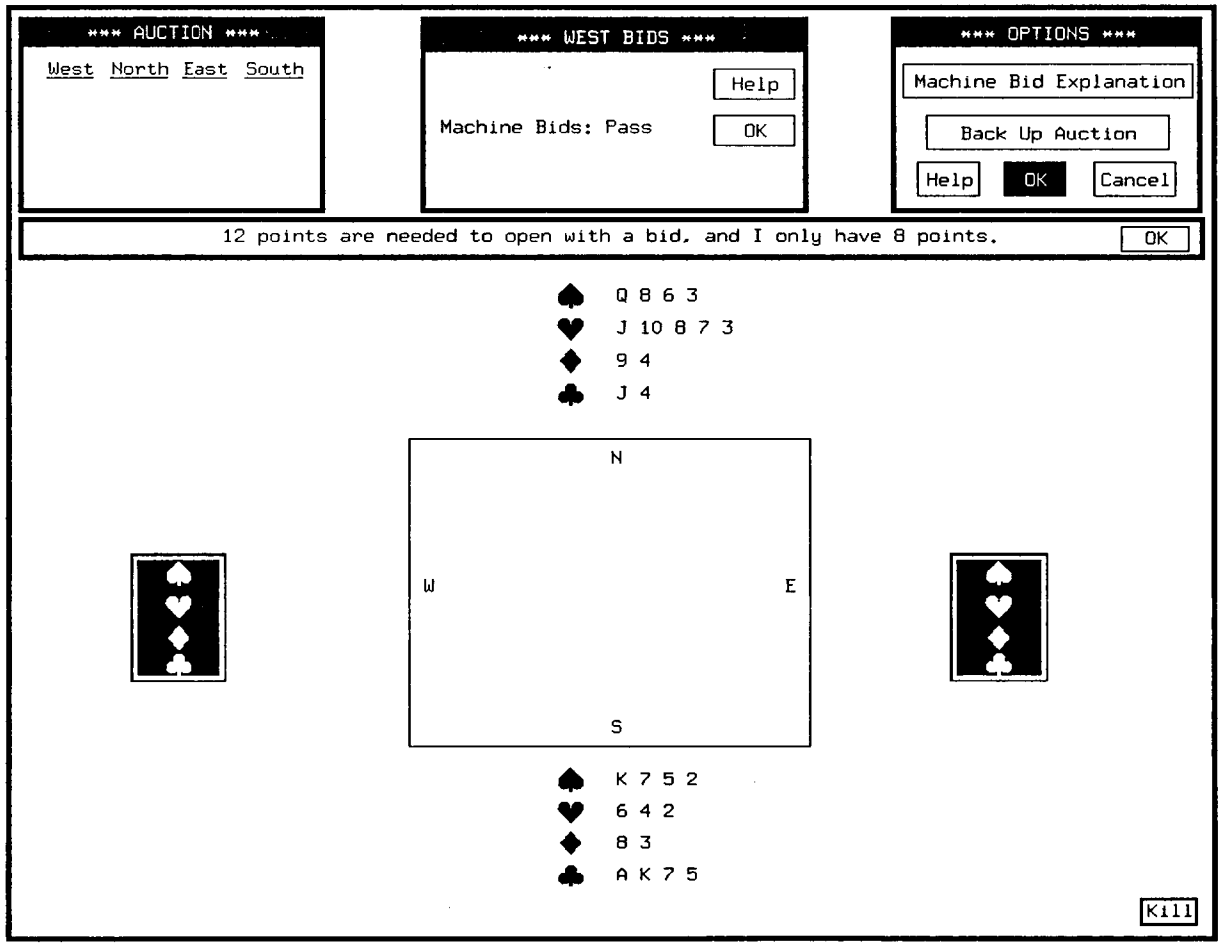


Figure 2.14 Machine Explanation Displayed

when the receiver recognizes and reads the file. An analogous file query and file response process occurs during the play of the game.

What information is included in the query file and in what format must be specified in the procedure *fprint_bid_query*. Although this procedure is currently empty, it will presumably be easy to write the information required by a machine bidder. This would certainly be the case for PRECISION which would need only the hand and the bids made so far. The reason we did not go further with interfacing with PRECISION is that PRECISION's developer has not yet decided how best to read and write information to files in Prolog, the language in which PRECISION is written. As soon as this is resolved, interaction between CBRIDGE and PRECISION should be possible.

Additional communication not currently provided for will need to be established to provide machine players with necessary information that would not have been made available in the bid or bid query. This additional information would be the final contract and the contents of the dummy hand. How this could best be accomplished is unclear because it falls between the cracks of our current dual-concept of a machine player.

With our dual model, it might be acceptable to simply leave PRECISION "hanging" or waiting for a query until the next auction. Assuming both the machine bidder and machine card selector were "cranked up" prior to the auction, CBRIDGE could then, by writing new queries to the new file */tmp/mach-play-query*, begin the query and response process with the machine card selector. It was for this reason that bid queries and play queries are written to different files. The final contract, all appropriate hand information, and any cards played so far in the trick could be included in the first play query. Thereafter, cards played in the previous trick by players coming after the machine card selector would need

to be communicated as well as cards played to date in the current trick. In order to facilitate backing up within a trick and to previous tricks, features which will be discussed in the next section, more information (such as all cards played to date and hand information) might need to be relayed to the machine card selectors at each query. At the end of the game, the machine card selector would "hang" or wait while another game and auction is begun. If the user chose to quit CBRIDGE at any time, special "kill" or "quit" symbols could be written to both bid and play query files to cause the machine bidder and the machine card selector to exit.

Currently, it is not possible for there to be more than one machine player in a game. One machine player could, however, play from one to all four positions or seats. One way to allow for multiple machine players would be to provide for different query and response files for each seat (e.g. */tmp/east_bid_query, /tmp/west_bid_query*, etc...) and require that each "cranked up" machine player knows its position and thereby its query and response files.

2.4 Error Handling and Backing Up

Error handling and backing up features allow the user to more confidently explore CBRIDGE facilities as well as try out various game strategies.

2.4.1 Error Trapping

Currently, there are no penalties for human players attempting to bid or play out of turn. If a human makes an illegal bid or play, an error message with some attempt at a diagnostic appears. The player may then enter another selection.

Error messages generated by an illegal bid include a reason for the error as well as recommendations for making a legal bid. For example, if West opens the auction with a bid of one diamond and North then bids one club, North is informed that the bid level is too low and a bid of two clubs is suggested (see Figure 2.15). Similarly, if an illegal card is played, a rule is generally stated along with a suggested "fix". For example, if East leads a trick with the 5 of diamonds and South, having diamonds in his or her hand, attempts to play the 2 of hearts, then South is informed that he or she must follow suit and play a diamond.

Unlike human player errors, machine player errors are not tolerated. If a machine player makes an illegal bid or play, the entire program exits. The assumption behind this unforgiving response is that once a machine player makes an illegal move, it is likely to continue to do so. At some point in the future, a machine player override option might be convenient here for testing purposes.

2.4.2 Backing Up

During the auction, back tracking is easily accomplished by simply selecting the "Back Up Auction" option (see Figure 2.16). Each time the user executes this function, the most recently entered bid is removed and the bid owner is allowed to re-bid. This process would involve no extra communication to our current machine bidder PRECISION, because it generates each bid independently, based only on the hand and bid list information provided with each query.

The same concept of back tracking one step at a time exists within the play of the game. Currently, it is limited, however, to within a single trick. When a human or machine player selects a card for play, an icon of the card appears on the table area of the screen. To

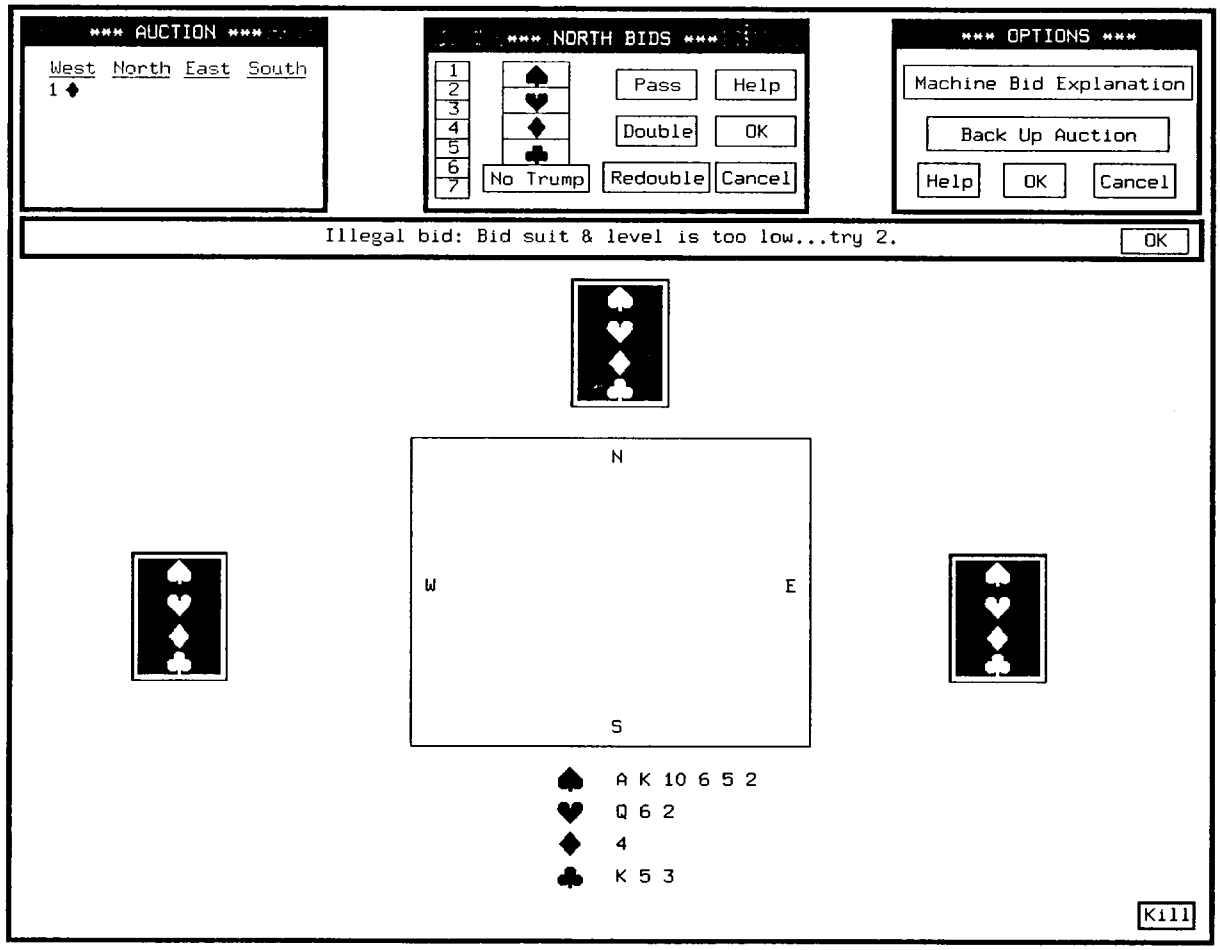


Figure 2.15 Sample Error Detection and Diagnostic

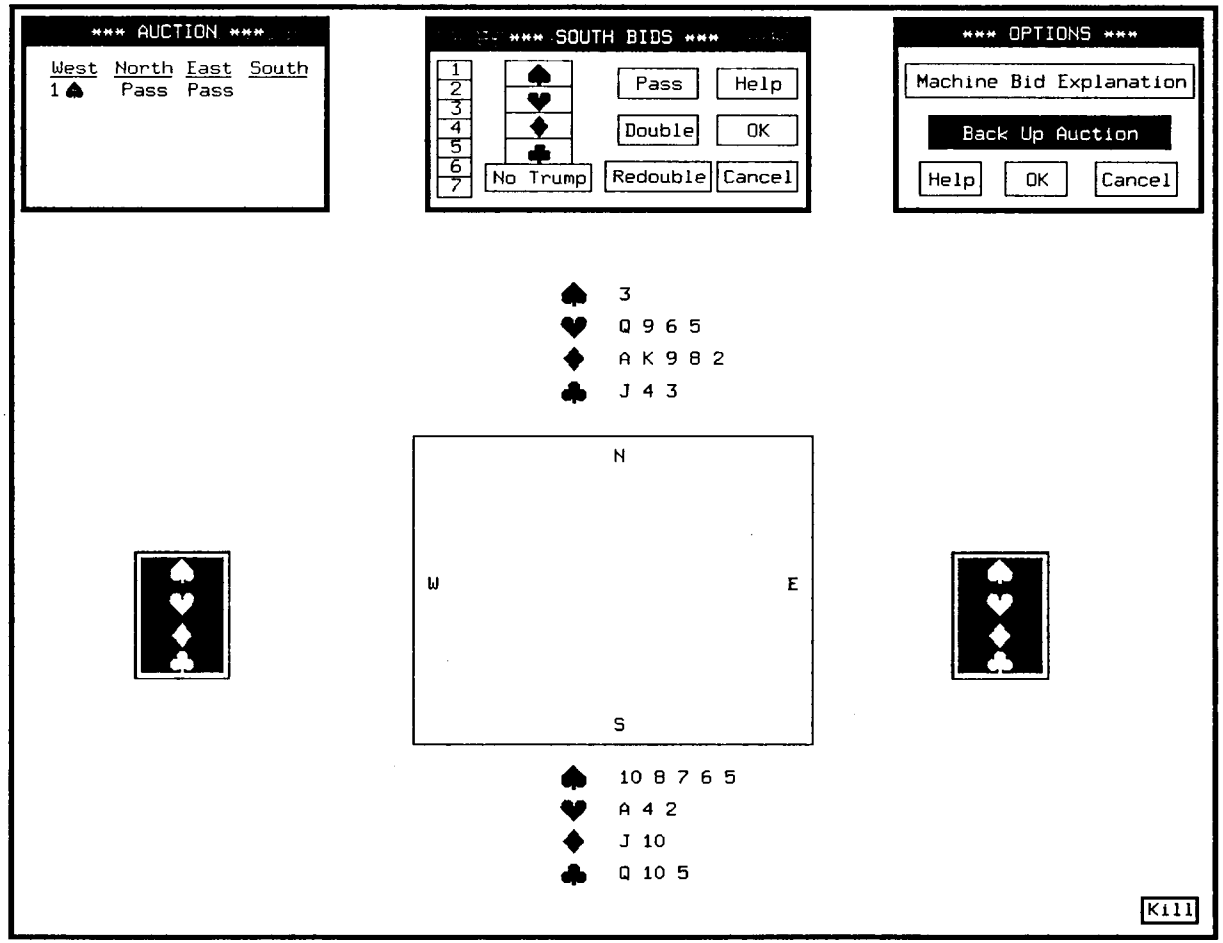


Figure 2.16 Back Up Auction Option

return a selected or played card to its owner's hand and give the owner the opportunity to play another card, the user simply picks the card icon on the table with the mouse. The card will not be returned to its owner's hand if it was not the most recent appearing icon on the table. In other words, backing up the play within a trick must occur in the reverse order that the cards were originally played in.

Although this feature is useful for human players, it can be redundant for machine players. Assuming that a machine player will make the same card selection given the same set of circumstances, the option to re-select a card does not make sense for machine players. In light of this, if it is a machine player's turn and the user elects to return the card to the player's hand (i.e. picks on the machine player's card icon), then it becomes the previous player's turn. This works well if one team of players are machines and one humans. It does not, however, make much sense if all players are machines or if two consecutive players (e.g. North and East) are machines. A better approach might be to revert to the most previous human player's turn in the trick, if a human player has played in this trick. As briefly mentioned earlier, a machine card selector might simply be provided in each query with all pertinent information for the card selection, thus making additional communication as a result of a back up unnecessary. It would, also, however, increase the amount of processing the selector would need to do and slow down response time. Another alternative would be to provide the number of steps backward the play had been taken and let the machine player take care of the rest.

Back tracking to previous tricks is not currently an option. This might be implemented in two ways, either via a numerical or pictorial representation. A "Back Up Trick" option similar to the "Back Up Auction" might be made available after playing the first trick in the simplest numerical scheme. Given a user selection of this option, a menu offering

similar to Figure 2.17 (with tricks already played being the only choices visible) could allow the user to specify the number of the trick he or she wishes to back up to and re-play.

The pictorial back up scenario would require many changes to CBRIDGE, but would result in a program that more closely resembled tournament style duplicate bridge. Each seat's played cards would be kept face down in a row in the order in which they were played. Each card would be placed either vertically if North/South won the trick that card was played in or horizontally if East/West won the trick. If the user should pick on a card in such a stack when the "Back Up Trick" option was not selected, the card could flip over to show its value. A user's pick on a card in this type of stack while the "Back Up Trick" option was selected could cause the play to back up and continue playing at the beginning of the specified trick in which the selected card was played. For a machine player, back tracking some number of tricks could be handled in the same manner decided upon for backing up within a trick.

2.5 Defaults and Automatic Play

Generating defaults and operating in automatic mode are two useful features for human players. These features can

1. help teach the game to novice users,
2. increase the pace of the game for more expert users, and
3. decrease the amount of effort required to make bids or plays.

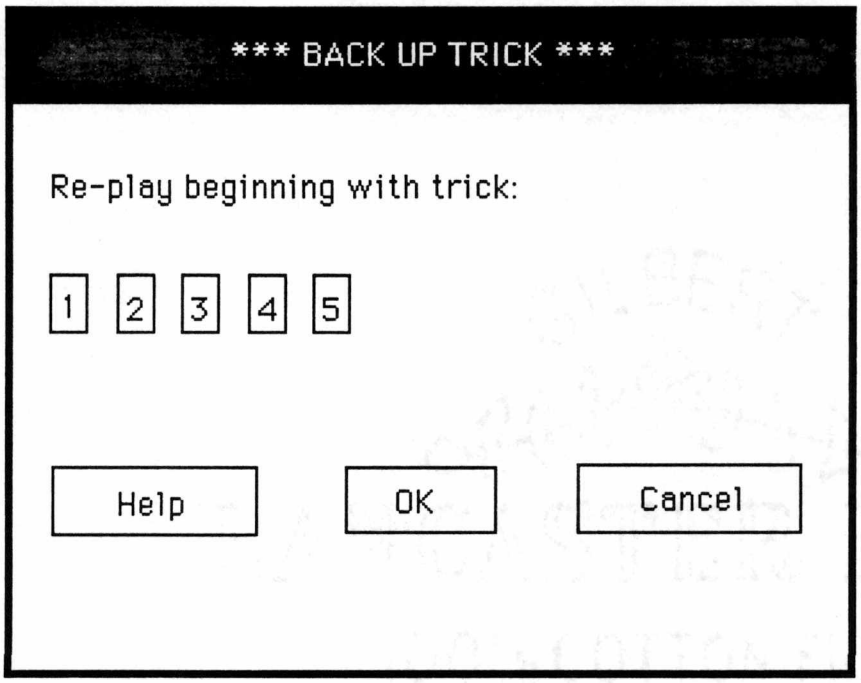


Figure 2.17 Proposed Back Up Trick Option

2.5.1 Default Bids and Card Selections

During the auction, pressing the middle mouse button during a human player's turn causes a default bid to be selected. The rules for this selection are very simple, but they demonstrate how default choices can be made available to the user. The first default choice is always a pass, and the second is the least next higher bid. Requesting a third default choice causes the same two bid sequence to repeat. At any time during the selection process, a default bid may be overridden by a user selection.

During the play of the game, the middle mouse button causes a default card to be selected for play. The rules for selecting a default suit and pip (the face value of a card) are somewhat more complex. The following series of evaluations determines the suit from which the default card will be selected.

```
if (this player leads the trick && has a trump card) then
        play from the trump suit
else if (this player did not lead the trick && has a card in the suit led) then
        play from the suit led
else if (this player has a card in his longest suit) then
        play from the longest suit
else
        play from the lowest suit in which this player has a card
```

A card is then selected from this suit according to the number of the default choice. Once a default suit and pip are selected they are treated as an already played card if more default cards are requested. The first default choice is the lowest card in the selected suit, and the next choice is the highest card in the selected suit. Thereafter, cards are selected low to

high. If all remaining cards in a players hand have been offered as a default choice and the user continues to request a default choice, the cycle of choices repeats. As with default bidding, a default play may be overridden at any time during the card selection process by a user card selection (i.e. picking on a card).

Both default bidding and default card selecting would benefit from an improved set of rules that makes use of more expert bridge knowledge. Incorporating some kind of point and distribution evaluation of the hand could be useful in generating a bid. This information would also need to be "tempered" by knowledge of bids made by other players so far in the auction. In fact, such an expert system is being developed by Wilson and Mutchler and is based on the Precision bidding system. Although this project is ultimately intended to become part of a machine player, it could be accessed as a sophisticated default bid generator as well.

Improvement of the rules for the default play selector could become as complex a task as a person had time for. After all, a very expert player selector is the kind of machine player Mutchler is hoping to develop. If this project eventually succeeds, then perhaps a modified version of the machine player (i.e. one that responds in a very limited amount of time and perhaps with more "guessing" as a result) could be tapped as a source for default play selection. At a more basic level, the play selection process could benefit from the expansion of the current very broad rules to include more specific knowledge about the appropriate situations for the application of the rule. For example, the first rule for suit selection says to play a trump if a player is leading the trick. What is more meaningful is something like the following.

```
if (this player leads the trick && is part of the contracting team) {
        if (defensive team has trump cards && my partner or I have
            a trump winner)
                play a trump
        else if (defensive team has no trumps && I do not need to save
                my or my partner's trumps to beat opponents "winners")
                play a trump
        else if (this player does not lead trick && does not have card
                in suit led && has a trump)
                play a trump
}
```

This example illustrates another concept, the idea of offensive and defensive playing, that should be taken into account. Other rules of thumb like second seat plays low and third and fourth seats play high, could be incorporated. More complex concepts like finessing, ducking and squeezing might possibly be implemented, but only if they would not slow down the near instantaneous response time required for default play generation. Given this time constraint, perhaps finessing or any single one of the more complex strategies, could be incorporated.

2.5.2 An Automatic Bidding and Playing Mode

An interesting additional feature for both the bidding and the play phases might be an automatic auction. During the auction, this might simply make the default bids for both human and machine players until the end of the auction or a human enters some override signal (for example, a button press). If the current default bidding scheme were used in this manner, it would be the equivalent of an "All Pass" option since the first default bid

choice is always to Pass. This would be convenient near the end of the auction, but not very useful before a final contract was made. To make this kind of automatic bidding more valuable, the default bid generator would need to be modified as discussed previously at least to the extent of not always selecting the Pass option first. Again, a more sophisticated automatic auction could easily be implemented by interfacing with the Precision expert bidding system being developed by Wilson and Mutchler.

Automatic play could be implemented at a variety of levels. Most simply, each player's card could be played without waiting for a user's confirmation to produce some form of automatic play. In the same vein, default play selections could be made for all players for one trick. The user might then have the opportunity to accept these plays or back up and change them manually. At the next level of complexity, default play selections could be made until the end of the play or a human enters an override signal. With an even more intelligent system, the bridge concept of *claims* might be implemented as some type of automatic play. A claim is when one player shows his/her remaining cards and asserts that the outcome of the game is certain. For example, if the following conditions exist:

- North has the lead and has three trumps,
- all other trumps have been played, and
- only three tricks remain to be played,

then North could show his/her cards and claim that the remaining tricks are his. If a claim is disputed by another player, play continues as usual. Otherwise, the game is over, and the final tricks are accepted as having turned out the way the player making the claim asserted.

There is an issue related to trick taking that should be mentioned and should be made a game parameter. After each trick, there is currently a two second pause designed to give human players time to see all the cards played in that trick. The pause length is hard-coded, but for both interactive and automatic play, the user ought to be able to specify either the fixed length of the pause or that the pause will be of variable length terminated by some user input (e.g. a button press).

2.6 Summary

CBRIDGE supports both human and machine players and provides a variety of instructive, useful and fun features. Numerous parameter options define the type of game to be played. For example, display options allow the user to play double-dummy bridge where all hands are visible or standard contract bridge in which only one hand is always visible to the player. There are event features to allow cheating or peeking at other players' hands. In addition, the auction may be bid interactively or read from a file, and hands may be randomly generated, selected from a file of particularly interesting hands, or specified by the user. During the play of the game mechanisms exist for generating default "moves" and for backing up. HELP is available throughout the game, and human players are almost always given a recommended action to correct any inappropriate response made. In its current state, CBRIDGE is a fun and user-friendly interface for human bridge players and a useful one for developing and testing machine players. In addition, CBRIDGE has been designed to facilitate many enhancements which may be implemented at a later time to provide an even better environment for human and machine interaction.

CHAPTER 3

PARALLEL ALPHA-BETA ALGORITHM STUDY

3.1 Introduction

The work in this chapter is an extension of the work done by Williams and Mutchler in developing a double-dummy machine player [Williams 1989]. As the double-dummy machine player they developed is only capable of completing on average a 32-card game in less than 10 hours, more techniques must be derived to speed up the search process, and one possibility is to search in parallel. In the ongoing search for faster computers and faster programs, the current trend in technology is to focus on concurrent processing. Although massively parallel computers are being built, the real challenge is to harness that power by developing parallel algorithms to solve problems faster. In this chapter, we will be looking at how some parallel algorithms might improve the performance of the alpha-beta search conducted by Williams and Mutchler's machine player.

Most evaluations of parallel alpha-beta algorithms described in the literature [Akl 1982, Finkel 1982, Marsland 1982, Marsland 1985, Hyatt 1989] have been limited to the algorithm's performance on chess-type game trees which have a uniform and generally large branching factor and a limited depth. Bridge trees, however, are relatively narrow and deep trees with a varying branching factor. The specific question we will try to answer in

this chapter then is "Do the reported benefits of the parallel alpha-beta algorithms derived from a consideration of chess-type trees apply more, less or equally to bridge trees?" In seeking to answer this question, we do not claim to have conducted a rigorous quantitative study. We do believe, however, that the set of experiments we have done strongly indicates that current parallel alpha-beta search algorithms are not as effective on bridge trees as they are on chess-type trees and that they will provide only slight improvements for our machine player.

The particular experiments we did were with three parallel alpha-beta algorithms: Baudet's aspiration algorithm [Marsland 1982], Finkel and Fishburn's principal variation algorithm [1982], and Marsland and Campbell's principal variation splitting algorithm [1982]. Since each of these three algorithms make calls to sequential alpha-beta, we implemented the parallel algorithms in Ibuki Common Lisp in order to utilize the serial alpha-beta routines Williams [1989] developed in this version of Lisp. Note that in our implementations of the three parallel alpha-beta algorithms, we substituted a version of serial alpha-beta that includes the good-ordering and strict bucketing techniques developed by Williams and Mutchler [Williams 1989] in place of all calls to "standard" alpha-beta. A discussion of serial alpha-beta and our modified version that includes the good-ordering and strict bucketing techniques follows.

Sequential alpha-beta is a left to right, depth-first search that cuts off and does not explore certain branches in the game tree. Intuitively, the unexamined possibilities are those that follow a move an opponent would not make. More specifically, alpha-beta implements the following pruning strategy:

At MAXIMIZING nodes:

if there is a MIN ancestor whose current value is less than beta,

cut off all other branches at this node

At MINIMIZING nodes:

if there is a MAX ancestor whose current value is greater than alpha,

cut off all other branches at this node

The alpha and beta variables mentioned above are the bounds within which the true minimax value is estimated to lie. Alpha and beta are typically assigned initial values of minus infinity and plus infinity respectively, and they are updated as the search proceeds. The interval between the original alpha and beta values is called the window.

The modified version of alpha-beta that we used in the three parallel alpha-beta algorithms differed from standard alpha-beta in two ways. First, a heuristic was used to try and order the nodes generated in a best to worst manner. In this way, the node or move that is expected to be best is examined first. This heuristic technique is called good-ordering. Ordinarily when the best moves are looked at first, the most pruning is done. The second addition made to standard alpha-beta is the strict bucketing technique. This strategy incorporates into the game tree search the basic bridge idea that some cards have equal trick taking value. For example, if one player has both the 5 and 6 of clubs, playing one of these cards will have the same impact on the trick as would playing the other card. Both the 5 and 6 of clubs will either win or lose the trick. In this case the 5 and 6 of clubs are grouped or bucketed together, and only one branch is taken in the game tree. In addition to bucketing consecutive cards, non-consecutive cards may also be bucketed if all intervening cards have already been played in a previous trick.

In practical terms, any bridge playing program using alpha-beta search that we develop would use the good-ordering and strict bucketing techniques. For this reason we replaced all calls in the parallel algorithms to standard alpha-beta with a call to our modified alpha-beta. Consequently, we also used the performance of our modified version of alpha-beta as the standard against which we compared the performance of the parallel alpha-beta algorithms.

In order to create a parallel environment for our experiments, we implemented a parallel controller program, PLISP, to provide communication between Lisp programs running on different Sun workstations via shared-access files. Although this communication is very slow and practically allows for only coarse-grained parallelism, we believe the relative merits of the parallel search algorithms as they apply to bridge specific search trees can be determined by keeping track of such statistics as CPU time, nodes expanded and nodes generated. We collected a variety of data during our experiments, but the measure we will be using to evaluate an algorithm's performance is CPU or run time. This run time measure will not include any communication time for two reasons:

(1) our communication times are greatly affected by the amount of traffic on the Sun network, and

(2) the communication time with PLISP is much greater than with any "real" parallel processing environment.

PLISP provides such functions as *send-msg, probe-msg* and *receive-msg.* Messages are sent in a three step process by the *send-msg* function in order to (1) minimize network accesses to two per message and (2) guarantee a receiver will not be able to read a message that has only been partially written. First, the message is written to a local intermediate file.

This file is then copied to another intermediate file local to the receiver (via the *rcp* system command). Lastly, the "atomic" Unix *mv* command is used to rename the intermediate message file to the predetermined incoming message file name in the publicly accessible */tmp* directory. The *probe-msg* function determines if there is an incoming message from a specified machine while the *receive-msg* function simply reads a message from a local file. Note that the *probe-msg* function checks only for local files and thus avoids accessing the network. For a program to send and receive messages with these functions, however, it must first initialize the machine names or "addresses" to which messages may be sent to or received from by calling the *crank-plisp* function.

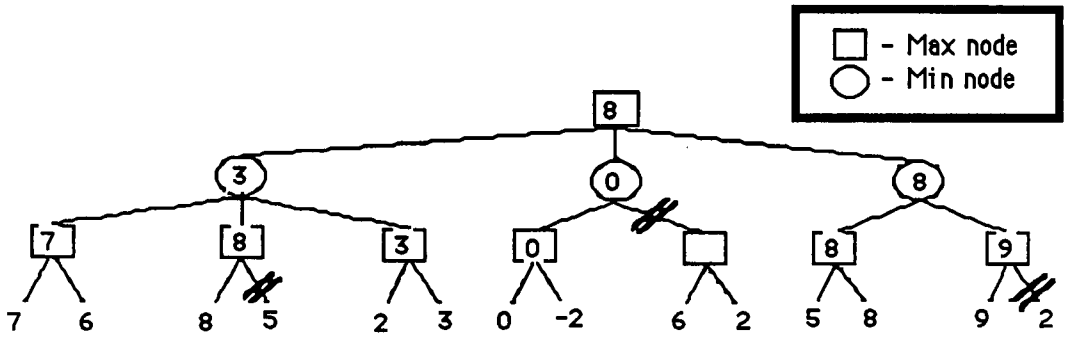
Before discussing the individual experiments, there are some technical features common to all experiments that should be noted. First, all graphs of run time data are presented as $\log_{10}$ curves for clarity. All standard deviations were calculated according to the following formula:

$$\delta = \sqrt{\frac{\Sigma\ (x - \bar{x})^2}{n - 1}}$$

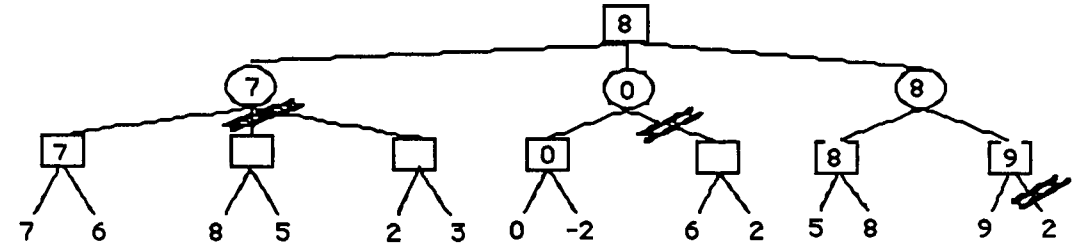
Finally, all results presented are the average of 10 trials. Due to limitations of available computing time, it was feasible only to run this small number of cases for all hand sizes and versions of the algorithms. Consequentially, we acknowledge that there are too few instances to "statistically" rely on our results. With these technical points regarding the presentation of our results in mind, let us examine in more detail the aspiration, principal variation and principal variation splitting algorithms and their performance on bridge specific game trees.

3.2 Aspiration Algorithm

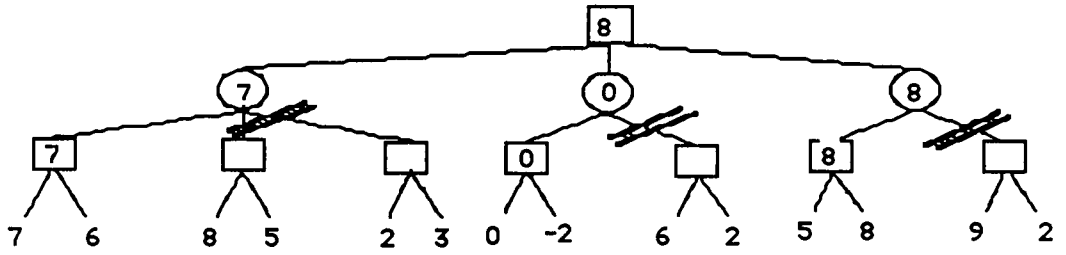
Aspiration search [Marsland 1982] is an enhancement to alpha-beta search which involves narrowing the initial alpha-beta window. Instead of searching an interval ranging from minus infinity to plus infinity, aspiration search considers a limited range within which the minimax value of the root is estimated to lie. There are three possible results given this type of search; either (1) the value returned is the true minimax value, or (2) the search fails high, or (3) the search fails low. The value returned is the true minimax value if it lies strictly between the original estimated alpha and beta bounds. This is true for the same reason that alpha-beta search is guaranteed to return a correct value. If aspiration search returns a value greater than or equal to the original estimated upper bound beta, then the true minimax value is greater than or equal to beta, and it is said to have failed high. Lastly, if the returned value is less than or equal to the initial estimated lower bound alpha, the true minimax value is less than or equal to alpha, and the search fails low. For a comparison of regular alpha-beta search and the three possible outcomes of aspiration search on a small tree refer to Figure 3.1. In this example, alpha-beta search with a window size of minus infinity to plus infinity searches 20 nodes, while aspiration search with a window size bracketing the true result (i.e. 7 to 9) searches only 15 nodes; in this instance aspiration search is a 25 percent improvement over alpha-beta. In the case of aspiration search failing low and failing high, 13 and 8 nodes respectively are searched, but to find the true result, new bounds must be generated and the entire tree researched. In the sequential aspiration algorithm, if the new bounds selected in the case of failing low are minus infinity to alpha and in the case of failing high are beta to plus infinity, then the true result is guaranteed to be returned from this second search. Obviously, the performance of sequential aspiration search depends on the accuracy of the initial alpha and beta estimates.



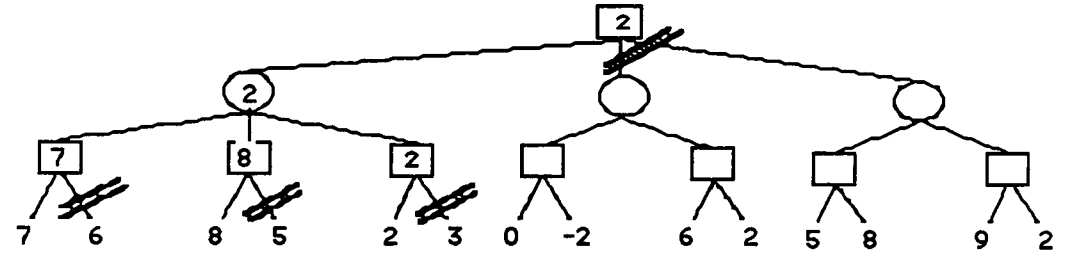
(a) Alpha-Beta search with window size minus infinity to plus infinity



(b) Aspiration search with window size 7 to 9



(c) Aspiration search with window size 10 to 12



(d) Aspiration search with window size 0 to 2

Figure 3.1 Sample Alpha-Beta and Aspiration Searches

Parallel aspiration search uses n processors to simultaneously search a tree with n disjoint alpha-beta windows whose union covers the entire range of possible values for the game. One of these processors will succeed in returning the true minimax value and will do so in no more time that a single processor searching with a window from minus infinity to plus infinity [Marsland 1982].

3.2.1 Experimental Design

We conducted experiments with two versions of parallel aspiration search. In the first version, all alpha-beta windows were of size two, such that all possible backed up values in a particular game were bracketed. For example, given a 1-suit, 4-card game in which the only possible backed up minimax values are 0 and 1, two processors would be used to run searches beginning with windows of -1 to 1 and 0 to 2. The single processor that returns a value strictly between its starting alpha and beta bounds has returned the correct minimax value.

In the second version, each alpha-beta window is of size one. As the result from all single processors must either fail low or high, the correct minimax value is found when two processors return the same value with one processor having failed high and one having failed low. For example, in a 1-suit, 8-card game since the possible minimax values are 0, 1 and 2, two processors, a and b, would begin searching with initial bounds of 0-1 and 1-2 respectively. If the true minimax value of this game is 1, then processor a would fail high returning 1, meaning that the minimax value is at least 1. Processor b would fail low returning 1 and meaning that the minimax value is no greater than 1. Both processor a and b would have to complete their searches before this version of the algorithm could determine that the correct result is indeed 1. The lowest and highest possible minimax values are the two exceptions to the previous rule. If a processor with the initial bounds

of 0 and 1 returned 0 as the minimax value, this would mean that the search could find no way to get a result higher than 0, and the search would technically fail low. But in the game of bridge we know that there is no possible lower result than 0. Given this knowledge, we can say that the true result must be 0. Similarly in the case of a processor with initial bounds of 12 and 13 returning the highest possible result with a standard deck, 13, we know that the maximum possible score is 13, and so we accept the technically failed high result as the true minimax value.

3.2.2 Performance Expectations

Experiments with randomly ordered, uniformly branching trees show that sequential aspiration searches on average give an improvement of 23 percent over alpha-beta search [Marsland 1982]; the range of 15 to 25 percent is estimated as the amount of improvement by Finkel and Fishburn [1982]. With parallel aspiration (regardless of the number of processors available) Baudet has found that a maximum speedup of 5 or 6 is possible, assuming that the terminal values are randomly distributed [Finkel 1982]. The disadvantage, however, of the reported improvements of both sequential and parallel aspiration search is that they will yield no speedup on perfectly ordered trees [Marsland 1985]. On perfectly ordered trees, aspiration search will in fact be slower than alpha-beta because of overhead costs. Parallel aspiration with only one processor will also be slower for the same reason. Thus, the literature suggests that with sequential aspiration search we may expect a 0 to 25 percent improvement over alpha-beta, and with parallel aspiration search we expect a speedup of 0 to 6.

These performance expectations, however, are not directly applicable to our experiments. First, bridge trees do not branch uniformly; the branching factor decrements every four levels. Second, our trees are not randomly ordered; we apply the good ordering techniques

developed by Williams and Mutchler [Williams 1989]. Third, the alpha-beta algorithm we use, both in our searches and as the basis for comparing our results, also includes the strict bucketing pruning technique developed by Williams and Mutchler [Williams 1989]. For these reasons, we do not expect our results to be exactly in sync with those reported in the literature.

3.2.3 Experimental Results

The results of running both versions of parallel aspiration search on a variety of hand sizes are shown in Figure 3.2, Figure 3.3, Figure 3.4 and Figure 3.5. The run time measure reported for the parallel aspiration version 1 is the run time of the processor returning the correct minimax value plus the run time overhead costs incurred by the master processor (the master processor determines the initial alpha-beta windows for all processors and identifies the processor which has returned the correct minimax value.) The run time measure reported for parallel aspiration version 2 is the run time of the slower of the one or two processors which return the bounding information for the true minimax value plus the overhead costs of the master processor.

Also included in these figures are the derived results for sequential aspiration search. Sequential aspiration results are needed in order to determine what speedup the parallel versions have achieved. By speedup we mean the time required for the serial version of an algorithm to complete divided by the time required for the parallel algorithm to complete. The serial versions of the above algorithms were easily derived from the parallel results once we assumed all processors' searches were run on a single machine one after the other until a minimax value was determined. In calculating the sequential results, a binary search approach was taken, and all possible outcomes were assumed to be equally likely. Because of this assumption, our binary search is in no way optimal. (To do an

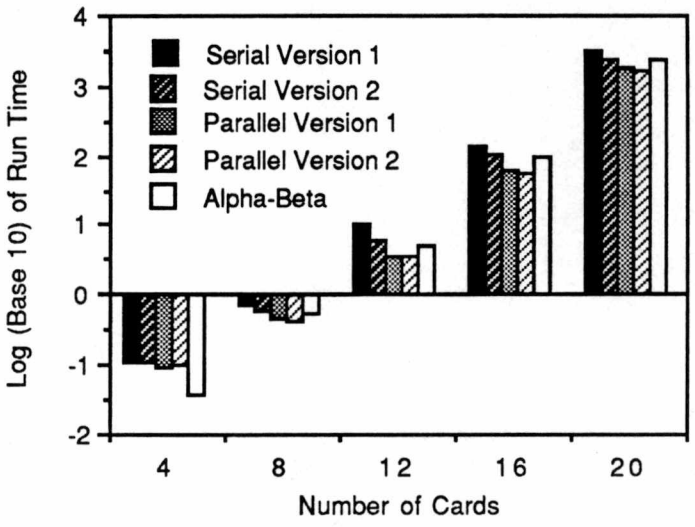


Figure 3.2 Aspiration Search Run Times, 1 Suited Hands

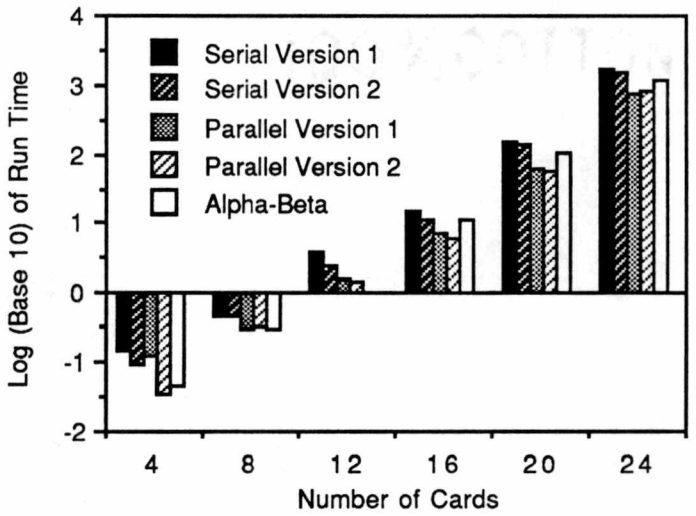


Figure 3.3 Aspiration Search Run Times, 2 Suited Hands

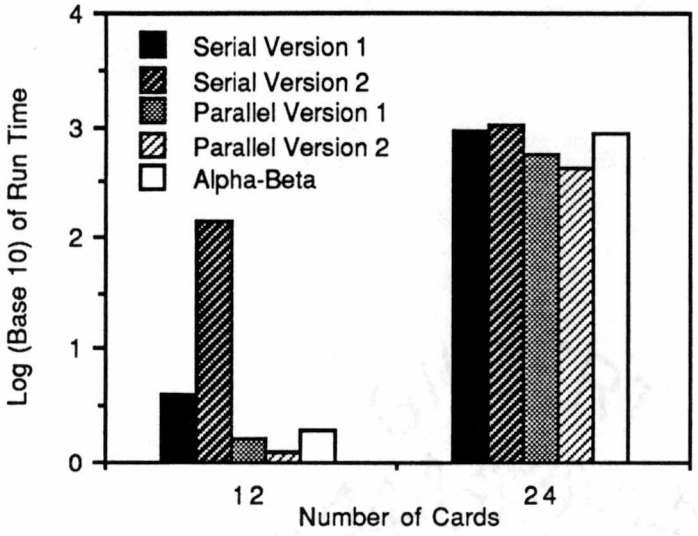


Figure 3.4 Aspiration Search Run Times, 3 Suited Hands

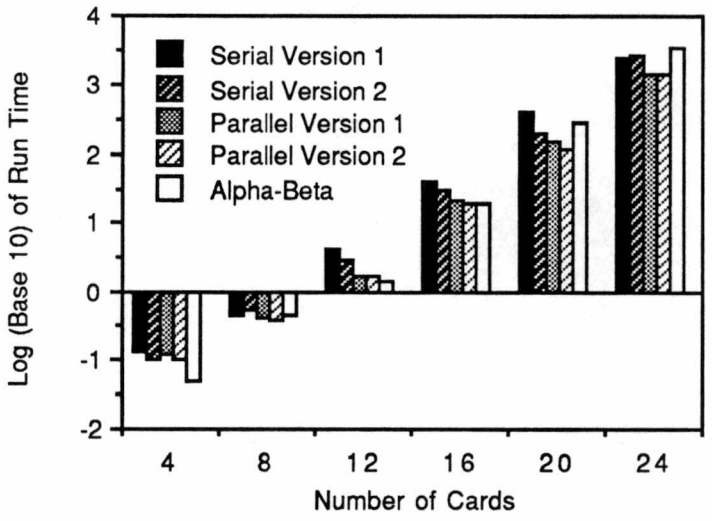


Figure 3.5 Aspiration Search Run Times, 4 Suited Hands

optimal binary search, we would need to do additional experiments to determine the distribution of the outcomes, and time constraints prohibited this.) In these calculations, we arbitrarily rounded values down. In the best case, the minimax value would generally be found after running only one search. In the worst case, the minimax value would be found after doing $\log_2 n$ searches where n is the number of possible search windows. The run time measure reported for both serial aspiration version 1 and version 2 is the actual mean of the trials under the aforementioned assumptions of a binary search technique. Since these values were calculated from the parallel aspiration results, they also erroneously include the overhead costs of a master processor. Since this value is typically under 0.5 seconds, it will cause the run times for the very small hands to be overestimated. For the mid-sized and larger hands, the effect of this one half second will be negligible.

Now, let us consider how our serial aspiration results compare with results reported in the literature. Our results show that serial aspiration version 1 and version 2 are in general slower than alpha-beta search using strict-bucketing and good-ordering (see Figure 3.6). In fact, sequential aspiration version 1 and version 2 are on average approximately 35 and 20 percent slower respectively than our alpha-beta. Speedups for version 1 range from 0.26 to 1.38, while speedups for version 2 range from 0.36 to 1.41. This is significantly less than the 0 to 25 percent improvement predicted in the literature. (In terms of speedup, a 25 percent improvement would be 1.25) It must be that serial aspiration search on bridge trees gives only a small improvement in the best case over our alpha-beta algorithm using strict-bucketing and good-ordering, as we will see.

It appears that sequential aspiration search does better on larger sized hands than on the smaller ones which is to be expected given our inclusion of the parallel overhead costs. Version 1 is approximately 43 percent slower than alpha-beta (values range from 0.32 to

Hand Size		Alpha-Beta with Good-Ordering & Strict Bucketing		Serial Aspiration Version 1			Serial Aspiration Version 2			
Number of Suits	Number of Cards	Run Time (sec.)	Standard Deviation	Run Time (sec.)	Standard Deviation	Speedup Over Alpha-Beta	Run Time (sec.)	Standard Deviation	Speedup Over Alpha-Beta	Speedup Over Version 1
1	4	0	0	0	0	0.34	0	0	0.36	1.08
	8	0	0	1	1	0.73	1	0	0.85	1.17
	12	5	2	10	8	0.50	6	2	0.87	1.75
	16	91	39	139	107	0.66	101	42	0.90	1.37
	20	2343	2127	3049	1807	0.79	2398	2093	0.98	1.27
2	4	0	0	0	0	0.32	0	0	0.47	1.50
	8	0	0	0	0	0.65	0	0	0.66	1.03
	12	1	0	4	3	0.26	2	1	0.43	1.63
	16	11	6	15	10	0.74	12	6	0.95	1.29
	20	101	96	149	71	0.68	139	120	0.73	1.07
	24	1143	872	1692	1065	0.68	1496	1040	0.76	1.13
3	12	2	0	4	3	0.50	2	1	0.80	1.62
	24	867	499	934	678	0.93	1046	742	0.83	0.89
4	4	0	0	0	0	0.37	0	0	0.48	1.31
	8	0	0	0	0	1.01	1	0	0.87	0.87
	12	2	1	4	3	0.49	3	2	0.70	1.43
	16	25	22	38	19	0.66	29	30	0.87	1.32
	20	279	351	379	118	0.74	198	182	1.41	1.91
	24	3286	4131	2385	1734	1.38	2596	2233	1.27	0.92

Figure 3.6 Comparison of Alpha-Beta and Serial Aspiration Search

1.01) on smaller hands with 8 cards or less, while on larger hands with 20 or more cards it is approximately 13 percent slower with values ranging from 0.68 to 1.38. Similarly, version 2 runs on average approximately 38 percent slower than alpha-beta on smaller hands and takes the same amount of time as alpha-beta on larger hands. Speedups range from 0.36 to 0.87 on the smaller hands and from 0.73 to 1.41 on the larger ones.

Comparing version 2 to version 1, we see that version 2 is on average approximately 29 percent faster than version 1. This average improvement is roughly the same even when just small or large hands are compared. Speedups range from 0.87 to 1.91.

Comparing the parallel aspiration results with the serial results (see Figure 3.7), we can see again that speedup is relative to the size, or depth, of the tree. For shallower trees (those of depth 8 or less) the benefits of parallel aspiration are slight, while for deeper trees (those of depth 20 or more) improvements with parallel aspiration are more noticeable. On average, parallel aspiration version 1 resulted in a speedup of 1.28 on the more shallow trees (values range from 1.06 to 1.61) and a speedup of 2.04 with the deeper trees (values range from 1.63 to 2.51). Likewise, parallel aspiration version 2 resulted in an average speedup of 1.20 on the smaller trees and 2.00 on the larger trees. Speedups on the smaller trees ranged from 1.00 to 1.50 and on the larger trees from 1.53 to 2.52. Version 2 performed approximately 11 percent faster overall than version 1, with values ranging from 12 percent slower to 31 percent faster.

In comparison to alpha-beta, parallel aspiration version 1 and version 2 gave an overall average speedup of 1.23 and 1.36 respectively. Speedups range from 0.36 to 2.39 with version 1 and from 0.35 to 2.42 with version 2. Parallel aspiration version 1 resulted in a speedup over alpha-beta of 0.75 on the more shallow trees and of 1.71 on the deeper trees.

Hand Size		Parallel Aspiration Version 1				Parallel Aspiration Version 2				
Number of Suits	Number of Cards	Run Time (sec.)	Standard Deviation	Speedup Over Serial Version 1	Speedup Over Alpha-Beta	Run Time (sec.)	Standard Deviation	Speedup Over Serial Version 2	Speedup Over Parallel Version 1	Speedup Over Alpha-Beta
1	4	0.09	0.03	1.21	0.41	0.11	0.03	1.00	0.82	0.35
	8	0.44	1.43	1.58	1.16	0.40	0.12	1.50	1.10	1.28
	12	3.57	1.43	2.81	1.40	3.44	1.29	1.67	1.04	1.45
	16	60.44	29.31	2.29	1.51	55.49	21.07	1.83	1.09	1.64
	20	1875.48	1953.04	1.63	1.25	1567.31	1620.61	1.53	1.20	1.49
2	4	0.12	0.05	1.14	0.36	0.09	0.02	1.00	1.31	0.48
	8	0.28	0.11	1.61	1.04	0.32	0.09	1.38	0.88	0.92
	12	1.55	0.47	2.45	0.65	1.46	0.45	1.59	1.06	0.68
	16	7.09	4.05	2.10	1.55	6.10	3.49	1.90	1.16	1.80
	20	61.28	50.70	2.42	1.65	55.25	44.12	2.52	1.11	1.83
	24	744.17	581.32	2.27	1.54	796.88	617.99	1.88	0.93	1.43
3	12	1.64	0.66	2.47	1.22	1.27	0.75	1.96	1.29	1.57
	24	554.30	351.17	1.69	1.56	427.89	283.26	2.46	1.30	2.03
4	4	0.12	0.04	1.06	0.39	0.10	0.02	1.00	1.20	0.47
	8	0.41	0.12	1.07	1.09	0.39	0.09	1.30	1.05	1.14
	12	1.73	1.11	2.37	1.16	1.68	1.06	1.70	1.03	1.19
	16	21.07	23.50	1.81	1.19	18.11	18.10	1.59	1.16	1.38
	20	150.75	189.14	2.51	1.85	118.82	141.73	1.67	1.27	2.35
	24	1376.30	993.02	1.73	2.39	1359.12	1048.18	1.91	1.01	2.42

Figure 3.7 Comparison of Serial and Parallel Aspiration Search

Speedups on the smaller trees ranged from 0.36 to 1.16 and on the larger trees from 1.25 to 2.39. Similarly, parallel aspiration version 2 resulted in a speedup over alpha-beta of 0.77 on the smaller trees (with values ranging from 0.35 to 1.28) and of 1.93 on the larger trees (with values ranging from 1.43 to 2.42).

This depth sensitivity in both the sequential and parallel versions of aspiration search may be related to the performance of the alpha-beta algorithm on bridge trees. In the alpha-beta algorithm, there are certain nodes that must be looked at; these are called critical nodes, and together they form the minimal tree that must be examined in order to generate the correct minimax value [Knuth 1975]. Since both the alpha-beta and aspiration algorithms must search this minimal tree, it is only in the search of the non-critical nodes that improvements can be made by aspiration search. At shallow depths, bridge trees contain very few non-critical nodes making the maximum possible speedup very small. In the tree depicted in Figure 3.8, there is no possible speedup as all nodes are critical ones. Both alpha-beta and aspiration search must examine all the nodes. This would certainly explain parallel aspiration search's slower performance on the 4-card games; aspiration has the extra overhead costs and must search the same number of nodes as alpha-beta.

Fortunately, more non-critical nodes are found in deeper trees which increases the maximum possible speedup. The tree in Figure 3.9, for example, has 49 non-critical nodes, and the maximum possible speedup is approximately 2.07 (the total number of nodes divided by the number of critical nodes). This maximum speedup could only be realized, however, in relation to an exhaustive or minimax search of this tree. The standard alpha-beta algorithm might prune some of these non-critical nodes and reduce the maximum possible speedup. The maximum attainable speedup by aspiration search would also be

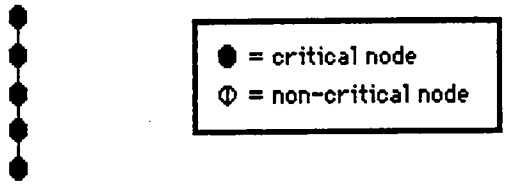


Figure 3.8 1-Suit 4-Card Minimax Tree

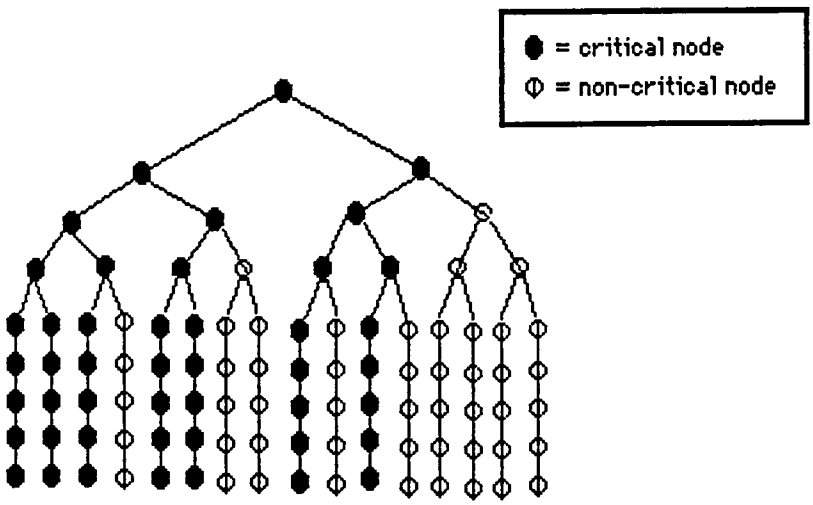


Figure 3.9 1-Suit 8-Card Minimax Tree

reduced by the additional pruning resulting from the good-ordering and strict bucketing techniques incorporated into our version of the alpha-beta algorithm.

3.3 Principal Variation Alpha-Beta Algorithm

Another parallel version of the alpha-beta algorithm is principal variation alpha-beta [Finkel 1982], or PValphabeta. This algorithm is based on the concept of a processor tree in which there are interior and leaf processor nodes. Each interior processor beginning with the root processor evaluates its position by generating the children for a given position and assigning each of these children to be evaluated by one of its slave processors. When all the children of a position have been evaluated or cutoff, an interior processor can compute its value. Each leaf processor evaluates its position by running the serial alpha-beta algorithm on its position. As soon as a processor is able, it reports its value or result to its master. All processor nodes are involved in searching positions in a particular game or lookahead tree. For example, given the processor tree in Figure 3.10 and the lookahead tree in Figure 3.11, the PValphabeta algorithm searches the lookahead tree in the manner shown in Figure 3.12. Notice that the bulk of the work is done during time steps 1, 2 and 3 and that this work is done in parallel.

3.3.1 Experimental Design

We ran experiments using two different processor trees; PValphabeta version 1 used a tree of seven processors (see Figure 3.10), and version 2 utilized seventeen processors as diagrammed in Figure 3.13. In an effort to focus more processors on the area where the alpha-beta algorithm spends most of its time searching we developed a third version of PValphabeta to test. Under optimal ordering conditions (i.e. the best move is always

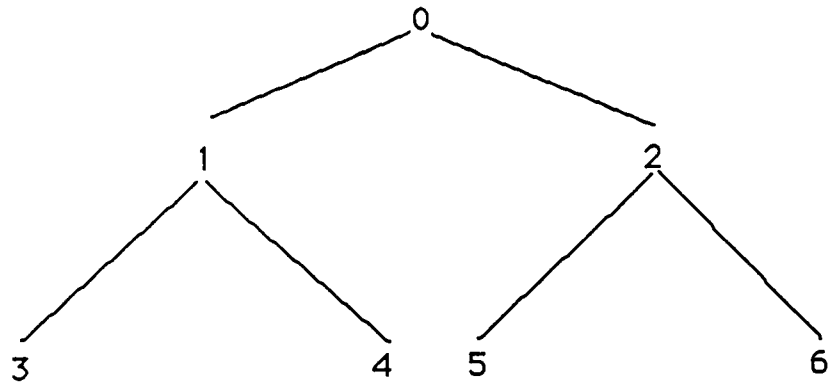


Figure 3.10 Sample Processor Tree

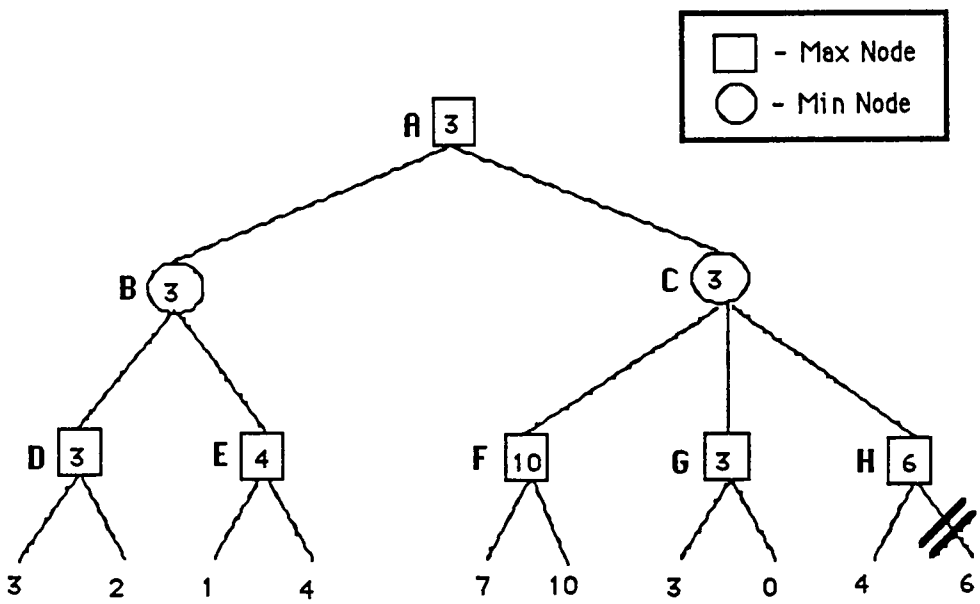


Figure 3.11 Sample Lookahead Tree

Time Steps	Processor 0	Processor 1	Processor 2	Processor 3	Processor 4	Processor 5	Processor 6
t = 0	Evaluate root. Generate B and C. Start Palphabeta on processor 1 with node B. Start Palphabeta on processor 2 with node C.						
t = 1		Evaluate B. Generate D and E. Set processor 3 to do alphabeta on D, and processor 4 to do palphabeta on E.	Evaluate C. Generate F, G and H. Set processor 5 to do alphabeta on F, and processor 6 to do alphabeta on G.				
t = 2				Do serial alphabeta on D. Return 3 to processor 1.	Do serial alphabeta on F. Return 4 to processor 1.	Do serial alphabeta on F. Return 10 to processor 2.	Do serial alphabeta on G. Return 3 to processor 2.
t = 3		Get results from processors 3 and 4. Evaluate node to 3. Return 3 to processor 0.	Get results from processors 5 and 6. Set processor 5 to do alphabeta on H.				
t = 4	Get result from processor 0.					Do serial alphabeta on G. Cutoff 1 of 2 children. Return 4 to processor 2.	
t = 5			Get result from processor 5. Evaluate node to 3. Return 3 to processor 0.				
t = 6	Get result from processor 2. Evaluate root to 3.						

Figure 3.12 Sample Application of PValphabeta Algorithm

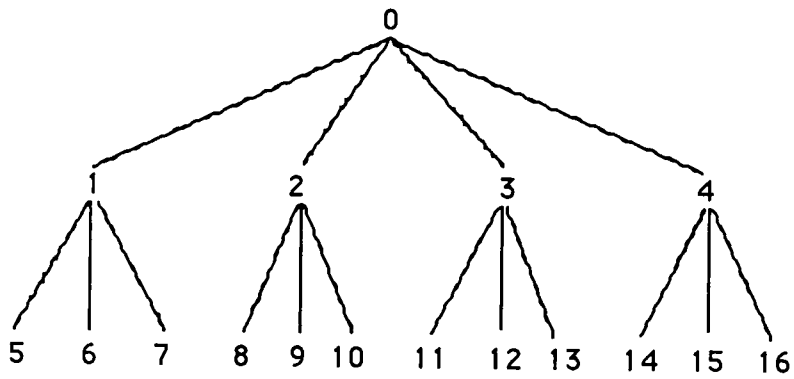


Figure 3.13 PValphabeta Version 2 Processor Tree

examined first), the principal variation is "...the largest part of the minimal tree that must be searched." [Marsland 1982, p. 549]. In the hopes of more rapidly searching the principal variation, PValphabeta version 3 uses a skewed processor tree (see Figure 3.14).

We also added a limiting parameter, *l*, to version 3 that would prevent a master from assigning a slave to evaluate a node in the lookahead tree if that node was within *l* levels of the bottom of the lookahead tree. This was done to try and keep the computation per communication ratio high. We ran version 3 with *l* values of 0, 2, 4, 8, 12 and 16.

3.3.2 Performance Expectations

Finkel and Fishburn report that the performance of PValphabeta is a function of the processor tree fanout, *f*, which they assume is uniform and the processor tree depth, *d*. In the case of a worst-first ordering of the lookahead tree (i.e. there are no alpha-beta cutoffs), speedup is reported to be f^d [Finkel 1982]. In the case of a best-first ordering of the lookahead tree, speedup is estimated to be equal to the square root of f^d [Finkel 1982]. No performance analysis was reported for an average case.

The major disadvantage of PValphabeta is that master processors wait until all their slaves have completed their tasks before assigning new tasks to the slaves. This might not have much effect if all slave tasks required approximately the same amount of processing time. In our case, given the nature of the alpha-beta algorithm, however, it would be almost impossible to ensure slave tasks were of equal length.

Given the theoretical predictions, our version 1 could result in a speedup between 2 and 4. As the fanout in our version 2 is not strictly uniform (it varies between 3 and 4), the

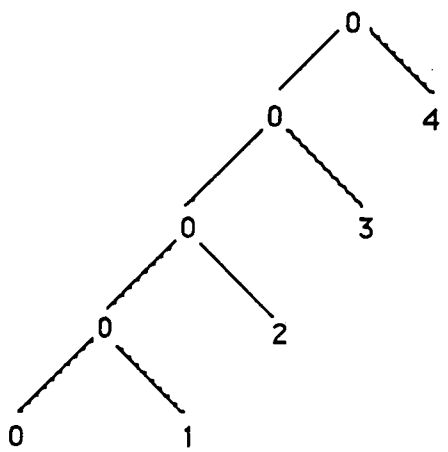


Figure 3.14 PValphabeta Version 3 Processor Tree

theoretical predictions regarding speedup is not defined. If we assume the fanout to be a uniform 3, then the possible speedup range is 3 to 9, and if we assume the fanout to be a uniform 4, then the range is 4 to 16. Since the theoretical mode of performance evaluation is even less applicable to version 3 than to version 2, no prediction will be made as to its performance.

Again, these expectations are not directly applicable to our experiments for the same reasons as mentioned in the discussion of the performance expectations of aspiration search (see p. 52).

3.3.3 Experimental Results

The run time results of our experiments with PValphabeta version 1, version 2 and version 3 on a variety of hand sizes are shown in Figures 3.15, 3.16, 3.17 and 3.18. The run time measure reported for version 1 and 2 is the sum of the run times for the interior processor nodes plus the run time of the slowest leaf processor. Because some of the interior processors run in parallel, our run time results will be somewhat overestimated. This overestimation will be slight as the average total run time of all interior processors is only 0.6 seconds. Because this average is the same for both version 1 that has only three interior processors and version 2 that has five interior processors it seems reasonable to assume that much of this time is incurred by the master processor 0 as initial overhead and should in fact be included in the reported run time. The run time measure reported for version 3 is the run time of the slowest processor plus a small "start up" cost initially incurred by processor 0.

Figure 3.19 shows how these results compare with alpha-beta and gives the actual speedup achieved by all versions of the PValphabeta algorithm. Again, we can see that speedup is

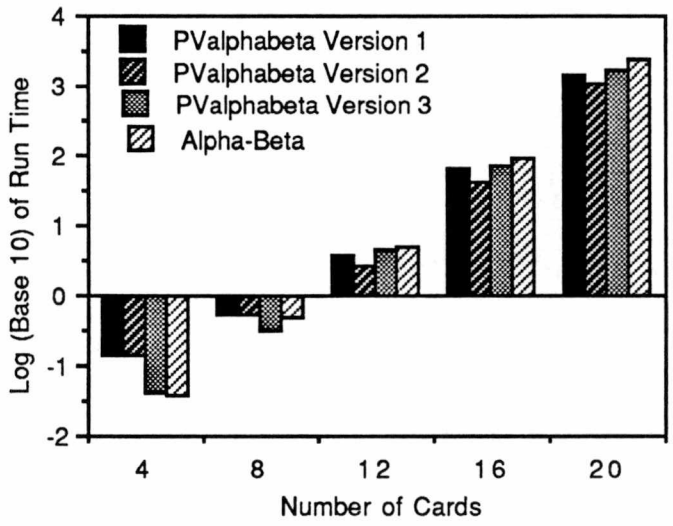


Figure 3.15 PValphabeta Search Run Times, 1 Suited Hands

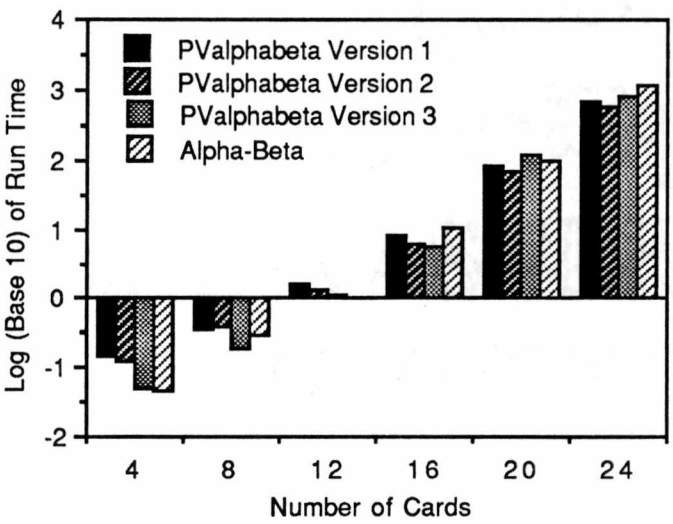


Figure 3.16 PValphabeta Search Run Times, 2 Suited Hands

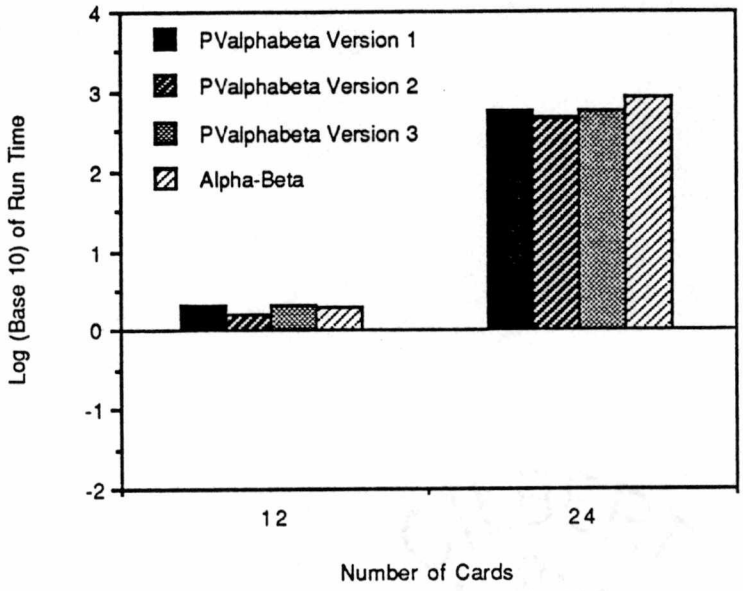


Figure 3.17 PValphabeta Search Run Times, 3 Suited Hands

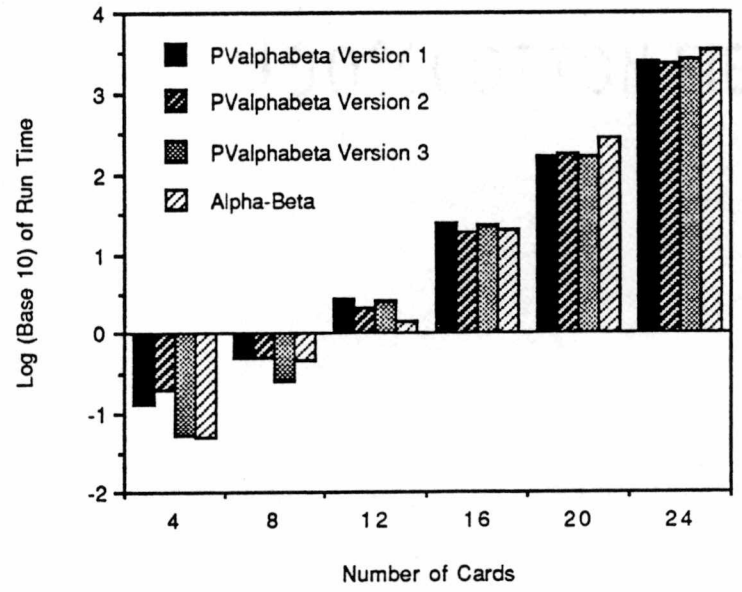


Figure 3.18 PValphabeta Search Run Times, 4 Suited Hands

Hand Size		PValphabeta Version 1			PValphabeta Version 2				PValphabeta Version 3				
Number of Suits	Number of Cards	Run Time (sec.)	Standard Deviation	Speedup Over Alpha-Beta	Run Time (sec.)	Standard Deviation	Speedup Over Alpha-Beta	Speedup Over Version 1	Run Time (sec.)	Standard Deviation	Speedup Over Alpha-Beta	Speedup Over Version 1	Speedup Over Version 2
1	4	0.15	0.04	0.25	0.14	0.02	0.27	1.07	0.04	0.01	0.95	0.38	3.50
	8	0.55	0.13	0.93	0.54	0.12	0.95	1.01	0.32	0.12	1.60	1.72	1.69
	12	3.86	2.31	1.25	2.72	0.61	1.84	1.42	4.52	2.30	1.11	0.85	0.60
	16	62.45	20.27	1.47	40.28	11.07	2.26	1.55	72.48	27.91	1.26	0.86	0.56
	20	1470.98	923.43	1.59	1069.72	573.72	2.19	1.38	1722.44	941.17	1.36	0.85	0.62
2	4	0.15	0.04	0.29	0.12	0.04	0.36	1.25	0.05	0.01	0.86	3.00	2.40
	8	0.35	0.17	0.84	0.38	0.15	0.77	0.92	0.18	0.14	1.63	1.94	2.11
	12	1.53	0.42	0.65	1.26	0.23	0.80	1.22	1.08	0.53	0.96	1.41	1.16
	16	8.36	3.56	1.32	6.16	2.81	1.79	1.36	5.79	3.66	1.90	1.44	1.06
	20	86.36	56.95	1.17	70.98	50.10	1.42	1.22	117.80	105.29	0.86	0.73	0.60
	24	728.29	440.08	1.57	557.55	289.51	2.05	1.31	878.70	478.70	1.30	0.83	0.63
3	12	2.05	0.57	1.00	1.62	0.34	1.23	1.27	2.10	1.09	0.97	0.97	0.77
	24	572.62	302.23	1.51	479.91	267.07	1.81	1.19	574.86	325.03	1.51	1.00	0.83
4	4	0.13	0.02	0.36	0.19	0.16	0.25	0.68	0.05	0.01	0.93	3.20	3.80
	8	0.47	0.04	0.95	0.49	0.08	0.91	0.96	0.25	0.04	1.78	1.86	1.96
	12	2.63	1.26	0.76	2.12	0.65	0.94	1.24	2.54	1.56	0.79	1.06	0.86
	16	24.44	24.52	1.02	18.69	15.04	1.34	1.31	22.05	17.51	1.13	1.11	0.85
	20	162.03	183.93	1.72	179.17	206.73	1.56	0.90	160.60	165.17	1.74	1.01	1.12
	24	2415.22	2958.80	1.36	2290.87	3197.02	1.43	1.05	2598.22	3550.87	1.26	0.93	0.88

Figure 3.19 Comparison of Alpha-Beta and PValphabeta Search

relative to the size, or depth, of the tree. For shallower trees (those of depth 8 or less) there are seemingly no benefits to PValphabeta, while for deeper trees (those of depth 20 or more) improvements are noticeable. On average, PValphabeta version 1 resulted in a speedup of 0.61 on the more shallow trees and a speedup of 1.49 with the deeper trees. Speedups on the more shallow trees ranged from 0.35 to 0.95 and from 1.17 to 1.72 on the deeper trees. Likewise, PValphabeta version 2 resulted in an average speedup of 0.59 on the smaller trees with values ranging from 0.25 to 0.95 and an average speedup of 1.74 on the larger trees with values ranging from 1.42 to 2.19. Overall, version 1 and version 2 were 5 and 27 percent faster respectively on average than alpha-beta. Speedups range from 0.25 to 1.72 with version 1 and from 0.25 to 2.26 with version 2.

The limiting parameter, *l*, in PValphabeta version 3 did not seem to significantly affect the results. The effect of *l* was most noticeable with the largest trees. However, the difference in run times of games searched with *l=0* and *l=16* was very small -- approximately 2 percent. For this reason, only the results of version 3 with *l=16* are reported as a basis for comparison. On average, PValphabeta version 3 resulted in a speedup of 1.26 over alpha-beta with values ranging from 0.79 to 1.90. On the more shallow trees of depth 8, PValphabeta version 3 realized an average speedup of 1.67, and on trees of depth 20 or more realized an average speedup of 1.34. Speedups on these more shallow trees ranged from 1.60 to 1.78 and from 0.86 to 1.74 on the deeper trees. On the mid-size trees of depth 12 and 16, an average speedup of 1.16 was realized with values ranging from 0.79 to 1.90.

Of the three versions of PValphabeta, version 3 was consistently better on the smaller trees of depth 8 and less, averaging 2.02 and 2.58 percent better than versions 1 and 2 respectively. Conversely, version 3 performed least well on the larger trees of depth 20

and more, taking on average 11 and 22 percent longer to run than versions 1 and 2 respectively. Version 2 performed better than version 1, averaging an overall speedup of 1.17 over version 1 with values ranging from 0.68 to 1.55. Whether this average improvement of 17 percent is worth the "expense" of using ten more processors is questionable.

3.4 Principal Variation Splitting Algorithm

Principal variation splitting [Marsland 1982], or PVSalphabeta, is the third and final algorithm we tested. PVSalphabeta is similar to PValphabeta except the principal variation is searched by all processors before the tree is decomposed for further examination. PVSalphabeta allows the first variation to be searched faster and "...ensures that all the processors at a given level start with a good window bound. This provides more cutoffs and allows the balance of the search to proceed more quickly." [Marsland 1985, p. 444]

Like PValphabeta, this algorithm is based on the concept of a processor tree, but in PVSalphabeta a master processor is never idle because it is always one of its own slaves. The root, or master, processor follows the principal variation generating nodes in the lookahead tree for *length* levels where *length* is an initial parameter value. In general the principal variation is the path from the root to an optimal terminal node, but in this case the principal variation is assumed to the be leftmost branch. (In reality, the principal variation would be the path formed by the leftmost branches only if all nodes were in an optimal order.) After traversing *length* levels, the master processor assigns each of the slave processors to evaluate the nodes that correspond to their position in the processor tree. When these nodes have been evaluated, any unexamined children at the level of the master

processor are again divided among the slave processors. When there are no more nodes to be examined at that level, the master processor backs up one level in the lookahead tree and again divides the nodes among the slave processors. This continues until the entire lookahead tree has been searched. (Pseudo-code for this algorithm is given in Figure 3.20.) For example, given the processor tree in Figure 3.21, the lookahead tree in Figure 3.22, and *length=1*, Figure 3.23 shows how the PVSalphabeta algorithm would generate the correct minimax value.

3.4.1 Experimental Design

All experiments with this algorithm used four processors organized into the tree format shown in Figure 3.20. Although we used the same processor tree in all experiments, we tested the performance of this algorithm with various *length* parameters. Trials were run with *length* values of 0, 2, 4, 8, 12 and 16.

3.4.2 Performance Expectations

Marsland and Campbell [1982] reported experimental results using PVSalphabeta (*l=1*) to search both "optimally ordered" and "strongly ordered" uniform lookahead trees of depth 4 and width 24. Optimally ordered trees are those in which the best move is always selected first for expansion. Trees were defined as strongly ordered if "...70% of the time the first branch from each node is best, [and]...90% of the time the best move is in the first quarter of the branches being searched." [Marsland 1982, p. 536] For processor trees with depth 2 and fanout 2, a speedup of 1.40 with optimally ordered trees and 1.09 with strongly ordered trees was reported over PValphabeta [Marsland 1982]. Given that our PValphabeta algorithm using a processor tree of depth 2 and fanout 2 resulted in a speedup of 1.05 over alpha-beta, this translates into a 1.47 and 1.14 speedup respectively by PVSalphabeta over alpha-beta. Marsland and Campbell's [1982] results with various

```
PVSalphabeta(position,alpha,beta,length) {
        if (length <= 0) return(treesplit(position,alpha,beta));
        children = generate(position);
        x = PVSalphabeta(children[1],alpha,beta,length-1);
        if (this is a max node && x > alpha) {
                alpha = x;
                if (alpha >= beta) return(alpha);
        } else if (this is a min node && x < beta) {
                beta = x;
                if (alpha >= beta) return(beta);
        } for all slaves {
                when (a slave is idle) {
                        value = treesplit(position,alpha,beta);
                        if (this is a max node && value > alpha) alpha = value;
                        else if (this is a min node && value < beta) beta = value;
                        if (alpha >= beta) {
                                if (this is a max node) return(alpha);
                                else if (this is a min node) return(beta);
                        }
                }
        }
        if (alpha >= beta) {
                if (this is a max node) return(alpha);
                else if (this is a min node) return(beta);
        }
}


treesplit(position,alpha,beta) {
        if (I am a leaf processor) return(alpha-beta(position,alpha,beta));
        children = generate(position);
        for all slaves {
                when (a slave is idle) {
                        value = treesplit(position,alpha,beta);
                        if (this is a max node && value > alpha) alpha = value;
                        else if (this is a min node && value < beta) beta = value;
                        if (alpha >= beta) {
                                if (this is a max node) return(alpha);
                                else if (this is a min node) return(beta);
                        }
                }
        }
        if (alpha >= beta) {
                if (this is a max node) return(alpha);
                else if (this is a min node) return(beta);
        }
}
```

Figure 3.20 PVSalphabeta Algorithm

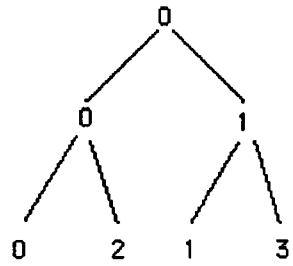


Figure 3.21 Sample Processor Tree

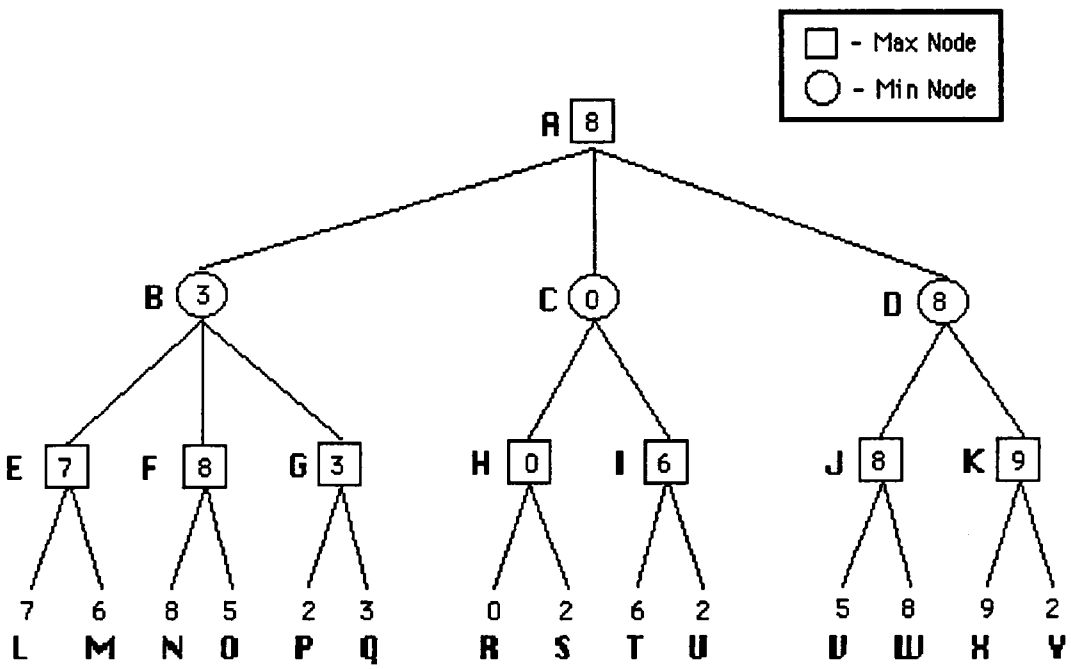


Figure 3.22 Sample Lookahead Tree

Time Steps	Processor 0	Processor 1	Processor 2	Processor 3
t = 0	Traverse the principal variation, expanding nodes A and B. Current depth in lookahead tree equals *length*.			
t = 1	All slaves assigned to evaluate nodes the "cover." Evaluate nodes E and L.	Evaluate nodes F and N.	Evaluate node M.	Evaluate node 0.
t = 2	One node remains to be evaluated at this level. Evaluate nodes G and P.		Evaluate node Q.	
t = 3	All nodes evaluated at this level. Back up one level and re-assign slaves to evaluate nodes they cover. Evaluate nodes C, H, R and S.	Evaluate nodes D, J, V and W.	Evaluate nodes I, T and U.	Evaluate nodes K, X and Y.

Figure 3.23 Sample Application of PVSalphabeta Algorithm

processor tree configurations showed an average speedup of 1.82 with optimally ordered trees and 1.35 with strongly ordered trees over PValphabeta. As compared with alpha-beta, PVSalphabeta would give a speedup of 1.91 and 1.42 respectively.

To determine how our good-ordering technique compares with Marsland and Campbell's definition of strong and optimal ordering, we ran a special minimax algorithm on various hand sizes, testing the ordering of 100 cases of each hand size. The results of these trials are presented in Figure 3.24. Overall, the first branch at a node turns out to be best 98 percent of the time. Since the 98 percent figure includes cases in which the branching factor is one, let us also look at the worst case we examined where the branching factor is four. In this case, the first branch at a node turns out to be best 83 percent of the time. Given the "best" and "worst" case figures of 98 and 83 percent respectively, our good-ordering certainly meets the first of Marsland and Campbell's requirements for strong ordering. The second requirement, however, is geared towards trees with a very wide branching factor. For trees with branching factors less than 8 to have the best move in the first quarter of the branches searched, the best move must be the first branch, and the first move being best is what is covered by requirement one. In short, we believe the second requirement is not applicable to our narrow bridge trees, and we will assume that our trees are strongly ordered by virtue of fulfilling Marsland and Campbell's first requirement. That being the case, we would expect our version of PVSalphabeta to be slightly better than PValphabeta version 1.

3.4.3 Experimental Results

The results of all our experiments with PVSalphabeta on a variety of hand sizes are show in Figures 3.25, 3.26, 3.27 and 3.28. The run time measure reported is the run time of the slowest processor which most often is the master, processor 0. Inherent in this measure,

		Branching Factors									
Hand Size		*b=1*	*b=2*		*b=3*			*b=4*			
Number of Suits	Number of Cards	% 1st	% 1st	% 2nd	% 1st	% 2nd	% 3rd	% 1st	% 2nd	% 3rd	% 4th
1	4	100									
	8	100	94	5							
	12	100	95	4							
2	4	100									
	8	100	93	6							
	12	100	92	7	80	18	1				
	16	100	93	6	87	7	4	83	11	5	0
3	12	100	91	8	84	10	5				
4	4	100									
	8	100	91	8							
	12	100	91	8	84	9	5				
Column Average		100	93	7	84	11	4	83	11	5	0

Figure 3.24 Percentage of Best Children by Position

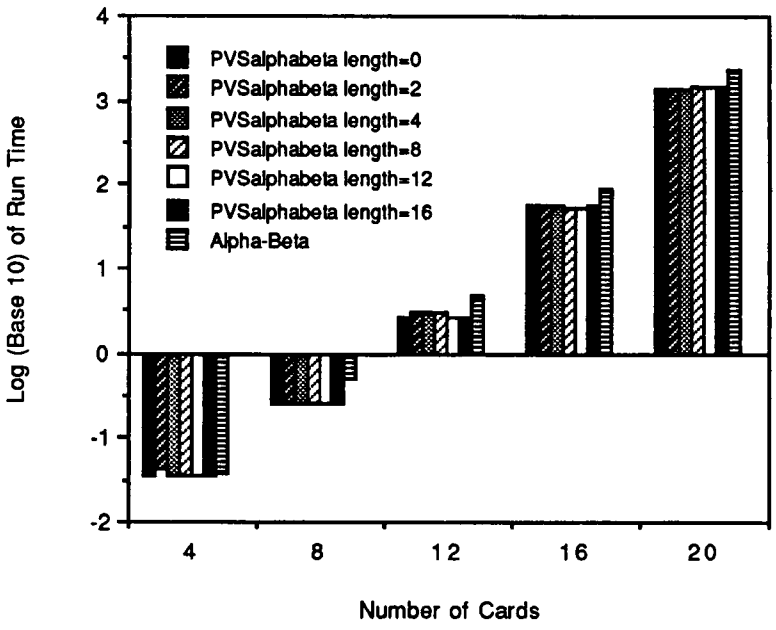


Figure 3.25 PVSalphabeta Search Run Times, 1 Suited Hands

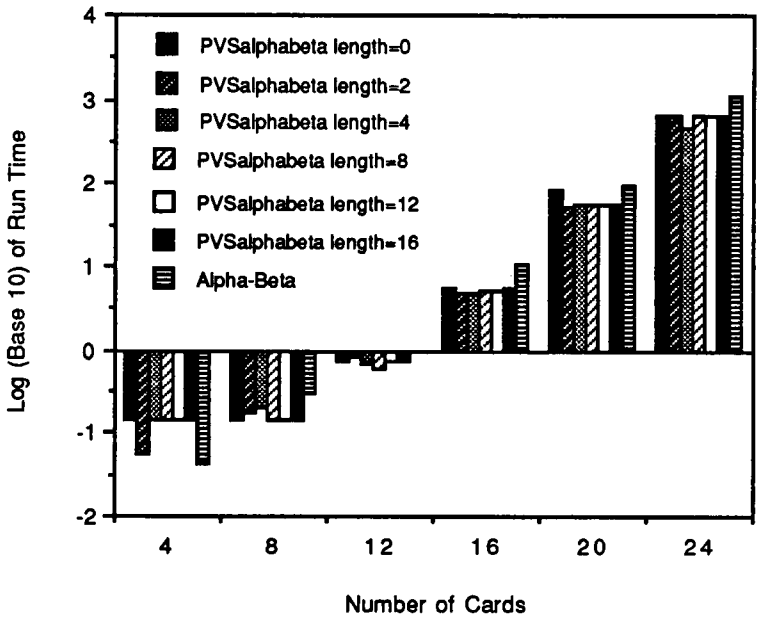


Figure 3.26 PVSalphabeta Search Run Times, 2 Suited Hands

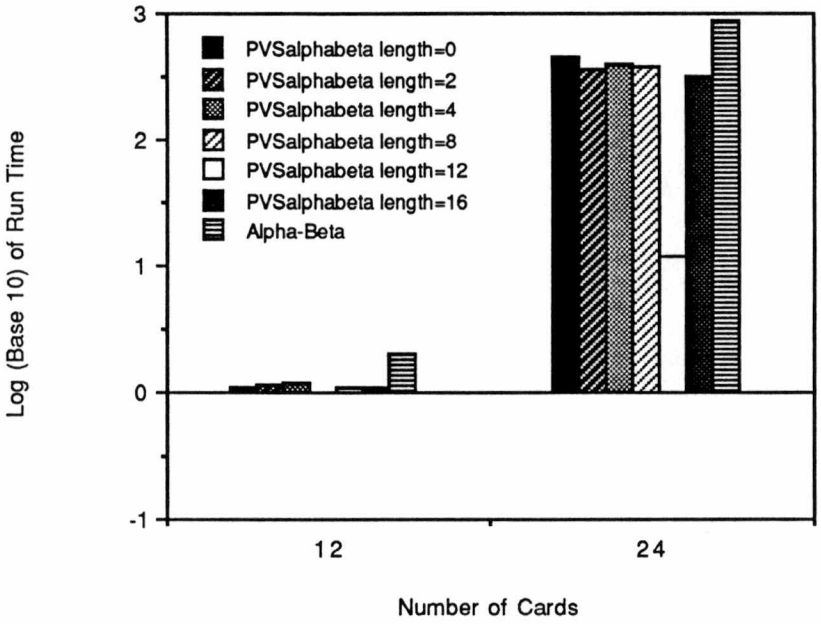


Figure 3.27 PVSalphabeta Search Run Times, 3 Suited Hands

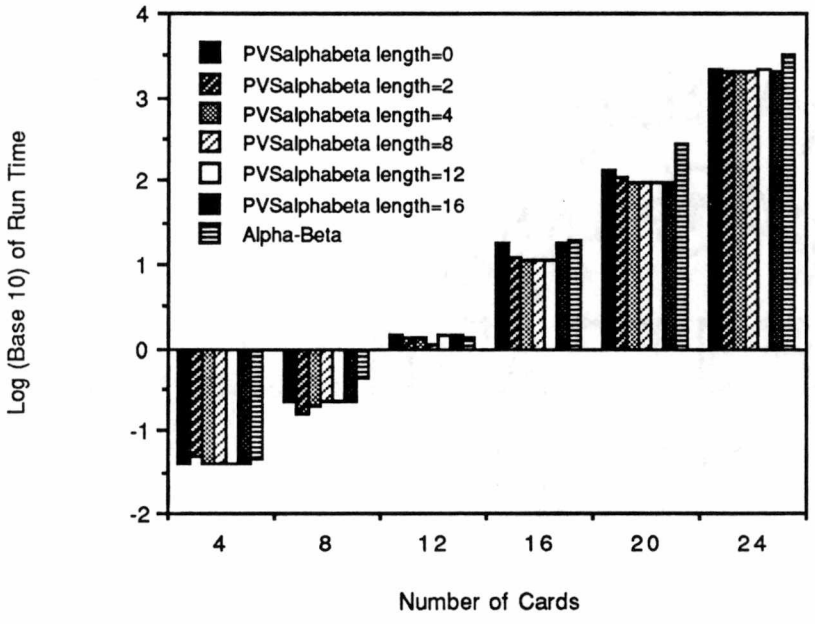


Figure 3.28 PVSalphabeta Search Run Times, 4 Suited Hands

however, is the assumption that the overall slowest processor never has to wait for another processor to finish its work before continuing.

Figure 3.29 shows how these results compare with alpha-beta and gives the actual speedup achieved by all versions of the PVSalphabeta algorithm. PVSalphabeta with all the *length* values performed well. The two tests with *length=2* and *length=8* did best resulting in respective average speedups of 1.72 and 1.73 over alpha-beta. Speedups range from 0.75 to 2.49 with *length=2* and from 0.29 to 2.85 with *length=8*. Overall the version with *length=0* did the least well, giving an average speedup of 1.54 over alpha-beta with values ranging from 0.29 to 2.09. On the smaller trees of depth 8 or less, the version with *length=2* gave the best average speedup of 1.51 while the version with *length=4* gave the worst average speedup of 1.31. Speedups on the smaller trees with *length=2* range from 0.75 to 2.78 and from 0.29 to 2.15 with *length=4*. On the larger hands with 20 or more cards, the versions using *length=4* and *length=16* performed best, giving an average speedup of 2.04 and 2.02 respectively. Speedups on these larger trees range from 1.52 to 2.79 with *length=4* and from 1.50 to 2.79 with *length=16*. Again, the version with *length=0* performed least well on the larger hands and gave an average speedup of 1.66 over alpha-beta with values ranging from 1.18 to 2.06.

3.5 Summary

Figure 3.30 shows how the best version of each of the three algorithms tested -- aspiration, PValphabeta and PVSalphabeta -- performed as compared with the alpha-beta algorithm that included the good-ordering and strict bucketing techniques. Although each of these versions of the algorithms resulted in an improvement over alpha-beta, overall they did not

Hand Size		PVSalphabeta								
		length = 0			length = 2			length = 4		
Number of Suits	Number of Cards	Run Time (sec.)	Standard Deviation	Speedup Over Alpha-Beta	Run Time (sec.)	Standard Deviation	Speedup Over Alpha-Beta	Run Time (sec.)	Standard Deviation	Speedup Over Alpha-Beta
1	4	0.04	0.02	0.95	0.04	0.01	0.95	0.04	0.02	0.95
	8	0.27	0.07	1.90	0.25	0.12	2.01	0.26	0.13	1.97
	12	2.76	1.17	1.81	3.19	1.26	1.57	3.09	1.30	1.62
	16	59.52	22.17	1.53	59.98	21.21	1.52	55.69	22.72	1.63
	20	1385.10	662.09	1.69	1386.67	1061.99	1.69	1410.90	10.68.14	1.66
2	4	0.15	0.07	0.29	0.06	0.01	0.75	0.15	0.07	0.29
	8	0.14	0.07	2.09	0.18	0.12	1.63	0.21	0.11	1.41
	12	0.78	0.27	1.28	0.84	0.30	1.19	0.71	0.35	1.41
	16	5.69	3.55	1.93	5.06	2.45	2.17	5.04	3.08	2.18
	20	85.85	77.57	1.18	53.25	38.52	1.90	57.56	40.08	1.75
	24	685.21	438.01	1.67	667.26	522.43	1.71	488.52	393.21	2.34
3	12	1.09	0.32	1.83	1.13	0.48	1.77	1.16	0.51	1.72
	24	456.87	236.98	1.90	352.90	177.91	2.46	396.37	201.04	2.19
4	4	0.04	0.01	1.11	0.05	0.02	0.93	0.04	0.01	1.11
	8	0.23	0.04	1.93	0.16	0.06	2.78	0.21	0.15	2.15
	12	1.47	0.64	1.36	1.37	0.96	1.46	1.33	0.83	1.50
	16	18.63	14.81	1.34	12.25	18.19	2.04	11.76	15.37	2.13
	20	135.45	151.73	2.06	112.22	147.67	2.49	99.83	158.25	2.79
	24	2250.33	3496.76	1.46	2035.71	3097.37	1.61	2158.06	3240.94	1.52

Figure 3.29 Comparison of Alpha-Beta and PVSalphabeta Search

Hand Size		PVSalphabeta								
		length = 8			length = 12			length = 16		
Number of Suits	Number of Cards	Run Time (sec.)	Standard Deviation	Speedup Over Alpha-Beta	Run Time (sec.)	Standard Deviation	Speedup Over Alpha-Beta	Run Time (sec.)	Standard Deviation	Speedup Over Alpha-Beta
1	4	0.04	0.02	0.95	0.04	0.02	0.95	0.04	0.02	0.95
	8	0.27	0.06	1.90	0.27	0.06	1.90	0.27	0.06	1.90
	12	3.12	1.26	1.60	2.76	1.17	1.81	2.76	1.17	1.81
	16	55.14	21.16	1.65	54.95	21.58	1.66	59.52	22.17	1.53
	20	1441.19	1073.92	1.63	1443.13	1072.28	1.62	1447.88	1072.33	1.62
2	4	0.15	0.07	0.29	0.15	0.07	0.29	0.15	0.07	0.29
	8	0.14	0.07	2.09	0.14	0.07	2.09	0.14	0.07	2.09
	12	0.62	0.37	1.61	0.78	0.27	1.28	0.78	0.27	1.28
	16	5.10	3.50	2.16	5.27	3.58	2.09	5.69	3.55	1.93
	20	55.84	37.88	1.81	56.33	37.98	1.79	56.21	37.95	1.80
	24	679.10	512.68	1.68	686.10	522.30	1.67	690.22	522.60	1.66
3	12	1.00	0.47	2.00	1.09	0.32	1.83	1.09	0.32	1.83
	24	374.20	199.22	2.32	379.47	202.84	2.28	316.46	208.18	2.74
4	4	0.04	0.01	1.11	0.04	0.01	1.11	0.04	0.01	1.11
	8	0.23	0.04	1.93	0.23	0.04	1.93	0.23	0.04	1.93
	12	1.18	0.79	1.69	1.47	0.64	1.36	1.47	0.64	1.36
	16	11.94	15.92	2.09	11.71	15.65	2.13	18.63	14.81	1.34
	20	97.90	104.68	2.85	97.61	104.29	2.86	100.01	104.39	2.79
	24	2152.21	3178.86	1.53	2186.41	3251.32	1.50	2185.17	3244.74	1.50

Figure 3.29 (Continued)

Algorithm	Number of Processors	Theoretical Speedup Expected	Experimental Speedup Achieved Over Alpha-Beta					
			Over All Trees		Over Small Trees (depth <= 8)		Over Large Trees (depth >= 20)	
			Mean	Range	Mean	Range	Mean	Range
Parallel Aspiration Version 2	1 to 7	0.0 to 6.0	1.36	0.35 to 2.42	0.77	0.35 to 1.28	1.93	1.43 to 2.42
PValphabeta Version 2	17	3.0 to 16.0	1.27	0.25 to 2.26	0.59	0.25 to 0.95	1.74	0.90 to 1.38
PVSalphabeta length=2	4	1.42	1.72	0.75 to 2.78	1.51	0.75 to 2.78	1.98	1.61 to 2.49

Figure 3.30 Comparison of Aspiration, PValphabeta and PVSalphabeta Searches

perform as well as the literature predicted. Parallel aspiration performed within the expected range giving an average speedup over alpha-beta of 1.36, but it was very definitely in the low part of the predicted range given for chess trees (0.0 to 6.0). PValphabeta performed way below the predicted range of 3.0 to 16.0 even with 17 processors at its disposal; PValphabeta averaged a 1.27 speedup on bridge trees. Only the PVSalphabeta algorithm performed better than the literature predicted, and we believe this may not be representative for two reasons. This may be due to the fact that the expected performance was reported in terms of improvement over PValphabeta for which we substituted our actual results using PValphabeta rather than the theoretical performance. In addition, the run time measure we reported for PVSalphabeta may underestimate the CPU time actually needed to complete a search with this algorithm. Our measure assumes that the overall slowest processor is always the slowest processor on each of the mini-tasks assigned to it, and as such the measure would not have taken into account any waiting on other processors that was done.

In sum, we believe the results of these experiments show that the reported benefits of these three parallel alpha-beta search algorithms that were developed for and tested on chess-type trees are not directly applicable to bridge trees. Since only relatively small factors of improvement seem possible with the parallel alpha-beta algorithms we tested, other techniques to improve the speed of our machine player will have to be found.

CHAPTER 4

CONCLUSIONS

As one of a series of Master's theses that will hopefully lead to the development of a world class contract bridge playing program, the two contributions of my work are:

- the design and implementation of the graphics interface CBRIDGE and
- an introductory exploration of the potential benefits of three parallel alpha-beta search algorithms.

CBRIDGE supports both human and machine players and provides a variety of instructive, useful and fun features. Numerous parameter options define the type of game to be played. For example, display options allow the user to play double-dummy bridge where all hands are visible or standard contract bridge in which only one hand is always visible to the player. There are event features to allow cheating or peeking at other players' hands. In addition, the auction may be bid interactively or read from a file, and hands may be randomly generated, selected from a file of particularly interesting hands, or specified by the user. During the play of the game, mechanisms exist for generating default "moves" and for backing up. HELP is available throughout the game, and human players are almost always given a recommended action to correct any inappropriate response made. In its current state, CBRIDGE is a fun and user-friendly interface for human bridge players and

a useful one for developing and testing machine players. In addition, CBRIDGE has been designed to facilitate many enhancements which may be implemented at a later time to provide an even better environment for human and machine interaction.

In our experimental study with three parallel alpha-beta search algorithms we sought to answer the question "Do the reported benefits of parallel alpha-beta algorithms derived from a consideration of chess type trees apply more, less or equally to bridge trees?" Our results showed that our best performing versions of parallel aspiration search, principal variation search and principal variation splitting search gave average speedups of 1.36, 1.27 and 1.72 respectively over alpha-beta. The predicted performance or speedup of these algorithms on chess type trees as reported in the literature was 0.0 to 6.0 for parallel aspiration, 3.0 to 16.0 for principal variation search and 1.42 for principal variation splitting search. Comparing the predicted and actual performances we see that principal variation search performed much less well on our bridge trees than on the chess trees. Parallel aspiration search on bridge trees performed at the very low end of the predicted range given for chess trees. Principal variation splitting performed better than predicted by approximately 30 percent. We believe, however, this surprising improvement may be due to artificial factors. First, the predicted speedup was in terms of improvement over the principal variation algorithm, and when we calculated the predicted speedup over alpha-beta, we used our real results with principal variation search as opposed to the theoretical speedup of principal variation over alpha-beta. Secondly, our run time measure for principal variation splitting search may underestimate the actual run time of the algorithm, because our measure assumes that the overall slowest processor is always the slowest and that it never finishes any of the mini-tasks "early" and waits for other processors to finish. Given these two considerations, principal variation splitting may not in fact perform better than expected; more study would be needed to positively determine this. Overall, however,

given the aspiration and the principal variation search results and the uncertain principal variation splitting results, we believe it is fair to say that the reported speedups of these three parallel alpha-beta algorithms derived from chess trees will not in general result in comparable maximum speedups on our bridge trees given our serial alpha-beta algorithm that includes good-ordering and strict bucketing techniques.

As the results of our experiments with three parallel alpha-beta algorithms -- aspiration, principal variation and principal variation splitting -- were rather disappointing, it would probably not be worth the programmer's time to integrate one of these searches into the current double-dummy player. First, the communication costs of our current parallel environment would more than consume any time savings created by the parallel algorithm. Second, the double-dummy player needs exponential improvements in time; at this stage of development a small, constant factor of speedup would not greatly affect the player's performance. If, however, we had access to a parallel environment with more realistic communication costs and our machine player was nearly "up to speed", then it might be worthwhile to implement a parallel search algorithm. Whether we would choose to implement one of the three algorithms tested in this work or not is questionable. One of the algorithms tested might suffice, but with more study other algorithms and methods might be developed to better utilize the potential power that is parallelism. Some ideas perhaps meriting further exploration are discussed in the next chapter.

CHAPTER 5

FUTURE WORK

In considering the overall goal of developing the world's best contract bridge playing program, there are two major goals to accomplish. First, since we have seen that existing parallel alpha-beta algorithms will not result in a tremendous speedup, other techniques must be developed to speed the double-dummy player up sufficiently so that it can play a full 52-card game in a reasonable amount of time. Then, a methodology for handling hidden information will have to be developed. Since it is difficult to envision exactly what will be necessary to achieve these goals, let us consider some suggestions for future work stemming directly from this thesis.

Many proposals for the enhancement of the graphics interface, CBRIDGE, have already been discussed in Chapter 2, but some of these additional ideas are listed below:

- improved user-friendly file formats for entering fixed auctions and fixed hands,
- allowing variable sized decks,
- improved rules for the generation of default bids and card selections which might be used to engineer an automatic mode of play,

- improved back up facilities to allow backing up to previous tricks, and
- the fine-tuning of the machine player communication issues.

While doing the parallel alpha-beta experiments, several possibilities meriting further consideration emerged. First, the parallel "environment" in which we conducted these experiments was an artificial construction from which we would not get any real time speedup. Therefore, should we decide to integrate a parallel search algorithm into the double-dummy player, a "real" time parallel environment needs to be accessed. Perhaps the Hypercube or Sequent machines at Oak Ridge will be installed with Lisp interpreters, and we can run in that parallel environment where communication times are realistic. Alternatively, a better communication system can be established using the Suns, perhaps making use of sockets to provide interprocess message passing. Secondly, given that speedups of up to 16.0 are reported using parallel algorithms developed for chess trees, new parallel algorithms specifically geared for pruning bridge trees could be developed.

In another scenario, we might develop heuristic plans or strategies to guide and limit the search. A plan might include such "high level" concepts as "play diamonds and then spades" as well as the more basic concepts like "if my partner plays high, I play low." The plans would have to be specific enough to enable the search to be completed quickly and broad enough to find a "good" solution. These plans might include independent evaluations of suits to determine how many tricks could be won in each suit. This type evaluation could certainly be done in parallel, and perhaps other aspects as well.

As another possibility, we could apply the parallelism to the current pruning techniques used in the double-dummy player and cut down on the per node expansion time. Williams [1989] reported that 43.6 nodes per CPU second were expanded when using the minimax search technique, while only 6.4 nodes per CPU second were expanded when all additional pruning techniques (good node order, strict bucketing, internal cutoffs, deferred branching, and hashing) were in use. As tens of thousands of nodes are expanded in games with 32 cards, reductions in the per node expansion time could be significant. For example, the good node ordering technique is essentially a sorting problem, and optimal parallel algorithms exist that take time O(log n) [Gibbons 1988, p. 181].

BIBLIOGRAPHY

BIBLIOGRAPHY

Adelson-Velsky, G.M., V.L. Arlazarov and M.V. Donskoy. *Algorithms for Games*. New York: Springer-Verlag, 1988.

Akl, S.G., D.T. Barnard and R.J. Doran. "Design, Analysis and Implementation of a Parallel Tree Search Algorithm." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-4, no. 2, March 1982, pp. 192-202.

Berlekamp, Elwyn R. "Program for Double-Dummy Bridge Problems -- A New Strategy for Mechanical Game Playing." *Journal of the ACM*, vol. 10, no. 4, 1963, pp. 357-364.

Campbell, M.S. and T.A. Marsland. "A Comparison of Minimax Tree Search Algorithms." *Artificial Intelligence*, vol. 20, no. 4, July 1983, pp. 347-367.

Cirelli, Colleen, et al. *Precision*. Unpublished manuscript. The University of Tennessee, Knoxville, Tennessee, 1989.

Computer Games II. Ed. David N.L. Levy. New York: Springer-Verlag, 1988.

Crawford, Chris. *The Art of Computer Game Design*. Berkeley, CA: Osborne, McGraw-Hill, Inc., 1984.

DeSatnick, Shelly. *Bridge for Everyone*. New York: First Avon Printing, 1982.

Finkel, R.A. and J.P. Fishburn. "Parallelism in Alpha-Beta Search." *Artificial Intelligence*, vol. 19, 1982, pp. 89-106.

Foley, J. D. and A. Van Dam. *Fundamentals of Interactive Computer Graphics*. Reading, Massachussetts: Addison-Wesley, 1982.

Gibbons, Alan and Wojciech Rytter. *Efficient Parallel Algorithms*. Cambridge, Great Britain: Cambridge University Press, 1988.

Hearn, Donald and Pauline M. Baker. *Computer Graphics*. Englwood Cliffs, New Jersey: Prentice-Hall, Inc., 1986.

Hyatt, R.M., B.M. Sutter and H.L. Nelson. "A Parallel Alpha/Beta Tree-Searching Algorithm." *University of Alabama at Birmingham Technical Report*, 1989.

Kaplan, Edgar, ed. *The Bridge World*, vol. 61, no. 2, November 1989, pp. 22-24.

Knuth, Donald E. and Ronald W. Moore. "An Analysis of Alpha-Beta Pruning." *Artificial Intelligence*, vol. 6, 1975, pp. 293-326.

Marsland, T.A. and M. Campbell. "Parallel Search of Strongly Ordered Game Trees." *Computing Surveys*, vol. 14, no. 4, December 1982, pp. 533-551.

Marsland, T.A. and F. Popowich. "Parallel Game-Tree Search." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-7, no. 4, July 1985, pp. 442-452.

Newborn, Monty and Danny Kopec. "Results of The Nineteenth ACM North American Computer Chess Championship." *Communications of the ACM*, vol. 32, no. 10, October 1989, pp. 1225-1230.

Quinn, M.J. *Designing Efficient Algorithms for Parallel Computers*. New York: McGraw-Hill, 1987.

Quinlan, J.R. "A Knowledge-Based System for Locating Missing high Cards in Bridge." In *Proceedings of the 4th IJCAI*, Tokyo, 1979.

Rich, Elaine. *Artificial Intelligence*. New York: McGraw-Hill Book Co., 1983.

Slagle, James R. and John K. Dixon. "Experiments with Some Programs that Search Game Trees." *Journal of the ACM*, vol. 16, no. 2, April 1969, pp. 189-207.

Stanier, Alan M. "BRIBIP: A Bridge-Bidding Program." In *Proceedings of the 4th IJCAI*, Tbilisi, 1975.

Stanier, Alan M. "Planning to Make Tricks at Bridge." In *Proceedings of the AISB Summer Conference*, 1976

Solomon, Eric. *Games Programming*. Cambridge, England: Cambridge University Press, 1984.

Watson, Louis H. *Watson's Classic Book on the Play of the Hand at Bridge*. New York: Barnes and Noble, Inc., 1969.

Williams, P. Susan McFaddin. *Search Techniques for the Game of Double-Dummy Bridge*. Master's thesis. The University of Tennessee, Knoxville, Tennessee, 1989.

APPENDIX

AN INTRODUCTION TO THE GAME OF BRIDGE

The following is an excerpt taken from Williams' thesis [1989]:

> . . .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,...the play begins with the player to the dealer's left playing a card followed by the other three players playing a card. All plyers 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.
>
> . . .[In Contract Bridge during the bidding phase] 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...[Regarding the scoring] An elaborate set of rules define the number of points awarded after a hand has been played...For a more detailed introduction to Contract Bridge, explore for example "Bridge For Everyone" by Shelly DeSatnick 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.

VITA

Elizabeth McFarlin Jones graduated with highest honors from The University of Tennessee, Knoxville in August of 1987 with a Bachelor of Arts Degree in Psychology. She was the Social Sciences Top Graduate in the College of Liberal Arts. That September she entered the graduate program in Computer Science at The University of Tennessee, Knoxville, where she met her future husband, Fermin Soriano Salkjelsvik. They were married in June of 1988. In May of 1989, she received the ACM Outstanding Graduate Teaching Assistant Award for teaching excellence. She graduated with a Master of Science Degree in Computer Science in May of 1990.