A Naïve Treatment of Self-Contained Digital Sentience in Pursuit of Erudition in a Stochastic Game

Robert Brady

April 14, 2005

Abstract

My tech-lab project deals with the field of Artificial Intelligence or more specifically, Machine Learning. I am designing an agent/environment for the card game of bridge. After it learns the rules, I will run simulations where it decides on its own what the best play is. The level of play for the agent will increase as the year continues because it will look up past decisions in its history to determine what the best bid or play is in a current state of the environment.

Background

Machine learning has been researched in the past and has dealt with bridge before. This area is new, though, and anything from intelligent agents for games to the traveling salesman problem count as part of it. An algorithm used for one problem can be applied in a similar manner to another such as the minimax algorithm or the backtracking search. To build on current work, there would have to be some sort of improvement on current bridge-playing agents such as Bridge Baron or GIB. Both of these programs play at a moderate level, but none of them can compare to an expert player.

The reason why an intelligent bridge-playing agent has been hard to program in the past is that bridge is a partially observable environment. In games such as chess, or checkers, the agent could conceivably come up with the best solution (given enough time to think about it) because it knows where everything is. In bridge, there are certain cards that haven't been played yet and although you may be able to guess where they are, you can not determine this with 100% certainty. This makes programming a learning agent for a partially observable environment much harder.

Progress

For the first semester, I worked on this program with regards to finishing programming in the rules to the game and some simple AI commands. When this was completed about a week before the semester was over, I began researching different AI algorithms that could be implemented for searching. This research halted once I realized the tree that would be searched was difficult to construct. I consulted my professional contact – Fred Gitleman – and he had also encountered this problem when programming a similar search algorithm. He talked me through the problems I had and gave some advice on where to find information that would help me with those problems.

During the third quarter, I worked solely on running an algorithm (the minimax algorithm) through a depth-first search with pruning of bad nodes. The algorithm still has a few problems and will hopefully be finished soon. Another portion of the code that I added this quarter deals with the machine learning part of my project. This part of the project stores information from the hand that the computer just played in a file that is essentially the computer's "brain." The brain stores information about how many tricks were taken with the combined hands in a trump suit or in no-trump. It uses this information for the bidding stage of the following hands. If the numbers it reads from the file are much lower than what it believes the current state of the environment is, it will bid higher and if the numbers are higher, it will try to refrain from bidding.

My future plans include testing of the program against other players at the school. I will use students from my tech-lab class for a preliminary test and then after the program has established a competitive nature with these kids, it will play against the bridge club on Fridays during school. I hope it will be able to compete with the students of the club at some point in the next quarter, but if this is an unrealistic goal, I will just try and improve upon it's algorithm as much as I possibly can before the end of the year. As it is currently, the program needs a little more work on the algorithm to make it fully operational before I open it up to tests from fellow students.

Code

This section is pretty much self explanatory. Some important sections are in bold and commented while the less important parts have been left out.

```
#include <stdlib.h>
#include <time.h>
#include <iostream.h>
#include <stdio.h>
#include <fstream.h>
   ifstream inf(”contracts.txt”);
ofstream outf;
   struct bid
/*
*Levels of bidding are 1-7 and suits are clubs, diamonds,
*hearts, spades, and no-trump represented by the first letter of
the name.
*Pass is represented by two zeros, double is X zero, and redouble
is XX.
*/
{
```

```
public:
char level;
char suit;
};
   struct contract
/*
* Stores same information as struct bid and also whether the
contract is doubled (1) or redoubled (2)
* Non-doubled contracts are (0)
*/
{
public:
char level;
char suit;
int X;
};
   int declarer=0;
contract this_hand;//global variable for the current contract
double contracts[14][2];
   class card
/*
*Creates a card with a suit and rank both of which are charac-
ters.
*0 is clubs
*1 is diamonds
*2 is hearts
*3 is spades
*The numbers 1-9 are represented by the characters '1'-'9' and
ten through
*ace are represented by the first letter.
*/
//has been removed
   class player
```

```
/*
*Creates an identity with a name and a hand of 13 cards. Also
stores whether
*a card has been played.
*/
{
void display();
card get_hand(int n);
void count_suit_length();
double count_points();
int hold_ace(char suit);
int hold_king(char suit);
int hold_queen(char suit);
int hold_jack(char suit);
int hold_ten(char suit);
double losers(char suit);
int sumspots(char suit);
int numspots(char suit);
double spots(char suit);
int suit_quality(char suit, double totalpoints, int suitlength);
void fix_hand(int n, card one);
void sort_hand();
int get_numclubs();
int get_numdiamonds();
int get_numhearts();
int get_numspades();
int getplayed(int x);
double getnumpoints();
void change_numcards(int n);
int not_legal_bid(bid test, bid current[60], int totalbids);
int balanced();
bid opening(bid current[60], int totalbids, player one);
bid bidding(bid current[60], int totalbids, player one);
```

//this will eventually be the function used for picking a bid during the bidding
int declarer (bid final[]);
int highest(card one, card two, card three, card four, char lead);
int legal_card(char suit, int x, card play);
int pick_card(bid final[], char suit, player &one, player &two, player &three, player &four, int p, card A, card B, card C, int ns, int ew);//this function determines the best play using a pruning search algorithm
card play_card(bid final[], char suit, player &one, player &two, player &three, player &four, int p, card A, card B, card C, int ns, int ew);
int play_trick_4_comps(bid final[], int winner, player &north, player &east, player &south, player &west, int ns, int ew);
int play_trick_3_comps(bid final[], int winner, player &north, player &east, player &south, player &west, int ns, int ew);
void show(player north, player east, player south, player west);
void show_pretty(player north, player east, player south, player west);
void make_seen(player one, player two, player three);
card get_seen(int a, int b);
};
class deck
/*
*Makes a standard 52 card deck.
*Has functions such as shuffling and dealing the deck.
*/
//removed
class FullDeal
{
FullDeal::FullDeal(player one, player two, player three, player four, int n)
{

```
all = new card*[4];
for (int a=0; a<4; a++)
all[a] = new card[13];
int m, o, p, place=0, num;
north=one;
east=two;
south=three;
west=four;
int sample[52];
int places[52];
for (m=0; m<52; m++)
{
sample[m]=0;
places[m]=0;
}
player *thing[5];
thing[1]=&one;
thing[2]=&two;
thing[3]=&three;
thing[4]=&four;
//marks cards that have been found
for (m=0; m<4; m++)
for (o=0; o<13; o++)
{
all[m][o]=thing[n]->get_seen(m, o);
if (all[m][o].getsuit()=='C')
sample[all[m][o].newrank()-2]=1;
else if (all[m][o].getsuit()=='D')
sample[13+all[m][o].newrank()-2]=1;
else if (all[m][o].getsuit()=='H')
sample[26+all[m][o].newrank()-2]=1;
else if (all[m][o].getsuit()=='S')
sample[39+all[m][o].newrank()-2]=1;
```

```
}
//creates array of places to replace the missing cards
for (m=0; m<52; m++)
if (sample[m]==0)
places[place++]=m;
//arranges all missing cards using array of places
for (m=0; m<4; m++)
for (o=0; o<13; o++)
if (all[m][o].getsuit()=='A')
for (p=0; p<place; p++)
{
if (places[p]<13)
{
num=places[p]+2;
all[m][o].change_suit('C');
}
else if (places[p]<26)
{
num=places[p]+2-13;
all[m][o].change_suit('D');
}
else if (places[p]<39)
{
num=places[p]+2-26;
all[m][o].change_suit('H');
}
else if (places[p]<52)
{
num=places[p]+2-39;
all[m][o].change_suit('S');
}
if (num<=9)
all[m][o].change_rank(num+'0');
```

```
if (num==10)
all[m][o].change_rank(’T’);
if (num==11)
all[m][o].change_rank(’J’);
if (num==12)
all[m][o].change_rank(’Q’);
if (num==13)
all[m][o].change_rank(’K’);
if (num==14)
all[m][o].change_rank(’A’);
}
}
};
    //this determines where the remaining cards are based upon
one player’s assumptions
class position
{
private:
int ns_tricks;
int ew_tricks;
card left[4][13];
public:
position::position()
{
}
position::position(int ns, int ew, card** stuff)
{
ns_tricks=ns;
ew_tricks=ew;
for (int n=0; n<4; n++)
for (int m=0; m<13; m++)
left[n][m]=stuff[n][m];
}
```

```
position& operator = (position& rhs)
{
ns_tricks=rhs.ns_tricks;
ew_tricks=rhs.ew_tricks;
for (int n=0; n<4; n++)
for (int m=0; m<13; m++)
left[n][m]=rhs.left[n][m];
}
position change_position(int ns, int ew, card stuff[4][13],
int p);
} ;
   struct leaf
{
public:
position *parent;
position *first;
position *second;
position *third;
position *fourth;
};
   void make_leaves(leaf *p, position *top, position *one, posi-
tion *two, position *three, position *four)
{
p->parent=top;
p->first=one;
p->second=two;
p->third=three;
p->fourth=four;
}
   int player::not_legal_bid(bid test, bid current[60], int total-
bids)
{
if (test.level==’0’)
```

```
return 0;
else if (totalbids==0 && test.level=='X')
return 1;
else if (totalbids==1 && test.suit=='X')
return 1;
else if (totalbids==1 && test.level=='X' && test.level=='0')
return current[0].level-'0';
else if (totalbids==2)
{
if (test.level=='X' && test.suit=='X')
if (current[1].level=='X')
return 0;
else
return 1;
}
else if (test.level=='X' && test.suit=='X')
{
if (current[totalbids-1].suit=='0')
{
if (current[totalbids-1].level=='X')
return 0;
else if (current[totalbids-1].level=='0' &&
current[totalbids-2].level=='0' &&
current[totalbids-3].level=='X' &&
current[totalbids-3].suit=='0')
return 0;
}
return 1;
}
else if (test.level=='X' && test.suit=='0')
{
if (current[totalbids-1].level!='0' && current[totalbids-1].level!='X')
{
```

```
if (current[totalbids-1].level>'0')
return 0;
else if (current[totalbids-1].level=='0' &&
current[totalbids-2].level=='0' &&
current[totalbids-3].level>'0')
return 0;
}
return 1;
}
int n=1;
while(current[totalbids-n].level=='X' —— current[totalbids-n].level=='0')
n++;
if (test.level<current[totalbids-n].level)
return 1;
else if (test.level==current[totalbids-n].level && test.suit<=current[totalbids-
n].suit)
return 1;
else if (test.level==current[totalbids-n].level && current[totalbids-
n].suit=='N')
return 1;
return 0;
}
int player::balanced()
{
if(numspades>=2 && numhearts>=2 && numclubs>=2 &&
numdiamonds>=2)
if(numspades<=5 && numhearts<=5 && numclubs<=5 &&
numdiamonds<=5)
return 1;
return 0;
}
bid player::opening(bid current[60], int totalbids, player one)
{
```

```
bid now;
now.suit='0';
now.level='0';
count_suit_length();
if(balanced() && count_points()>=21 && count_points()<=23)
{
if (suit_quality('C', count_points(), numclubs)>=2 &&
suit_quality('D', count_points(), numdiamonds)>=2 &&
suit_quality('H', count_points(), numhearts)>=2 &&
suit_quality('S', count_points(), numspades)>=2)
now.suit='N';
now.level='2';
}
else if (balanced() && count_points()>=25 && count_points()<=27)
{
now.suit='N';
now.level='3';
}
else if (one.count_points()>=16)
{
now.suit='C';
now.level='1';
}
else if (one.count_points()<=15 && one.count_points()>=11 &&
numspades>=5)
{
now.suit='S';
now.level='1';
}
else if (one.count_points()<=15 && one.count_points()>=11 &&
numhearts>=5)
{
now.suit='H';
now.level='1';
}
else if (balanced() && one.count_points()<=15 && one.count_points()>=13)
{
now.suit='N';
now.level='1';
}
else if (one.count_points()>=11 && numclubs>=6)
{
now.suit='C';
now.level='2';
}
else if (one.count_points()>=11 && numclubs==5 && (numspades==4
—— numhearts==4))
{
now.suit='C';
now.level='2';
}
else if (one.count_points()<=15 && one.count_points()>=11)
{
now.suit='D';
now.level='1';
}
else if (one.count_points()<=10 && one.count_points()>=5 &&
numspades==6)
{
now.suit='S';
now.level='2';
}
else if (one.count_points()<=10 && one.count_points()>=5 &&
numhearts==6)
{
now.suit='H';
now.level='2';
}
else if (one.count_points()<=10 && one.count_points()>=5 &&
numdiamonds==6)
{
now.suit='D';
now.level='2';
}
else if (one.count_points()<=10 && one.count_points()>=5 &&
numspades==7)
{
now.suit='S';
now.level='3';
}
else if (one.count_points()<=10 && one.count_points()>=5 &&
(numhearts==7 —— numhearts==8))
{
now.suit='H';
now.level='3';
}
else if (one.count_points()<=10 && one.count_points()>=5 &&
numdiamonds==7)
{
now.suit='D';
now.level='3';
}
else if (one.count_points()<=10 && one.count_points()>=5 &&
numclubs==7)
{
now.suit='C';
now.level='3';
}
else if (one.count_points()<=10 && one.count_points()>=5 &&
numclubs>=8 && numclubs<=10)
{
now.suit='C';
now.level='4';
}
else if (one.count_points()<=10 && one.count_points()>=5 &&
numdiamonds>=8 && numdiamonds<=10)
{
now.suit='D';
now.level='4';
}
else if (one.count_points()<=10 && one.count_points()>=5 &&
(numhearts==9 —— numhearts==10))
{
now.suit='H';
now.level='4';
}
else if (one.count_points()<=10 && one.count_points()>=5 &&
(numspades==9 —— numspades==10))
{
now.suit='S';
now.level='4';
}
else if (numspades==13)
{
now.suit='S';
now.level='7';
}
else if (numhearts==13)
{
now.suit='H';
now.level='7';
}
```

```
else if (numdiamonds==13)
{
now.suit='D';
now.level='7';
}
else if (numclubs==13)
{
now.suit='C';
now.level='7';
}
else if (one.count_points()==37) {
now.suit='N';
now.level='7';
}
if (not_legal_bid(now, current, totalbids))
now=bidding(current, totalbids, one);
thistime[bidnumber]=now;
bidnumber++;
return now;
}
bid player::bidding(bid current[60], int totalbids, player one)
{
bid now;
int suit;
now.level=rand()%7+'1';
suit=rand()%8;
if (!suit)
{
now.suit='0';
now.level='0';
}
if (suit==1)
now.suit='C';
```

```
if (suit==2)
now.suit='D';
if (suit==3)
now.suit='H';
if (suit==4)
now.suit='S';
if (suit==5)
now.suit='N';
if (suit==6)
{
now.suit='0';
now.level='X';
}
if (suit==7)
{
now.suit='X';
now.level='X';
}
if (not_legal_bid(now, current, totalbids))
now=one.bidding(current, totalbids, one);
thistime[bidnumber]=now;
bidnumber++;
return now;
}
int player::declarer (bid final[])//this function finds who the de-
clarer of a certain hand is based upon the giving bidding
{
this_hand.X=0;
this_hand.level='0';
this_hand.suit='0';
bid one, two, three, store1, store2, store3;
store1.suit='0';
store1.level='0';
```

```
store2.suit='0';
store2.level='0';
store3.suit='0';
store3.level='0';
int place=0;
for (int n=0; n<60; n+=3)
{
one=final[n];
two=final[n+1];
three=final[n+2];
if (one.level!='0' && store1.level=='0')
{
store1=one;
if (store1.level!='X')
place=n;
}
else if (one.level!='0' && store2.level=='0')
{
store2=one;
if (store2.level!='X')
place=n;
}
else if (one.level!='0' && store3.level=='0')
{
store3=one;
if (store3.level!='X')
place=n;
}
else if (one.level!='0')
{
store1=store2;
store2=store3;
store3=one;
```

```
if (store3.level!='X')
place=n;
}
if (two.level!='0' && store1.level=='0')
{
store1=two;
if (store1.level!='X')
place=n+1;
}
else if (two.level!='0' && store2.level=='0')
{
store2=two;
if (store2.level!='X')
place=n+1;
}
else if (two.level!='0' && store3.level=='0')
{
store3=two;
if (store3.level!='X')
place=n+1;
}
else if (two.level!='0')
{
store1=store2;
store2=store3;
store3=two;
if (store3.level!='X')
place=n+1;
}
if (three.level!='0' && store1.level=='0')
{
store1=three;
if (store1.level!='X')
```

```
place=n+2;
}
else if (three.level!='0' && store2.level=='0')
{
store2=three;
if (store2.level!='X')
place=n+2;
}
else if (three.level!='0' && store3.level=='0')
{
store3=three;
if (store3.level!='X')
place=n+2;
}
else if (three.level!='0')
{
store1=store2;
store2=store3;
store3=three;
if (store3.level!='X')
place=n+2;
}
if (final[0].level=='0' && final[1].level=='0' && final[2].level=='0'
&& final[3].level=='0')
return 0;
if (one.level=='0' && two.level=='0' && three.level=='0')
{
if (store2.level=='0' && store3.level=='0')
{
this_hand.suit=store1.suit;
this_hand.level=store1.level;
}
else if (store3.level=='0' && store2.level=='X')
```

```
{
this_hand.X=1;
this_hand.suit=store1.suit;
this_hand.level=store1.level;
}
else if (store3.level=='0')
{
this_hand.suit=store2.suit;
this_hand.level=store2.level;
}
else if (store3.suit=='X' && store3.level=='X')
{
this_hand.X=2;
this_hand.suit=store1.suit;
this_hand.level=store1.level;
}
else if (store3.level=='X' && store3.suit=='0')
{
this_hand.X=1;
this_hand.suit=store2.suit;
this_hand.level=store2.level;
}
else if (store3.suit=='X')
{
this_hand.X=2;
this_hand.suit=store1.suit;
this_hand.level=store1.level;
}
else
{
this_hand.suit=store3.suit;
this_hand.level=store3.level;
}
```

```
}
}
if (place%4==0)
return 1;
if (place%4==1)
return 2;
if (place%4==2)
return 3;
if (place%4==3)
return 4;
return 0;
}
int player::highest(card one, card two, card three, card four, char
lead)
{
if ((one.getsuit()==this_hand.suit && two.getsuit()!=this_hand.suit)
——
(one.getsuit()==this_hand.suit && one.newrank()>two.newrank())
——
(one.newrank()>two.newrank() && two.getsuit()!=this_hand.suit
&& one.getsuit()==lead) ——
(two.getsuit()!=lead && one.getsuit()==lead && two.getsuit()!=this_hand.suit))
{
if ((one.getsuit()==this_hand.suit && three.getsuit()!=this_hand.suit)
——
(one.getsuit()==this_hand.suit && one.newrank()>three.newrank())
——
(one.newrank()>three.newrank() && three.getsuit()!=this_hand.suit
&& one.getsuit()==lead) ——
(three.getsuit()!=lead && one.getsuit()==lead && three.getsuit()!=this_hand.suit))
{
if ((one.getsuit()==this_hand.suit && four.getsuit()!=this_hand.suit)
——
(one.getsuit()==this_hand.suit && one.newrank()>four.newrank())
——
(one.newrank()>four.newrank() && four.getsuit()!=this_hand.suit
&& one.getsuit()==lead) ——
(four.getsuit()!=lead && one.getsuit()==lead && four.getsuit()!=this_hand.suit))
return 1;
else
return 4;
}
else
{
if ((three.getsuit()==this_hand.suit && four.getsuit()!=this_hand.suit)
——
(three.getsuit()==this_hand.suit && three.newrank()>four.newrank())
——
(three.newrank()>four.newrank() && four.getsuit()!=this_hand.suit
&& three.getsuit()==lead) ——
(four.getsuit()!=lead && three.getsuit()==lead && four.getsuit()!=this_hand.suit))
return 3;
else
return 4;
}
}
else
{
if ((two.getsuit()==this_hand.suit && three.getsuit()!=this_hand.suit)
——
(two.getsuit()==this_hand.suit && two.newrank()>three.newrank())
——
(two.newrank()>three.newrank() && three.getsuit()!=this_hand.suit
&& two.getsuit()==lead) ——
(two.getsuit()!=lead && three.getsuit()==lead && two.getsuit()!=this_hand.suit))
{
```

```
if ((two.getsuit()==this_hand.suit && four.getsuit()!=this_hand.suit)
——
(two.getsuit()==this_hand.suit && two.newrank()>four.newrank())
——
(two.newrank()>four.newrank() && four.getsuit()!=this_hand.suit
&& two.getsuit()==lead) ——
(four.getsuit()!=lead && two.getsuit()==lead && four.getsuit()!=this_hand.suit))
return 2;
else
return 4;
}
else
{
if ((four.getsuit()==this_hand.suit && three.getsuit()!=this_hand.suit)
——
(four.getsuit()==this_hand.suit && four.newrank()>three.newrank())
——
(four.newrank()>three.newrank() && three.getsuit()!=this_hand.suit
&& four.getsuit()==lead) ——
(three.getsuit()!=lead && four.getsuit()==lead && three.getsuit()!=this_hand.suit)
return 4;
else
return 3;
}
}
}
int player::legal_card(char suit, int x, card play)
{
if (suit=='a' && played[x]==1)
return 0;
else
{
if (suit=='C' && numclubs==0 && played[x]==0)
```

```
return 1;
else if (suit==’D’ && numdiamonds==0 && played[x]==0)
return 1;
else if (suit==’H’ && numhearts==0 && played[x]==0)
return 1;
else if (suit==’S’ && numspades==0 && played[x]==0)
return 1;
else if (play.getsuit()!=suit && suit!=’a’)
return 0;
else if (played[x]==1)
return 0;
else
return 1;
}
}
//this function determines the card to play
int player::pick_card(bid final[], char suit, player &one,
player &two, player &three, player &four, int p, card
A, card B, card C, int ns, int ew)
{
int n, m; //assignment of cards to what is known
one.make_seen(two, three, four);
   //guesses for hand
// not yet implemented
   //set up all outstanding cards
int num;
if (name[0]==’N’)
num=1;
else if (name[0]==’E’)
num=2;
else if (name[0]==’S’)
num=3;
else if (name[0]==’W’)
```

```
num=4;
FullDeal temp(one, two, three, four, num);
   //create list of possible plays
int possible[13], places[13];
num=0;
for (n=0; n<13; n++)
if (possible[n]=legal_card(suit, n, hand[n]))//this works
dont change it 12/9/04
num++;
if (num==1)
for (n=0; n<13; n++)
if (possible[n])
return n;
num=0;
for (n=0; n<13; n++)
places[n]=0;
for (n=0; n<13; n++)
if (possible[n])
places[num++]=n+1;
   //put possibilities into a tree
leaf top;
if (numcards<=4)
{
position now(ns, ew, temp.get_all());
//make_leaves(&top, &now, &now.change(1), &now.change(2),
&now.change(3), &now.change(4));
   //use search algorithm to determine best possibility
// :run for averages and pick the best number based on
score after hand
}
   //return this value to play_card function
return rand()%13;
}
```

```
card player::play_card(bid final[], char suit, player &one, player
&two, player &three, player &four, int p, card A, card B, card
C, int ns, int ew)
{
card play;
int n, win;
if (numcards==1)
for (n=0; n<13; n++)
if (played[n]==0)
{
play=hand[n];
played[n]=1;
numcards=0;
return play;
}
count_suit_length();
int x=pick_card(final, suit, one, two, three, four, p, A, B, C, ns,
ew);
play=hand[x];
if (legal_card(suit, x, play)==0)
play=play_card(final, suit, one, two, three, four, p, A, B, C, ns,
ew);
for (n=0; n<13; n++)
if (play.getsuit()==hand[n].getsuit() && play.getrank()==hand[n].getrank())
played[n]=1;
return play;
}
int player::play_trick_4_comps(bid final[], int winner, player &north,
player &east, player &south, player &west, int ns, int ew)
{
card n, e, s, w;
card blank(0, 'A');
if (winner==1)
```

```
{
n=north.play_card(final, 'a', north, east, south, west, 1, blank,
blank, blank, ns, ew);
north.change_numcards(-1);
e=east.play_card(final, n.getsuit(), north, east, south, west, 2,
n, blank, blank, ns, ew);
east.change_numcards(-1);
s=south.play_card(final, n.getsuit(), north, east, south, west, 3,
n, e, blank, ns, ew);
south.change_numcards(-1);
w=west.play_card(final, n.getsuit(), north, east, south, west, 4,
n, e, s, ns, ew);
west.change_numcards(-1);
n.trick_diamond(n, e, s, w);
return highest(n, e, s, w, n.getsuit());
}
else if (winner==2)
{
e=east.play_card(final, 'a', north, east, south, west, 1, blank,
blank, blank, ns, ew);
east.change_numcards(-1);
s=south.play_card(final, e.getsuit(), north, east, south, west, 2,
e, blank, blank, ns, ew);
south.change_numcards(-1);
w=west.play_card(final, e.getsuit(), north, east, south, west, 3,
e, s, blank, ns, ew);
west.change_numcards(-1);
n=north.play_card(final, e.getsuit(), north, east, south, west, 4,
e, s, w, ns, ew);
north.change_numcards(-1);
n.trick_diamond(n, e, s, w);
return highest(n, e, s, w, e.getsuit());
}
```

```
else if (winner==3)
{
s=south.play_card(final, ’a’, north, east, south, west, 1, blank,
blank, blank, ns, ew);
south.change_numcards(-1);
w=west.play_card(final, s.getsuit(), north, east, south, west, 2,
s, blank, blank, ns, ew);
west.change_numcards(-1);
n=north.play_card(final, s.getsuit(), north, east, south, west, 3,
s, w, blank, ns, ew);
north.change_numcards(-1);
e=east.play_card(final, s.getsuit(), north, east, south, west, 4, s,
w, n, ns, ew);
east.change_numcards(-1);
n.trick_diamond(n, e, s, w);
return highest(n, e, s, w, s.getsuit());
}
else if (winner==4)
{
w=west.play_card(final, ’a’, north, east, south, west, 1, blank,
blank, blank, ns, ew);
west.change_numcards(-1);
n=north.play_card(final, w.getsuit(), north, east, south, west, 2,
w, blank, blank, ns, ew);
north.change_numcards(-1);
e=east.play_card(final, w.getsuit(), north, east, south, west, 3,
w, n, blank, ns, ew);
east.change_numcards(-1);
s=south.play_card(final, w.getsuit(), north, east, south, west, 4,
w, n, e, ns, ew);
south.change_numcards(-1);
n.trick_diamond(n, e, s, w);
return highest(n, e, s, w, w.getsuit());
}
}
    int main()
{
srand(time(NULL));
int n, score;
player north("North"), south("South"), east("East"), west("West");
deck test(1);
test.shuffle();
test.deal(east, south, west, north);
//test.input_hands(east, south, west, north);
bids now(1, north, east, south, west);
north.show_pretty(north, east, south, west);
now.show_bidding();
cout << endl;
n=now.show_contract_and_declarer(north);
declarer=n;
n++;
if (n==5)
n=1;
if (n==0)
return 0;
score=now.play_hand_4_comps(n, north, east, south, west);
cout << "Score:" << score << endl;
return 0;
}
```

References

1. Fred Gitelman (fred@bridgebase.com) – A programmer who also is an expert bridge player. He advised me on how to look through a tree to find the solution comparable to a minimax search. 2. Russell, S. and Norvig, P. Artificial Intelligence A Modern Approach Seco
Prentice Hall, NJ. 2003.