University of Victoria

CENG420

Artificial Intelligence

Instructor: Dr. Stephen Neville

2-Connect-2 AI Engine Project

Prepared by

Stefan Pantazi

Table of contents

1 Introduction ... 2
1.1 The 2-Connect-2 game concept ... 2
1.2 The 2-Connect-2 game controller ... 2
1.3 The 2-Connect-2 game tournament ... 3
1.4 The approach for the design, building and testing the AI engine ... 3
2 Definitions ... 4
3 Data Structures ... 5
3.1 The game board ... 5
3.2 The row indexes of the next move ... 5
3.3 The columns sorting indexes and counts ... 5
3.4 The 2c2 group records ... 6
3.5 The board map and the board map counts ... 7
4 Programming language choices ... 8
5 Search methodology ... 9
6 Game heuristics ... 11
6.1 The "space gap" heuristic ... 12
6.2 The "odd-even hot square" heuristic ... 13
6.3 The "double hot square" situation ... 14
6.4 The defensive/offensive heuristic ... 15
6.5 The current version of the heuristic ... 15
7 Improvements ... 16
8 Known Bugs ... 16
9 Discussion & Conclusions ... 16
10 References ... 17
11 Appendices ... 18

1 Introduction

1.1 The 2-Connect-2 game concept

This project objective is the design, implementation and testing of an artificial intelligence (AI) engine capable of playing a turn based board game, 2-Connect-2. The 2-Connect-2 game concept was derived from the classic Connect-4 game, which consists in a vertical board with six rows and seven columns. In the original Connect-4 game, there are 21 red and 21 black game pieces, which could be placed, one every move, on any board column, falling down to the last row or piling up on the last piece introduced in the same column. In Connect-4 the winner would have to achieve a number of 4 pieces in line in any of the possible directions: horizontal, vertical, diagonal (first or second). The 2-Connect-2, besides introducing a variable board size, adds a neutral game piece type that can be played by both of the players and changes the winning condition to requiring a winning group of pieces to contain two neutral pieces. In Figure 1, the black player played the second and has a winning line consisting of two black pieces and two green pieces.

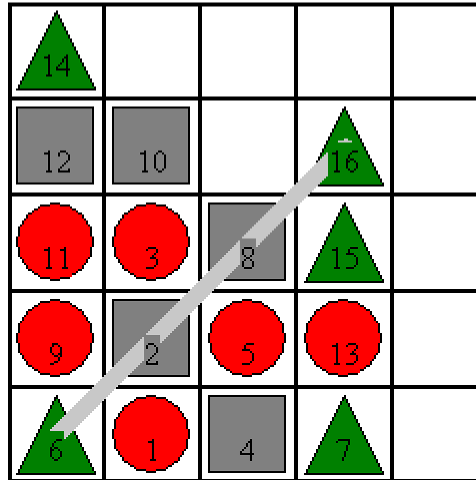

Figure 1. Black player winning a 2-Connect-2 game played on a atypical board with five rows and five columns. In order to be distinguished on a black&white printout, the neutral green game pieces are shown as triangles, the black player pieces are shown as rectangles and the red player pieces are shown as circles. The game history can be reconstructed using the move number on each game piece

1.2 The 2-Connect-2 game controller

In order to support the development and testing of the 2-Connect-2 AI engines, a game controller was created for Linux and Sun Solaris platforms. The controller manages 2-Connect-2 games between any combination of human and computer players and also provides timing information on the player moves and games played. In addition, with a simple board inverting routine, it simplifies the AI engines communication interface requirements by always returning game boards from the red player perspective. Hence, the AI engines, which are stand alone executables designed to run on a Sun Solaris platform, will have to be build to always play as the red player.

The communication of the current board and next move information between the standalone AI engines and the game controller is made through the standard input (stdin) and standard output (stdout) using a clear specification (1). In addition, the AI engines may pass debugging information on the standard

error. Under the current game specifications, the AI engines are required to read a board, calculate the next move in less than 3 minutes, pass the move to the game controller and then exit. In addition, the design of the AI engines must also take into account the total game time limit of 20 minutes (1).

1.3 The 2-Connect-2 game tournament

After the design and implementation phases, the AI engine may be tested in a tournament, against other engines. In order to encourage the development of winning AI engines, the tournament controller awards 3 points for each win and 1 point for each tie game.

1.4 The approach for the design, building and testing the AI engine

Because the development of the AI engine was done on a Windows platform using the Object Pascal programming language and the Delphi IDE, a separate game controller was needed. The Windows game controller provided the initial insights on the game playing strategies, the possibility to easily print game board visuals with particular game situations and the debugging support for the AI routines development. As a downside, the Object Pascal code had to be regularly translated into ANSI C, in order to be portable to the Sun Solaris platforms where the game tournament was running. However, the translation was helpful with spotting minor bugs in several instances.

Initially the search technique was a simple 1-ply search which evolved gradually in 25 subsequent versions of the AI engine, towards a more efficient search which was an adaptation of the classic MiniMax algorithm with Alpha-Beta pruning. During the first 20 versions, the search space-heuristic tradeoff was biased towards a better informed heuristic in the detriment of the search space which was limited to 2 ½ plies (i.e. 5 moves ahead). Given the results in the tournament, starting with the version 21 of the AI engine, the design was modified so the AI engine was able to perform a search of 3 ½ plies deep but using a less informed heuristic.

2 Definitions

Although the 2-Connect-2 game specification is clear, sometimes the choice of words may mislead the reader, especially when discussing various aspects of the heuristic development. Therefore, before advancing into the AI engine design and implementation description, the following concept should be defined, as they will be used consistently throughout the document and in the submitted code.

Row count	the number of rows of the board
Column count	the number of columns of the board
Row index	a number in the interval [0,Row Count-1]; the row with the index of 0 is located at the topmost side of the board
Column index	a number in the interval [0,Column Count-1]; the Column with the index of 0 is located at the leftmost side of the board
Even row	a row having an even index (i.e. 0, 2, 4, etc.)
Odd row	a row having an odd index (i.e. 1, 3, 5, etc)
2C2 direction	one of the four possible directions on the 2-Connect-2 board: row direction (i.e. horizontal), column direction (i.e. vertical), first diagonal and second diagonal; the first diagonal direction is the left-top-corner→right-bottom-corner direction
2C2 game piece	the 2C2 game pieces can be neutral (e.g. green) and player specific (e.g. red or black)
Square	a square on the board which can be located by its row and column indexes; initially a square contains a space and can be filled with a neutral piece or with a player specific piece
Free square	a square that does not contain any game piece
2C2 Group	a group of four squares on the 2Connect-2 board which are consecutive in one of the 2C2 directions
Hot square	a square on the board that, when filled with a piece by any of the players, ends the game through a win or a tie situation. A hot square is always part of a 2C2 group having three of the squares already occupied in one of the two following configurations: o 1 space, 1 neutral piece, 2 player specific pieces o 1 space, 2 neutral pieces, 1 player specific piece
Cold square	a square adjacent to a hot square having a row index equal to that of the hot square plus one
Even hot square	a hot square located on an even row
Odd hot square	a hot square located on an odd row
Even board	a game board with an even number of squares
Odd board	a game board with an odd number of squares
Win hot square	a hot square that, when filled, causes the player which made the last move to win
Loss hot square	a hot square that, when filled, causes the player which made the move before the last move to lose
Tie hot square	a hot square that, when filled, causes a tie game
Space gap	the minimum number of moves needed to fill a 2C2 Group completely; for a hot square, the space gap could be thought of as the analogy with the "temperature" of the hot square, which represents the minimum number of moves to turn the hot square into a win

Table 1. AI engine design concept definitions

3 Data Structures

Because of the variable size of the board, all the array structures in the AI engine are dynamic arrays whose dimensions are determined at run time and for which memory is allocated dynamically.

3.1 The game board

The most important data structure, the game board, is a dynamic two-dimensional array of integers, which can contain only four possible integer values.

```
..
  int gpRed=0, gpBlack=1, gpGreen=2, gpSpace=3;
..
```

Previously to the board array allocation, the correct determination of the number of rows and columns based on the information passed by the game controller, is necessary.

```
  int **Board;
..
//allocating the rows
  Board=(int **) calloc(RowCount,sizeof(int *));
//allocating the columns
  for (i = 0; i < RowCount; i++)
  {
    Board[i]=(int *) calloc(ColCount,sizeof(int));
  }
..
```

3.2 The row indexes of the next move

In addition to the board array, a necessary data structure is an one-dimensional dynamic array of integers used to stored the row indexes of the next move.

```
  int *NextRowIdx;
```

Initially, this array is initialized to have all its elements equal to –1, after which it will be filled with the appropriate row indexes, resulted from the parsing of the board string received from the game controller.

3.3 The columns sorting indexes and counts

As of version 21 of the AI engine, a column sorting procedure was introduced in order to speed up the alpha beta pruning search. The sorting is based on the assumption that the columns having the biggest number 2C2 groups containing a free square in that column are the most important and they should be searched first. Hence the need to keep track, in separate data structures, the information about the number of pieces placed in each column and also about the indexing resulted form the search.

```
  int *ColumnSortIdx;
  int *ColumnSortCounts;
```

Because the sorting procedure is done only once, at the initialization of the AI engine, the search needs not be very efficient, but straightforward. A descending bubble-sort was therefore implemented.

```
//bubble sorting the board columns based on ColumnSortCounts[j]
//the columns with least pieces are the most important
  for (j = 0; j<ColCount-1; j++)
  {
    for (k=0; k<ColCount-1-j; k++)
    {
      if (ColumnSortCounts[k]<ColumnSortCounts[k+1])
      {
//here we swap the values
        TempValue=ColumnSortCounts[k];
        ColumnSortCounts[k]=ColumnSortCounts[k+1];
        ColumnSortCounts[k+1]=TempValue;
        TempValue=ColumnSortIdx[k];
        ColumnSortIdx[k]=ColumnSortIdx[k+1];
        ColumnSortIdx[k+1]=TempValue;
      }
    }
  }
```

3.4 The 2c2 group records

The playing of the 2-Connect-2 game involves capturing information about the various 2C2 groups on the game board. In addition, this information would have to be accessed very often by the heuristic functions that guide the move decision process. Hence, the information about these groups should be kept separately, in their own data structures, which are implemented as a two-dimensional dynamic arrays of group records.

```
  typedef struct TGroupRecType
  {
    int Count[4];
//number of moves away from filling the group
    int SpaceGap;
    int P1Row;
    int P1Col;
    int Direction;
    int *P[4];
  } GroupRec;

  GroupRec **RowGroup, **ColGroup, **FirstDiagGroup, **SecDiagGroup;
```

The GroupRec structure contains information about the counts of the pieces of each type, about the space gap (see definitions section), the row and column indexes of the 2C2 group's first square on the board and the direction of the 2C2 group which can take the following integer values:

```
  int wdRow=0, wdColumn=1, wdFirstDiagonal=2, wdSecondDiagonal=3;
```

In addition, the GroupRec structure contains four pointers to the corresponding actual integer elements in the game board array.

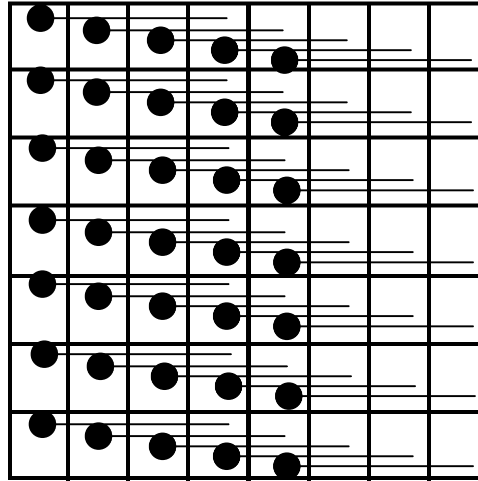

Figure 2. The 2C2 groups in the row direction. The first square of the group is shown using a black dot

On a board with 7 rows and 8 columns there are 35 2C2 groups in the row direction, organized in a two-dimensional matrix of 7 rows and 5 columns, i.e. 0..RowCount-1,0..ColumnCount-4 (Figure 2). Similarly, there are 32 2C2 groups in the column direction, i.e. 0..RowCount-4,0..ColumnCount-1, and 20 2C2 groups in the two diagonal directions, i.e. 0..RowCount-4,0..ColumnCount-4. These arrays have been used initially for the win function development (1) and in the first versions of the AI engine, for calculating the heuristic function values.

3.5 The board map and the board map counts

As of version 21 of the AI engine, another data structure has been added. In order to speed up the access to the 2c2 group records, this data structure maintains, for each square on the game board, a list of pointers to the 2C2 groups that contain that square. In addition, a two dimensional dynamic array of integers, identical to the game board array, is used to store the different 2C2 counts associated with each square on the game board.

```
GroupRec ****BoardMap;
int **BoardMapCount;
```

To serve its purpose, the BoardMap data structure must be implemented as a three-dimensional dynamic array of pointers, with its third dimension equal to 16, as this represents the maximum number of 2C2 groups that a particular square may belong to (Figure 3). The construction of the board map is done only once, at the initialization of the AI engine. Subsequently, during the move decision process (i.e. the alpha-beta algorithm), for each move tried, the 2C2 group records are updated accordingly, in order to reflect the changes on the game board. The current state of the search is pushed or popped from two additional one-dimensional arrays that keep track of the row and column indexes of the searched moves.

```
int SaveColMoveIdx[100];
int SaveRowMoveIdx[100];
```

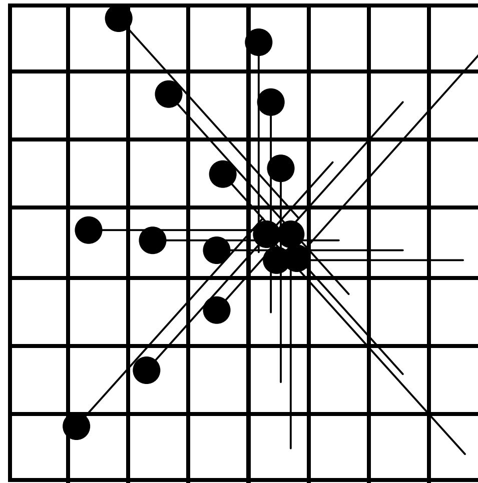

Figure 3. The maximum number of 2C2 groups that a square may belong to

4 Programming language choices

The choice of programming language was twofold. The first choice was based on the extensive experience on programming using Object Pascal and the Borland Delphi IDE 6.0, which is a Rapid Application Development (RAD) tool with very good debugging facilities. This allowed the creation of a separate game controller that could run under a Windows environment and would help the implementation, debugging and testing of the resulting AI code (Figure 4).

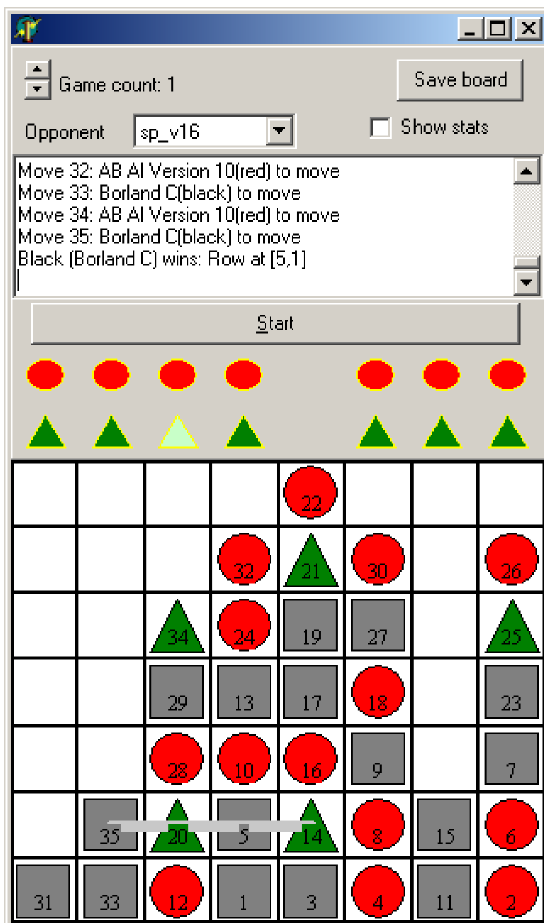

Figure 4. The Windows version of the game controller

Although there are Pascal compilers that would compile code in order to run on a Sun Solaris platform (www.FreePascal.org), the hassle of having it installed on the laboratory machines and also the fact that it is an immature environment based on open source contributions, resulted in the decision of having the Object Pascal code translated in ANSI C code that could be compiled using the GNU C compiler (i.e. gcc). The ANSI C code has also been compiled using the free Borland C compiler 5.5 for Windows, for testing purposes.

5 Search methodology

The approach taken to explore the search space. Must include within the discussion the rationale behind the engineering design decision which were made.

As stated in the introduction, the initial search technique was limited to a simple 1-ply search with a simple heuristic. Subsequently, the search was evolved gradually during the first 20 versions towards a more efficient search based on the adaptation of the classic MiniMax algorithm with Alpha-Beta pruning. After the first implementation of the Alpha-Beta pruning, a few versions of the AI engine have been dedicated to the debugging of the algorithm whose implementation turned out to be not trivial. One example that illustrates a bug in the search algorithm is shown in Figure 5 where, the red player AI engine, although it was its turn to move, was not going for the move in column 2 leading to a sure win for the red because it would think that in this way it may help the black player win with a subsequent move on the same column. The search algorithm and the heuristic calculation must have included information that would help the search algorithm stop after a win situation.

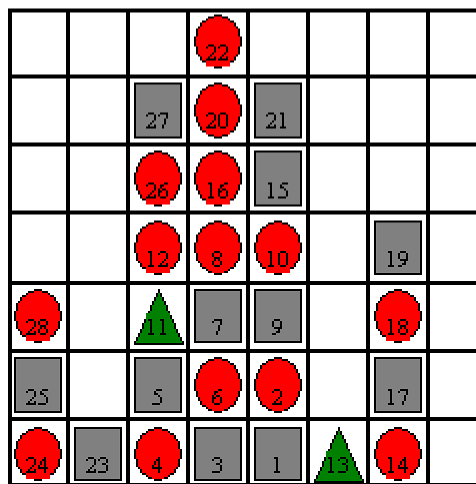

Figure 5. Example of an implementation error in the search algorithm. The red player (circles) will not go for the obvious winning move in column 2

As far as the search depth is concerned, the first 20 versions of the AI engine, could only search a 2 ½ plies deep game space (i.e. 5 moves), due to the computationally expensive heuristic function employed for evaluation. Starting with version 21, by employing the board map data structure, by searching the columns in the descending order of their importance and by simplifying the heuristic function, the search depth has been extended with another ply (i.e. 3 ½ plies or 7 moves). The board map and the column sorting routines are performed once, after the game board creation and parsing by calling the BuildBoardMap() function. This function creates the board maps, scans the 2C2 groups on the current board received form the game controller, initializes the 2C2 groups records and then indexes the columns in the descending order of the number of 2C2 groups containing a free square in those columns. While the creation of the board map has the obvious reason to provide fast access to the 2C2 group records that contain a particular square, the improvement brought by sorting of the columns by the number 2C2 groups containing a free square in that column had to be tested in order to be proven useful. The test was done simply by counting the number of nodes opened by the search routine with and without the column sorting. For a 7 row 8 column empty board, the number of 2C2 groups containing a free square in a column are the following: 31, 46, 61, 76,76, 61, 46, 31. After sorting, the obtained columns index 4,5,3,6,2,7,1,8 will enable the search of the 4th and 5th columns first as they

contain free squares which are part of most 2C2 groups (i.e. 76 2C2 groups, on an empty board). The assumption is supported by the testing results in the following table.

Move number	Without sorting		With sorting	
	Nodes expanded	Move time (sec)	Nodes expanded	Move time (sec)
1	3,567,757	51.69	519,924	8.43
3	1,025,607	13.63	1,088,025	16.29
5	1,432,204	17.81	295,918	4.09

Table 2. The number of nodes expended and move time, with and without column sorting

The most current search routines are based on five functions:

```
void AlphaBeta(void)
float EvaluateMaxNodeChildren(float Alpha, float Beta)
float EvaluateMinNodeChildren(float Alpha, float Beta)
float DoMove(int APlayer, int ACol, int APiece)
void UndoLastMove(void)
```

The AlphaBeta() function provides the starting point of the search procedure. The main "for" loop is executed ColCount times and contains a red piece move and a green piece move. The way the moves are executed is by calling the function DoMove() with the appropriate parameters, which returns the estimated value of the heuristic function for that move. For example, for a red piece move the following code is executed:

```
HeuristicValue=DoMove(gpRed,ColumnSortIdx[jc],gpRed);
if ((HeuristicValue<INFINITY)&&(MoveCount<MAX_MOVES))
      HeuristicValue=EvaluateMinNodeChildren(AlphaRoot,INFINITY);
if (HeuristicValue>AlphaRoot)
{
  AlphaRoot=HeuristicValue;
  BestMoveColIdx=ColumnSortIdx[jc];
  BestMovePiece=gpRed;
};
UndoLastMove();
if (AlphaRoot>=INFINITY) break;
```

The checks before calling the EvaluateMinNodeChildren() function are necessary in order to stop the search in the case of an imminent win situation (i.e. HeuristicValue>=INFINITY). In the case of the green piece move, a special check for an opponent win situation (HeuristicValue<=-INFINITY) is also performed:

```
HeuristicValue=DoMove(gpRed,ColumnSortIdx[jc],gpGreen);
if ((HeuristicValue>-INFINITY)
  &&(HeuristicValue<INFINITY)
  &&(MoveCount<MAX_MOVES))
HeuristicValue=EvaluateMinNodeChildren(AlphaRoot,INFINITY);
if (HeuristicValue>AlphaRoot)
{
  AlphaRoot=HeuristicValue;
  BestMoveColIdx=ColumnSortIdx[jc];
  BestMovePiece=gpGreen;
}
UndoLastMove();
if (AlphaRoot>=INFINITY) break;
```

Further, if the EvaluateMinNodeChildren() and EvaluateMaxNodeChildren() functions are called they will also execute a loop and search for possible moves until the condition

```
if (Alpha>=Beta) break;
```

is met or the current search depth equals the value of the maximum specified ply depth.
The current search depth of 3 ½ plies starts with a Max move and ends also with a Max move, this accounting for 7 moves.

6 Game heuristics

A detailed description of the heuristics which were employed including issues such as the admissibility of the heuristic (if applicable). Why were the heuristics chosen? etc. Each employed heuristic should have its own subsection discussing it.

During the first 20 versions of the AI engine, the search space-heuristic tradeoff was biased towards a better informed heuristic. Given the results in the tournament, starting with version 21, the design was modified so the AI engine was able to perform a deeper search but employing a less computationally expensive heuristic.

All the heuristic functions employed were based on a basic decision structure built on the information about the 2C2 groups, encoded in the group record structures. This algorithm basically assigns the players scores based on the count information stored in each group record. For each group record, the following decisions are performed:

- if a 2C2 group has:
  - more than 3 pieces of a kind, but not blanks ,
  - more than a black piece and more than a red piece,

that group contains a non winning combination that has no influence on any of the scores

- if a 2C2 group has:
  - one or two red pieces
  - one or two red pieces and one green piece

that group has a potential for winning for the red player and the red player score increases with a value proportional with the number of free squares in that group

- if a 2C2 group has:
  - one or two black pieces
  - one or two black pieces and one green piece

that group has a potential for winning for the black player and the black player score increases with a value proportional with the number of free squares in that group

- if a 2C2 group has one or two green pieces that group has a potential for winning for the both players and the players' scores increase with values proportional with the number of free squares in that group
- if a 2C2 group has two red pieces and two green pieces, the red player wins and the red player score becomes INFINITY
- if a 2C2 group has two black pieces and two green pieces, the black player wins and the black player score becomes INFINITY

The final value returned by the heuristic was a real value obtained by subtracting the black player score from the red player score. Evidently, the MiniMax algorithm with Alpha-Beta pruning would try to find a move that maximizes the value of this heuristic. This basic heuristic captures information and

assigns scores based on the 2C2 group content information without considering the location of the groups.

6.1 The “space gap” heuristic

One of the first improvements of the heuristic calculation was based on the insight that the 2C2 groups located at the top of the board are less important than those located at the bottom. However, as the game progresses, the 2C2 groups at the top of the board become increasingly important and this should reflect in the heuristic calculation. Hence, the concept of “space gap” for each 2C2 group was introduced and the GroupRec data structure modified accordingly. For example, in Figure 6 (left), the horizontal 2C2 groups have space gap values of 2 (red) and 5 (black) while the diagonal 2C2 groups have a space gap value of 4 (red) and 2 (black). However, since the red player started the second and has the next move (i.e. #36) this will result in a win situation for the red player in the horizontal 2C2 group on the third row, group that has the lowest space gap value.

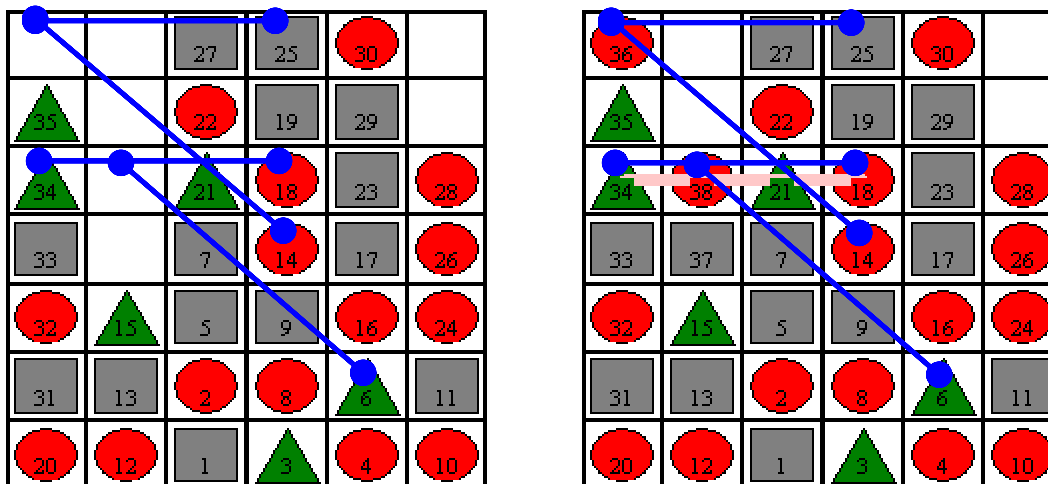

Figure 6. Although they are all potentially winning situations, the 2C2 groups have different weights in the heuristic calculation which cause the red player to win; the “space gap” values of each group capture the information needed to assign the proper weighting

Capturing the space gap information in the heuristic calculation is computationally expensive as one has to iterate not only through all the four squares of each 2C2 group that contains the last moved piece, but also through all the 2C2 groups that have a square located in the column where the last move was made, in order to accurately decrement the space gap values of the affected 2C2 groups. In addition, the calculation of the space gaps is dependent on the direction of the 2C2 groups.

A particular attention has been paid to the column (i.e. vertical) 2C2 groups space gap information use in the heuristic calculation. Because of the 2-Connect-2 game specification (i.e. the gravity factor), the space gap value of column 2C2 groups with winning potential contribute to the calculation of the heuristic in an undesired way which causes the AI engine to attempt to complete these groups as in this way their space gap value would decrease. This will however not cause any winning situation, except in very specific instances where the opponent, by completing the column 2C2 group in order to counteracting the win situation, will contribute to decreasing the space gap of another 2C2 group to a value of 1, which represents a sure win situation. As a slight advantage, by constructing these column 2C2 groups, the AI will occupy board space and proceed with an “offensive” approach which increases its chances to win.

6.2 The "odd-even hot square" heuristic

In the example given in Figure 6, a contribution to the win can also be attributed to the particular player order and also to another factor related to the game board size. For this, we defined the concepts of a hot square, cold square, even hot square, odd hot square, even board and odd board (see definitions section). In addition, we distinguish between the players order by defining the first player having odd move numbers (i.e. 1, 3, 5, etc) and the second player having the even move numbers (i.e. 2, 4, 6 etc). Under the assumption that a game progression did not encounter a win situation, the development of the heuristics described in this section were initially based on the following simple insights:

- on a even board, the second player will always have the last move
- on a odd board, the situation reverses and the first player gets the last move

By generalizing, it has also been recognized that:

- on a even board, if the second player had a even hot square, that hot square is a win hot square or a tie hot square as the first player will be forced to make a move on the cold square associated with the even hot square
- on a even board, if the first player had a odd hot square, that hot square is a win hot square or a tie hot square as the second player will be forced to make a move on the cold square associated with the odd hot square
- on a odd board, the situation reverses and the first player can only turn into win hot squares the even hot squares

For example, in Figure 7, the second player (black rectangles) has two even hot squares and although the game could have been continued a few moves beyond move #14, the red player recognized that the game was lost.

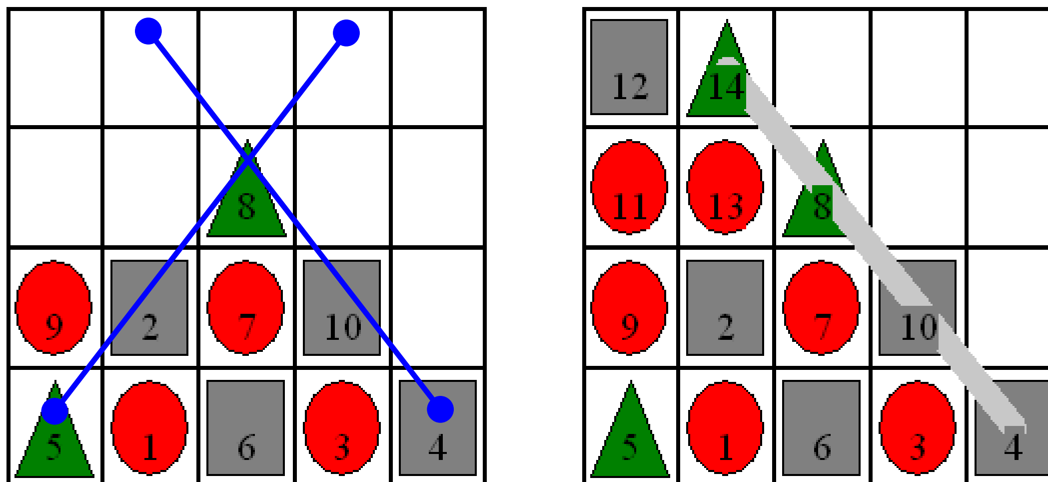

Figure 7. The second player (black) has two even hot squares located in the top row one of which became a win hot square in move #14

Similarly, if the black player is the first player, only a odd hot square could become a win hot square, as in Figure 8.

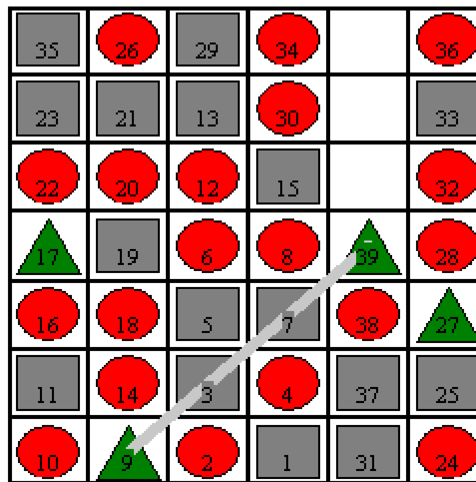

Figure 8. On an even board (7 rows, 6 columns) an odd hot square located in the 5th column, becomes a win hot square for the first player (black)

The situation could be further complicated by the existence of multiple hot squares, on different rows and columns. Obviously, the hot squares with a lower space gap override the ones with higher space gap values. It can be shown and this situation is already depicted in Figure 7, while on a even board:

- for the second player, if the sum of the hot square rows is even and there are no other hot square with lower space gap values, one of those hot squares is a win hot square or a tie hot square
- for the first player, if the sum of the hot square rows is odd and there are no other hot square with lower space gap values, one of those hot squares is a win hot square or a tie hot square
- on a odd board, the situation reverses

In addition, the interaction between the existence of multiple hot squares from both players complicates the analysis even further and suggest that the implementation of a heuristic function based on these insights is going to be prohibitive computationally. Keeping track not only of the hot squares disposition for each player but also of the interaction and overriding entails use of appropriate data structures and processor power with impact on the search depth. This has already bee proved to be a relatively good but not enough powerful approach to the 2-Connect-2 game playing, where, because of the existence of the green neutral game pieces, searching depth seems to have priority over the heuristic sophistication. As a practical approach, using these insights, one could gear the heuristic search toward favoring the hot squares which are located in rows which match a player parity. This was done by calculating, for each move, the value of a variable called PlayerSituation and which encodes all theinformation about the player order (i.e. MoveCount%2) and the current board type (i.e. odd or even, given by MAX_MOVES%2, where MAX_MOVES=ColCount*RowCount).

```
PlayerSituation=(MoveCount%2+MAX_MOVES%2)%2;
```

If the player situation value matches the parity of a hot square then that hot square contribution to the heuristic function is going to be increased by a constant.

6.3 The “double hot square” situation

It is easy to imagine that a situation when one of the players has 2 consecutive hot squares (double hot square) could lead to a sure win. In a game between rational opponents, one way to arrive at this situation is by making a single move which implies that two 2C2 groups containing the hot squares are crossing each other, as in Figure, i.e. they have at least a common square.

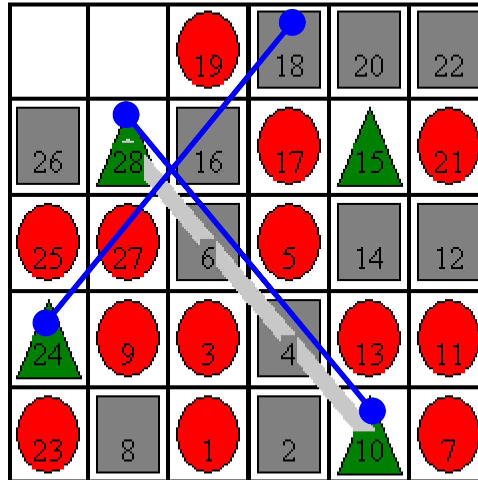

Figure 9. The black player had a double hot square win situation

This situation is also restricted by the fact that if one of the two 2C2 groups has a vertical direction, its hot square must have a lower space gap then the space gap of the adjacent hot square. This has been implemented with the hope that the AI engine would maximize the number of adjacent hot squares, however, it has proven to also be an expensive heuristic which required hot square information at the board level, not only in the neighborhood of the last move.

6.4 The defensive/offensive heuristic

This approach was based on the idea that the first player has a slight advantage and should play a more offensive game while the second player should play a more defensive game. The offensive/defensive strategy could be implemented easily by increasing the weight of the first player in the calculation of the heuristic function. This causes the second player to use less neutral pieces as they will increase the advantage of the first player and implicitly will decrease the final value of the heuristic.

6.5 The current version of the heuristic

As stated before, the heuristic underwent 22 consecutive changes, some of them major. As of version 21, the search has been made more efficient at the expense of the sophistication of the heuristic function. However, while the work on improving the search was significant, the search itself has only been extended with another ply and timing issues have started to appear. In addition, the testing against the older versions of the AI engine shows that the new version it is being consistently outperformed. This could be related to bugs in the approach but also to the heuristic calculation which now lacks information at the board level, as it only scans the neighborhood of the last move.

The parameters that the current heuristic is operating with are the following:

```
float HOT_SQUARE_VALUE=1000.0;
float SPACE_GAP_ZOOM=10.0;
float TERITORY_VALUE=100000.0;
float FIRST_DEFENSIVE_STRATEGY=1.0;
float SECOND_DEFENSIVE_STRATEGY=1.0;
int PLY_DEPTH=3;
```

Although some of them are self explanatory (i.e. HOT_SQUARE_VALUE and PLY_DEPTH) the other must be explained. The space gap zoom attempts to achieve a better differentiation between the different values that the space gap may have, by multiplying them with that constant value. The territory value is actually the constant that contributes to the calculation of the values associated with a particular space gap as for example:

```
RedScore+=TERITORY_VALUE/(1.0+SPACE_GAP_ZOOM*((float)Group->SpaceGap-1.0));
```

The first and second defensive strategy are used to setup a different weighting of the red player and black player scores into the final heuristic value.

7 Improvements

The first improvement of the AI engine is related to the timing issues that appeared consecutive to extending of the search space. The implementation of a solution to this problem is envisioned as making use of code testing on the Sun Solaris machines, testing that would give a rough estimation of what are the computational limitations, expressed in an upper bound on the number of search nodes that can be expanded in 20 seconds, which is the theoretical limit for making a move, given a game limit of 20 minutes on a board of 7 rows and 8 columns. The development of a more sophisticated heuristic, using some of the ideas described in the previous section, while keeping an efficient search of at least 3 plies is a second priority.

Based on the experience with the development of the 25 versions of the AI engine, one could make a fair assumption that the improvements that this AI engine is a process that could benefit from an approach that would help fine-tuning the heuristic parameters in order to achieve good results. This will involve probably a self-learning approach (i.e. artificial neural network) which would be given the task to find out the proper weighting of the constant parameters involved in the heuristic function calculation. This could be achieved by having the AI engine playing against the other versions, while trying to maximize its wins. This approach entails a whole set of issues such as the design, training and testing of the artificial neural network. These are tasks that require a significant time investment.

8 Known Bugs

From the beginning, the design and implementation approach for the AI engine was hindered by portability issues which caused some limitations in the efficiency of the C code. However, the game controller available for the Windows platform made it easy to test and debug new versions of the engine which currently does not suffer from bugs in its code, but rather from the design approach that proved unsuccessful in the long run and which has had to be changed recently in order to improve the overall performance.

9 Discussion & Conclusions

A significant amount of work in the design, implementation and testing of the AI engine was consumed. The effort was also increased by the portability issues which have been addressed by translating the Object Pascal code into ANSI C. The development of a sophisticated heuristic proved very challenging and in the same time enhanced the appreciation of the complexity of combinatorial problems. The implementation of the Minimax algorithm with alpha beta pruning, while straightforward in the text books turned out to be not trivial when applied to a real game. The provision of the 2Connect-2 tournament was also invaluable for the testing and development of the AI engine and

made the whole process of learning not only challenging but exciting. Finally, the understanding that in order to improve the engine performance, an adaptive approach in the form of an artificial neural network may be desirable, is a piece of knowledge that no teacher would be able to explain.

12. Source Code must be attached as an appendix to the report. Additionally, the directory and file names for the source code and executables must be included and have the permissions set such that it is accessible for copying. Source code/executables may be compared to ensure they are not substantial copies of other groups work.

10 References

1. http://www.ece.uvic.ca/~sneville/ceng420/Project.htm. CENG 420 - Course Project 2-Connect-2 accessed July 25, 2003.

11 Appendices

```
#include <stdio.h>
#include <stdlib.h>

#define Max(A,B) ((A) > (B) ? (A) : (B));
#define Min(A,B) ((A) < (B) ? (A) : (B));

//some functions prototypes
float EvaluateMinNodeChildren(float Alpha, float Beta);
float DoMove(int APlayer, int ACol, int APiece);
void UndoLastMove(void);
//game parameters
  float INFINITY=10000000000000.0;
//  float ODD_EVEN_2C2_GROUP_VALUE=1000.0;
  float HOT_SQUARE_VALUE=1000.0;
  float SPACE_GAP_ZOOM=10.0;
  float TERITORY_VALUE=100000.0;
  float FIRST_DEFENSIVE_STRATEGY=1.0;
  float SECOND_DEFENSIVE_STRATEGY=1.0;
  int PLY_DEPTH=3;
/*****************************************************************************\
* Global Types/Structs
\*****************************************************************************/
//board directions consts
  int wdRow=0, wdColumn=1, wdFirstDiagonal=2, wdSecondDiagonal=3;
//game pieces consts
  int gpRed=0, gpBlack=1, gpGreen=2, gpSpace=3;
  char PIECE_CHAR[4]={'r','b','g','s'};
//the last move column is 0-based
  typedef struct TGroupRecType
  {
    int Count[4];
//score for moves away from winning
    int SpaceGap;
    int P1Row;
    int P1Col;
    int Direction;
    int *P[4];
  } GroupRec;

/*****************************************************************************\
* Global Variables                                                            *
\*****************************************************************************/
  int RowCount, ColCount, LastColMoveIdx, LastRowMoveIdx;
//board rows, columns - all 0-based arrays
  int **Board;
//the map with the 2c2 groups associated with each square on the board
  GroupRec ****BoardMap;
  int **BoardMapCount;
  int MAX_MOVES, MoveCount;
//the row of the new moves
  int *ColumnSortIdx;
  int *ColumnSortCounts;
  int *NextRowIdx;
//auxiliary board info
//  RowGroup=array[1..ROW_COUNT][1..COL_COUNT-3] of GroupRec;
//  ColGroup=array[1..ROW_COUNT-3][1..COL_COUNT] of GroupRec;
```

```
//  DiagGroup=array[1..ROW_COUNT-3][1..COL_COUNT-3] of GroupRec;
  GroupRec **RowGroup, **ColGroup, **FirstDiagGroup, **SecDiagGroup;
  float RedScore=0, BlackScore=0;
  float DefensiveStrategy;
//AI stuff
int SaveColMoveIdx[100];
int SaveRowMoveIdx[100];
int SavePointer=0;
int MaxCol;
char MaxPiece;
float TheMaxValue,TheMinValue,HeuristicValue;
int NodesVisited;
int CurrentDepth;

void ErrLog(char * aLine)
{
  fprintf(stderr,aLine);
}

void CreateBoard(void)
{
      int i, j;
  char PieceChar;
  scanf("(%d,%d,%d", &ColCount, &RowCount, &LastColMoveIdx);
//  sscanf(TEST_BOARD1,"(%d,%d,%d", &ColCount, &RowCount, &LastColMoveIdx);
//  puts(TEST_BOARD1);
  if ((RowCount<4)||(ColCount<4)) ErrLog("Invalid board sizes.");
  if ((LastColMoveIdx<0)||(LastColMoveIdx>ColCount)) ErrLog("Invalid last move.");
//from now on, zero based col move, it is an index not a count
  LastColMoveIdx--;
//allocating the rows
  Board=(int **) calloc(RowCount,sizeof(int *));
//allocating the board map rows
  BoardMapCount=(int **) calloc(RowCount,sizeof(int *));
  BoardMap=(GroupRec ****) calloc(RowCount,sizeof(GroupRec ***));
//allocating the columns
      for (i = 0; i < RowCount; i++)
  {
    Board[i]=(int *) calloc(ColCount,sizeof(int));
    BoardMapCount[i]=(int *) calloc(ColCount,sizeof(int));
    BoardMap[i]=(GroupRec ***) calloc(ColCount,sizeof(GroupRec **));
//there can be at most 16 2c2 groups that each square on the board may be part of
      for (j = 0; j < ColCount; j++) BoardMap[i][j]=(GroupRec **)
calloc(16,sizeof(GroupRec *));
  }
//allocating the current row indexes
  NextRowIdx=(int *) calloc(ColCount,sizeof(int));
  ColumnSortIdx=(int *) calloc(ColCount,sizeof(int));
  ColumnSortCounts=(int *) calloc(ColCount,sizeof(int));
  for (j = 0; j < ColCount; j++)
  {
    NextRowIdx[j]=-1;
    ColumnSortIdx[j]=j;
  }
  MAX_MOVES=RowCount*ColCount;
  MoveCount=0;
      for (i = 0; i < RowCount; i++)
            for (j = 0; j < ColCount; j++)
    {
```

```
                  scanf(",%c", &PieceChar);
//          PieceChar=TEST_BOARD1[5+2*i*ColCount+2*(j+1)];
//          printf("%c",PieceChar);
          switch (PieceChar)
          {
            case 'r': Board[i][j]=gpRed; MoveCount++;  break;
            case 'b': Board[i][j]=gpBlack; MoveCount++; break;
            case 'g': Board[i][j]=gpGreen; MoveCount++; break;
            case 's': Board[i][j]=gpSpace; NextRowIdx[j]=i; break;
          }
      }
  if (MoveCount>=MAX_MOVES) ErrLog("Full board.");
  if (LastColMoveIdx>-1)
  {
//calculating the last move row index; NextRowIdx is a 0-based array
    LastRowMoveIdx=NextRowIdx[LastColMoveIdx]+1;
    if (LastRowMoveIdx>RowCount-1) ErrLog("Invalid last move.");
//    else fprintf(stderr,"Last move:
(%d,%c)",LastColMoveIdx+1,PIECE_CHAR[Board[LastRowMoveIdx][LastColMoveIdx]]);
  }
//init the additional Group structures
//rowGroup
  RowGroup=(GroupRec **) calloc(RowCount,sizeof(GroupRec *));
      for (i = 0; i < RowCount; i++) RowGroup[i]=(GroupRec *) calloc(ColCount-
3,sizeof(GroupRec));
//colGroup
  ColGroup=(GroupRec **) calloc(RowCount-3,sizeof(GroupRec *));
      for (i = 0; i < RowCount-3; i++) ColGroup[i]=(GroupRec *)
calloc(ColCount,sizeof(GroupRec));
//firtsdiagGroup
  FirstDiagGroup=(GroupRec **) calloc(RowCount-3,sizeof(GroupRec *));
      for (i = 0; i < RowCount-3; i++) FirstDiagGroup[i]=(GroupRec *)
calloc(ColCount-3,sizeof(GroupRec));
//secdiagGroup
  SecDiagGroup=(GroupRec **) calloc(RowCount-3,sizeof(GroupRec *));
      for (i = 0; i < RowCount-3; i++) SecDiagGroup[i]=(GroupRec *)
calloc(ColCount-3,sizeof(GroupRec));
}

/*****************************************************************************\
* Destroy
\*****************************************************************************/
void DestroyBoard(void)
{
//board
  int i,j;
      for (i=0;i<RowCount;i++)
  {
    free(Board[i]);
    free(BoardMapCount[i]);
      for (j=0;j<ColCount;j++) free(BoardMap[i][j]);
    free(BoardMap[i]);
  }
  free(Board);
  free(BoardMapCount);
  free(BoardMap);
//current row indexes array
  free(NextRowIdx);
  free(ColumnSortCounts);
```

```
  free(ColumnSortIdx);
//rowGroup
      for (i = 0; i < RowCount; i++) free(RowGroup[i]);
  free(RowGroup);
//colGroup
      for (i = 0; i < RowCount-3; i++) free(ColGroup[i]);
  free(ColGroup);
//firstdiagGroup
      for (i = 0; i < RowCount-3; i++) free(FirstDiagGroup[i]);
  free(FirstDiagGroup);
//secdiagGroup
      for (i = 0; i < RowCount-3; i++) free(SecDiagGroup[i]);
  free(SecDiagGroup);
}

//////////////////////////////////////////////////////
//  ROW 2c2 group scan
//////////////////////////////////////////////////////
void ScanRow(int ARowIdx, int Start2C2GroupColIdx, int End2C2GroupColIdx)
{
  int i,j,k;
  for (i=Start2C2GroupColIdx;i<=End2C2GroupColIdx; i++)
  {
    for (k=gpRed; k<=gpSpace; k++) RowGroup[ARowIdx][i].Count[k]=0;
    RowGroup[ARowIdx][i].Direction=wdRow;
    RowGroup[ARowIdx][i].P1Row=ARowIdx;
    RowGroup[ARowIdx][i].P1Col=i;
    for (j=0; j<=3; j++)
    {
      RowGroup[ARowIdx][i].Count[Board[ARowIdx][i+j]]++;
      RowGroup[ARowIdx][i].P[j]=&Board[ARowIdx][i+j];
    }
  }
}
//////////////////////////////////////////////////////
//  COL 2c2 group scan
//////////////////////////////////////////////////////
void ScanCol(int AColIdx, int Start2C2GroupRowIdx, int End2C2GroupRowIdx)
{
  int i,j,k;
  for (i=Start2C2GroupRowIdx; i<=End2C2GroupRowIdx; i++)
  {
    for (k=gpRed; k<=gpSpace; k++) ColGroup[i][AColIdx].Count[k]=0;
    ColGroup[i][AColIdx].Direction=wdColumn;
    ColGroup[i][AColIdx].P1Row=i;
    ColGroup[i][AColIdx].P1Col=AColIdx;
    for (j=0; j<=3; j++)
    {
      ColGroup[i][AColIdx].Count[Board[i+j][AColIdx]]++;
      ColGroup[i][AColIdx].P[j]=&Board[i+j][AColIdx];
    }
  }
}
//////////////////////////////////////////////////////
//  First diagonal 2c2 group scan
//////////////////////////////////////////////////////
void ScanFirstDiag(int Start2C2GroupRowIdx, int Start2C2GroupColIdx, int
A2C2GroupCount)
{
```

```
  int i,j,k;
  for (i=0; i<A2C2GroupCount; i++)
  {
    for (k=gpRed; k<=gpSpace; k++)
FirstDiagGroup[Start2C2GroupRowIdx+i][Start2C2GroupColIdx+i].Count[k]=0;

FirstDiagGroup[Start2C2GroupRowIdx+i][Start2C2GroupColIdx+i].Direction=wdFirstDiago
nal;

FirstDiagGroup[Start2C2GroupRowIdx+i][Start2C2GroupColIdx+i].P1Row=Start2C2GroupRow
Idx+i;

FirstDiagGroup[Start2C2GroupRowIdx+i][Start2C2GroupColIdx+i].P1Col=Start2C2GroupCol
Idx+i;
    for (j=0; j<=3; j++)
    {

FirstDiagGroup[Start2C2GroupRowIdx+i][Start2C2GroupColIdx+i].Count[Board[Start2C2Gr
oupRowIdx+i+j][Start2C2GroupColIdx+i+j]]++;

FirstDiagGroup[Start2C2GroupRowIdx+i][Start2C2GroupColIdx+i].P[j]=&Board[Start2C2Gr
oupRowIdx+i+j][Start2C2GroupColIdx+i+j];
    }
  }
}

//////////////////////////////////////////////////////
//  Second diagonal procs
//////////////////////////////////////////////////////
void ScanSecDiag(int Start2C2GroupRowIdx, int Start2C2GroupColIdx, int
A2C2GroupCount)
{
  int i,j,k;
//here is tricky
  for (i=0; i<A2C2GroupCount; i++)
  {
    for (k=gpRed; k<=gpSpace; k++)
SecDiagGroup[Start2C2GroupRowIdx+i][Start2C2GroupColIdx+i].Count[k]=0;

SecDiagGroup[Start2C2GroupRowIdx+i][Start2C2GroupColIdx+i].Direction=wdSecondDiagon
al;
    SecDiagGroup[Start2C2GroupRowIdx+i][Start2C2GroupColIdx+i].P1Row=RowCount-1-
Start2C2GroupRowIdx-i;

SecDiagGroup[Start2C2GroupRowIdx+i][Start2C2GroupColIdx+i].P1Col=Start2C2GroupColId
x+i;
    for (j=0; j<=3; j++)
    {

SecDiagGroup[Start2C2GroupRowIdx+i][Start2C2GroupColIdx+i].Count[Board[RowCount-1-
Start2C2GroupRowIdx-i-j][Start2C2GroupColIdx+i+j]]++;

SecDiagGroup[Start2C2GroupRowIdx+i][Start2C2GroupColIdx+i].P[j]=&Board[RowCount-1-
Start2C2GroupRowIdx-i-j][Start2C2GroupColIdx+i+j];
    }
  }
}

void BuildBoardMap(void)
```

```
{
  int TempValue,i,j,k;
  int Start2C2GroupRowIdx,
  End2C2GroupRowIdx,
  Start2C2GroupColIdx,
  End2C2GroupColIdx,
  TopLeftIdx,
  The2C2GroupCount;
//this is done only ONCE
  for (j = 0; j <ColCount; j++)
  {
    ColumnSortCounts[j]=0;
      for (i = 0; i < RowCount; i++)
    {
//assume you move at i,j; which 2c2 groups are affected by this move?
//procedure LastMoveScanRow(ARowMove,AColMove:Integer);
      Start2C2GroupColIdx=Max(0,j-3);
      End2C2GroupColIdx=Min(j,ColCount-4);
      The2C2GroupCount=End2C2GroupColIdx-Start2C2GroupColIdx+1;
      BoardMapCount[i][j]=The2C2GroupCount;
      for (k=0; k<The2C2GroupCount; k++)
      {
        BoardMap[i][j][k]=&RowGroup[i][Start2C2GroupColIdx+k];
        ScanRow(i,Start2C2GroupColIdx,End2C2GroupColIdx);
        if (Board[i][j]==gpSpace) ColumnSortCounts[j]++;
      }
//procedure LastMoveScanCol(ARowMove,AColMove:Integer);
      Start2C2GroupRowIdx=Max(0,i-3);
      End2C2GroupRowIdx=Min(i,RowCount-4);
      The2C2GroupCount=End2C2GroupRowIdx-Start2C2GroupRowIdx+1;
      for (k=0; k<The2C2GroupCount; k++)
      {
        BoardMap[i][j][BoardMapCount[i][j]]=&ColGroup[Start2C2GroupRowIdx+k][j];
        BoardMapCount[i][j]++;
        ScanCol(j,Start2C2GroupRowIdx,End2C2GroupRowIdx);
        if (Board[i][j]==gpSpace) ColumnSortCounts[j]++;
      }
//procedure LastMoveScanFirstDiag(ARowMoveIdx,AColMoveIdx:Integer);
      TopLeftIdx=Min(i,j);
      Start2C2GroupColIdx=Max(j-TopLeftIdx,j-3);
      Start2C2GroupRowIdx=Max(i-TopLeftIdx,i-3);
      End2C2GroupColIdx=Min(j,ColCount-4);
      End2C2GroupRowIdx=Min(i,RowCount-4);
      The2C2GroupCount=Min(End2C2GroupColIdx-Start2C2GroupColIdx,End2C2GroupRowIdx-
Start2C2GroupRowIdx);
      The2C2GroupCount++;
      if (The2C2GroupCount>0)
      {
        for (k=0; k<The2C2GroupCount; k++)
        {

BoardMap[i][j][BoardMapCount[i][j]]=&FirstDiagGroup[Start2C2GroupRowIdx+k][Start2C2
GroupColIdx+k];
          BoardMapCount[i][j]++;
          ScanFirstDiag(Start2C2GroupRowIdx,Start2C2GroupColIdx,The2C2GroupCount);
          if (Board[i][j]==gpSpace) ColumnSortCounts[j]++;
        }
      }
//procedure LastMoveScanSecDiag(ARowMoveIdx,AColMoveIdx:Integer);
```

```
      TopLeftIdx=Min(RowCount-1-i,j);
      Start2C2GroupColIdx=Max(j-TopLeftIdx,j-3);
      Start2C2GroupRowIdx=Max(RowCount-1-i-TopLeftIdx,RowCount-1-i-3);
      End2C2GroupColIdx=Min(j,ColCount-4);
      End2C2GroupRowIdx=Min(RowCount-1-i,RowCount-4);
      The2C2GroupCount=Min(End2C2GroupColIdx-Start2C2GroupColIdx,End2C2GroupRowIdx-
Start2C2GroupRowIdx);
      The2C2GroupCount++;
      if (The2C2GroupCount>0)
      {
        for (k=0; k<The2C2GroupCount; k++)
        {

BoardMap[i][j][BoardMapCount[i][j]]=&SecDiagGroup[Start2C2GroupRowIdx+k][Start2C2Gr
oupColIdx+k];
          BoardMapCount[i][j]++;
          ScanSecDiag(Start2C2GroupRowIdx,Start2C2GroupColIdx,The2C2GroupCount);
          if (Board[i][j]==gpSpace) ColumnSortCounts[j]++;
        }
      }
    }
  }
//bubble sorting sorting the columns based on ColumnSortCounts[j]
//the columns with least pieces are the most important
  for (j = 0; j<ColCount-1; j++)
  {
    for (k=0; k<ColCount-1-j; k++)
    {
      if (ColumnSortCounts[k]<ColumnSortCounts[k+1])
      {
//swaping the values the values
        TempValue=ColumnSortCounts[k];
        ColumnSortCounts[k]=ColumnSortCounts[k+1];
        ColumnSortCounts[k+1]=TempValue;
        TempValue=ColumnSortIdx[k];
        ColumnSortIdx[k]=ColumnSortIdx[k+1];
        ColumnSortIdx[k+1]=TempValue;
      }
    }
  }
}

//////////////////////////////////////////-----------------------------------------
---------------------------
///////////ALPHA BETA
//////////////////////////////////////////////
//the move column and move piece define the node
float EvaluateMaxNodeChildren(float Alpha, float Beta)
{
  float HeuristicValue;
  int jc=0;
  CurrentDepth++;
//for all children of this node do the evaluate MIN
  while (jc<ColCount)
  {
//we check wether the move is valid
    if (NextRowIdx[ColumnSortIdx[jc]]>=0)
    {
//get the max value of the red child
```

```
      HeuristicValue=DoMove(gpRed,ColumnSortIdx[jc],gpRed);
      if
((HeuristicValue<INFINITY)&&(CurrentDepth<PLY_DEPTH)&&(MoveCount<MAX_MOVES))
HeuristicValue=EvaluateMinNodeChildren(Alpha,Beta);
      if (HeuristicValue>Alpha) Alpha=HeuristicValue;
      UndoLastMove();
      if (Alpha>=Beta) break;
//get the max value of the green child
      HeuristicValue=DoMove(gpRed,ColumnSortIdx[jc],gpGreen);
      if
((HeuristicValue<INFINITY)&&(CurrentDepth<PLY_DEPTH)&&(MoveCount<MAX_MOVES))
HeuristicValue=EvaluateMinNodeChildren(Alpha,Beta);
      if (HeuristicValue>Alpha) Alpha=HeuristicValue;
      UndoLastMove();
      if (Alpha>=Beta) break;
    }
    jc++;
  }
  CurrentDepth--;
  return Alpha;
}

//the move column and move piece define the node
float EvaluateMinNodeChildren(float Alpha, float Beta)
{
  float HeuristicValue;
//for all children of this node do the evaluate MAX
  int jc=0;
  while (jc<ColCount)
  {
//we check wether the move is valid
    if (NextRowIdx[ColumnSortIdx[jc]]>=0)
    {
//we get the min value of the black child
      HeuristicValue=DoMove(gpBlack,ColumnSortIdx[jc],gpBlack);
      if ((HeuristicValue>-INFINITY)&&(MoveCount<MAX_MOVES))
HeuristicValue=EvaluateMaxNodeChildren(Alpha,Beta);
      if (HeuristicValue<Beta) Beta=HeuristicValue;
      UndoLastMove();
      if (Alpha>=Beta) break;
//we get the min value of the green child
      HeuristicValue=DoMove(gpBlack,ColumnSortIdx[jc],gpGreen);
      if ((HeuristicValue>-INFINITY)&&(MoveCount<MAX_MOVES))
HeuristicValue=EvaluateMaxNodeChildren(Alpha,Beta);
      if (HeuristicValue<Beta) Beta=HeuristicValue;
      UndoLastMove();
      if (Alpha>=Beta) break;
    }
    jc++;
  }
  return Beta;
}

void AlphaBeta(void)
{
  int BestMoveColIdx=-1;
  int BestMovePiece=gpRed;
  float HeuristicValue, AlphaRoot;
  int jc;
```

```
  if (MoveCount%2==0) DefensiveStrategy=FIRST_DEFENSIVE_STRATEGY;
  else DefensiveStrategy=SECOND_DEFENSIVE_STRATEGY;
  SavePointer=0;
  CurrentDepth=0;
  AlphaRoot=-INFINITY;
//this is the root node and has 2 MIN children
//we need to get the maximum value of theirs
  for (jc=0;jc<ColCount;jc++)
  {
    if (NextRowIdx[ColumnSortIdx[jc]]>=0)
    {
///checking the red
      HeuristicValue=DoMove(gpRed,ColumnSortIdx[jc],gpRed);
      if ((HeuristicValue<INFINITY)&&(MoveCount<MAX_MOVES))
HeuristicValue=EvaluateMinNodeChildren(AlphaRoot,INFINITY);
      if (HeuristicValue>AlphaRoot)
      {
        AlphaRoot=HeuristicValue;
        BestMoveColIdx=ColumnSortIdx[jc];
        BestMovePiece=gpRed;
      };
      UndoLastMove();
      if (AlphaRoot>=INFINITY) break;
//MIN green
      HeuristicValue=DoMove(gpRed,ColumnSortIdx[jc],gpGreen);
//we do depth-first
      if ((HeuristicValue>-
INFINITY)&&(HeuristicValue<INFINITY)&&(MoveCount<MAX_MOVES))
HeuristicValue=EvaluateMinNodeChildren(AlphaRoot,INFINITY);
      if (HeuristicValue>AlphaRoot)
      {
        AlphaRoot=HeuristicValue;
        BestMoveColIdx=ColumnSortIdx[jc];
        BestMovePiece=gpGreen;
      }
      UndoLastMove();
      if (AlphaRoot>=INFINITY) break;
    }
  }
  if (BestMoveColIdx==-1)
  {
    if (MoveCount<MAX_MOVES-1)
    {
//      fprintf(stderr,"I think I am going to lose, but hey, it was fun!\n");
//although we lose anyway find the best move, who knows
      for (jc=0;jc<ColCount;jc++)
      {
        if (NextRowIdx[ColumnSortIdx[jc]]>=0)
        {
//try the red
          HeuristicValue=DoMove(gpRed,ColumnSortIdx[jc],gpRed);
          if (HeuristicValue>AlphaRoot)
          {
            AlphaRoot=HeuristicValue;
            BestMoveColIdx=ColumnSortIdx[jc];
            BestMovePiece=gpRed;
          };
          UndoLastMove();
//try the green
```

```
          HeuristicValue=DoMove(gpRed,ColumnSortIdx[jc],gpGreen);
          if (HeuristicValue>AlphaRoot)
          {
            AlphaRoot=HeuristicValue;
            BestMoveColIdx=ColumnSortIdx[jc];
            BestMovePiece=gpGreen;
          }
          UndoLastMove();
        }
      }
    }
    else fprintf(stderr,"Illegal. shouldn''t have gotten here. The board is
full.\n");
  }
/*  if (AlphaRoot>=INFINITY) fprintf(stderr,"I'm quite sure I am going to win.\n");
  else
  {
    if (AlphaRoot==0.0)
    {
//possible tie
      if (RedScore>=INFINITY) fprintf(stderr,"So, what do you think of that tie,
huh?\n");
      if (MoveCount>=MAX_MOVES-1) fprintf(stderr,"The board is full. I guess we can
call it a tie.\n");
    }
  }*/
  printf("(%d,%c)\n",BestMoveColIdx+1,PIECE_CHAR[BestMovePiece]);
}


  void UndoLastMove(void)
  {
    int APiece=Board[LastRowMoveIdx][LastColMoveIdx];
    int k;
    for (k=0; k<BoardMapCount[LastRowMoveIdx][LastColMoveIdx]; k++)
    {
      BoardMap[LastRowMoveIdx][LastColMoveIdx][k]->Count[gpSpace]++;
      BoardMap[LastRowMoveIdx][LastColMoveIdx][k]->Count[APiece]--;
    }
    Board[LastRowMoveIdx][LastColMoveIdx]=gpSpace;
    NextRowIdx[LastColMoveIdx]++;
    MoveCount--;
    LastColMoveIdx=SaveColMoveIdx[SavePointer];
    LastRowMoveIdx=SaveRowMoveIdx[SavePointer];
    SavePointer--;
  }

float DoMove(int APlayer, int ACol, int APiece)
{
  int SpaceGap,ColumnSpaceGap,PlayerSituation;
  int RedHotSquareCount,BlackHotSquareCount;

  int i,j,k;
  GroupRec * Group;
//-----------
  NodesVisited++;
//we do not check here whether the move is valid
  SavePointer++;
  SaveColMoveIdx[SavePointer]=LastColMoveIdx;
```

```
  SaveRowMoveIdx[SavePointer]=LastRowMoveIdx;
  LastColMoveIdx=ACol;
  LastRowMoveIdx=NextRowIdx[LastColMoveIdx];
  NextRowIdx[LastColMoveIdx]--;
  MoveCount++;
  Board[LastRowMoveIdx][LastColMoveIdx]=APiece;
///////////////
  RedScore=0.0;
  BlackScore=0.0;
//here we begin calculating the heuristics value
//here we update the 2c2Groups information
  for (k=0; k<BoardMapCount[LastRowMoveIdx][LastColMoveIdx];k++)
  {
    BoardMap[LastRowMoveIdx][LastColMoveIdx][k]->Count[gpSpace]--;
    BoardMap[LastRowMoveIdx][LastColMoveIdx][k]->Count[APiece]++;
  }
//calculating the scores based on all the groups that are affected by this move
//the ones affected are the groupa which contain a square that is locate in the
move column
//here calculation of SpaceGap
  RedHotSquareCount=0;
  BlackHotSquareCount=0;
  for (i=0; i<=NextRowIdx[LastColMoveIdx]+1;i++)
  {
    for (k=0;k<BoardMapCount[i][LastColMoveIdx];k++)
    {
      Group=BoardMap[i][LastColMoveIdx][k];
      SpaceGap=0;
      switch (Group->Direction)
      {
//wdRow
        case 0:
        {
          for (j=0; j<4;j++)
          {
            if (*Group->P[j]==gpSpace) SpaceGap=SpaceGap+NextRowIdx[Group-
>P1Col+j]-Group->P1Row+1;
          }
          break;
        }
//wdColumn the space gap is actually the difference between the current row value
and theGroup row index of this 2c2 group
        case 1:
        {
          ColumnSpaceGap=NextRowIdx[Group->P1Col]-Group->P1Row+1;
          if (ColumnSpaceGap>0) SpaceGap=SpaceGap+ColumnSpaceGap;
          break;
        }
        case 2:
        {
//wdFirstDiagonal
          for (j=0; j<4;j++)
          {
            if (*Group->P[j]==gpSpace) SpaceGap=SpaceGap+NextRowIdx[Group-
>P1Col+j]-(Group->P1Row+j)+1;
          }
          break;
        }
        case 3:
```

```
        {
//wdSecondDiagonal
          for (j=0;j<4;j++)
          {
            if (*Group->P[j]==gpSpace) SpaceGap=SpaceGap+NextRowIdx[Group-
>P1Col+j]-(Group->P1Row-j)+1;
          }
          break;
        }
      }
      if (SpaceGap<0) fprintf(stderr,"Error calculating spacegap");
      Group->SpaceGap=SpaceGap;
//here calc score
      if ((Group->Count[gpRed]<3)&&(Group->Count[gpBlack]<3)&&(Group-
>Count[gpGreen]<3)&&!((Group->Count[gpRed]>0)&&(Group->Count[gpBlack]>0)))
      {
        if (Group->Count[gpRed]>0)
        {
          if ((Group->Count[gpRed]==2)&&(Group->Count[gpGreen]==2))
          {
            RedScore=(float)(RedScore+INFINITY);
          }
          else
          {

RedScore=RedScore+(float)TERITORY_VALUE/(1.0+SPACE_GAP_ZOOM*((float)Group-
>SpaceGap-1.0));
            if ((Group->Count[gpSpace]==1)&&(Group->Direction!=wdColumn))
RedHotSquareCount++;
          }
        }
        else
        {
          if (Group->Count[gpBlack]>0)
          {
            if ((Group->Count[gpBlack]==2)&&(Group->Count[gpGreen]==2))
            {
              BlackScore=(float)(BlackScore+INFINITY);
            }
            else
            {

BlackScore=BlackScore+(float)DefensiveStrategy*TERITORY_VALUE/(1.0+SPACE_GAP_ZOOM*(
(float)Group->SpaceGap-1.0));
              if ((Group->Count[gpSpace]==1)&&(Group->Direction!=wdColumn))
BlackHotSquareCount++;
            }
          }
          else
          {

RedScore=RedScore+(float)TERITORY_VALUE/(1.0+SPACE_GAP_ZOOM*((float)Group-
>SpaceGap-2.0));

BlackScore=BlackScore+(float)DefensiveStrategy*TERITORY_VALUE/(1.0+SPACE_GAP_ZOOM*(
(float)Group->SpaceGap-2.0));
          }
        }
      }
```

```
    }
  }
  RedScore=RedScore+HOT_SQUARE_VALUE*(float)RedHotSquareCount;
  BlackScore=BlackScore+HOT_SQUARE_VALUE*(float)BlackHotSquareCount;
//end score calculation
  if (RedScore>=INFINITY)
  {
    if (BlackScore>=INFINITY) return (float) 0.0;
    else return (float)(INFINITY+1.0);
  }
  else
  {
    if (BlackScore>=INFINITY) return (float)(-INFINITY-1.0);
    else return (float)(RedScore-BlackScore);
  }
}

////////////////////////////////
//Program grad-SP main
//////////////////////////////////

int main(void)
{
  fprintf(stderr,"grad-SP v23\n");
  CreateBoard();
  BuildBoardMap();
  AlphaBeta();
  DestroyBoard();
  return 0;
}
```

```
unit ai;
interface
uses u2c2types,sysutils,graphics,classes,stats,dialogs,dateutils;
{$define NO_DEBUG}
{$define NO_AB_DEBUG}
{$define SCORE_DEBUG}
{$define SORTING_ON}
const
  INFINITY:Real=10000000000000.0;
  HOT_SQUARE_VALUE:Real=100000.0;
  SPACE_GAP_ZOOM:Real=2.0;
  TERITORY_VALUE:Real=10000.0;
  FIRST_DEFENSIVE_STRATEGY:Real=1.1;
  SECOND_DEFENSIVE_STRATEGY:Real=1.1;

//offensive-defensive strategy; if this is over 1, then Black gets bigger weights,
therefore we get defensive
//of less than 1, we are offensive, black weights are set to a lower value
//actual tree depthis PlyDepth +1
  MIN_PLY_DEPTH=3;
  BOARD_TYPE:array[0..1] of string=('even-tiled','odd-tiled');
  THREAT_TYPE:array[0..1] of string=('even','odd');
  PLAYER_ORDER:array[0..1] of string=('first-odd','second-even');

function AIMove(ABoard:string):string;
function EvaluateMinNodeChildren(Alpha,Beta:Real):Real;
function EvaluateMaxNodeChildren(Alpha,Beta:Real):Real;
function DoMove(APlayer,ACol,APiece:Integer):Real;
procedure UndoLastMove;
procedure ScanRow(ARowIdx,Start2C2GroupColIdx,End2C2GroupColIdx:Integer);
procedure ScanCol(AColIdx,Start2C2GroupRowIdx,End2C2GroupRowIdx:Integer);
procedure
ScanFirstDiag(Start2C2GroupRowIdx,Start2C2GroupColIdx,A2C2GroupCount:Integer);
procedure
ScanSecDiag(Start2C2GroupRowIdx,Start2C2GroupColIdx,A2C2GroupCount:Integer);

implementation
var
  DefensiveStrategy:Real;
  AIRowCount,
  AIColCount,
  AILastColMoveIdx,
  AILastRowMoveIdx,
  AIMAX_MOVES,
  AIMoveCount:      Integer;
  AIBoard:          TBoard;
  AIBoardMapCount:  TBoard;
  AIBoardMap:       TBoardMap;
//the row of the new moves
//  NextRowIdx=   array[0..COL_COUNT-1] of Integer;
  AIColumnSortIdx:  array of Integer;
  AIColumnSortCounts:array of Integer;
  AINextRowIdx:     array of Integer;

//  AIHotSquareRowMap:array of Integer;
  AIRowGroup,
  AIColGroup,
  AIFirstDiagGroup,
  AISecDiagGroup:    TGroupArray;
```

```
  AIBoardStr:       string;

////////////////////////////////////////////////////////////////////////////////////
//////////////////SIMPLE AI
var
AISaveColMoveIdx,
AISaveRowMoveIdx:array[1..100] of Integer;
NodesVisited,
PlyDepth,
AISavePointer:  Integer;
RedScore,
BlackScore:   Real;
CurrentDepth: Integer;

procedure CreateBoard;
var
  i,j,
  ConvCode: Integer;
  PieceChar:  Char;

function GetNextToken:string;
var
  CommaPos: Integer;
begin
  CommaPos:=Pos(',',AIBoardStr);
  if CommaPos>0 then
  begin
    Result:=Copy(AIBoardStr,1,CommaPos-1);
    Delete(AIBoardStr,1,CommaPos);
  end
  else Result:=Copy(AIBoardStr,1,Length(AIBoardStr)-1);
end;

begin
  try
//delete the first paranthesis
    Delete(AIBoardStr,1,1);
    Val(GetNextToken,AIColCount,ConvCode);
    Val(GetNextToken,AIRowCount,ConvCode);
    if (AIRowCount<4) or (AIColCount<4) then raise Exception.Create('Invalid board
sizes.');
    Val(GetNextToken,AILastColMoveIdx,ConvCode);
    if (AILastColMoveIdx<0) or (AILastColMoveIdx>AIColCount) then raise
Exception.Create('Invalid last move.');
//from now on, zero based col move, it is an index not a count
    Dec(AILastColMoveIdx);
//allocate dynamically the board
    SetLength(AIBoard,AIRowCount,AIColCount);
//boardmap stuff
    SetLength(AIBoardMap,AIRowCount,AIColCount);
    SetLength(AIBoardMapCount,AIRowCount,AIColCount);
//
    SetLength(AINextRowIdx,AIColCount);
    SetLength(AIColumnSortIdx,AIColCount);
    SetLength(AIColumnSortCounts,AIColCount);
//end new
    for j:=0 to AIColCount-1 do
    begin
      AIColumnSortIdx[j]:=j;
```

```
      AINextRowIdx[j]:=-1;
    end;
    AIMAX_MOVES:=AIRowCount*AIColCount;
    AIMoveCount:=0;
    for i:=0 to AIRowCount-1 do
    begin
      for j:=0 to AIColCount-1 do
      begin
        PieceChar:=GetNextToken[1];
        case PieceChar of
        'r':
          begin
            AIBoard[i][j]:=gpRed;
            Inc(AIMoveCount);
          end;
        'b':
          begin
            AIBoard[i][j]:=gpBlack;
            Inc(AIMoveCount);
          end;
        'g':
          begin
            AIBoard[i][j]:=gpGreen;
            Inc(AIMoveCount);
          end;
        's':
          begin
            AIBoard[i][j]:=gpSpace;
            AINextRowIdx[j]:=i;
          end;
        end;
      end;
    end;
    if AIMoveCount>=AIMAX_MOVES then ShowMessage('Full board.');
    if AILastColMoveIdx>-1 then
    begin
//NextRowIdx is a 0-based array
      AILastRowMoveIdx:=AINextRowIdx[AILastColMoveIdx]+1;
      if AILastRowMoveIdx>AIRowCount-1 then raise Exception.Create('Invalid last
move.');
    end;
//init the info structures
    SetLength(AIRowGroup,AIRowCount,AIColCount-3);
    SetLength(AIColGroup,AIRowCount-3,AIColCount);
    SetLength(AIFirstDiagGroup,AIRowCount-3,AIColCount-3);
    SetLength(AISecDiagGroup,AIRowCount-3,AIColCount-3);
  except
    on E:Exception do ShowMessage('EXCEPTION during creating the board:
'+E.Message);
  end;
end;

procedure DestroyBoard;
begin
//free the dynamically allocated board
  try
    Finalize(AIBoard);
    Finalize(AIBoardMap);
    Finalize(AIBoardMapCount);
```

```
//
    Finalize(AINextRowIdx);
    Finalize(AIColumnSortIdx);
    Finalize(AIColumnSortCounts);
//
    Finalize(AIRowGroup);
    Finalize(AIColGroup);
    Finalize(AIFirstDiagGroup);
    Finalize(AISecDiagGroup);
  except
    on E:Exception do ShowMessage('EXCEPTION during destroying the board:
'+E.Message);
  end;
end;

////////////////////////////////////////////////////
//Scan rows
////////////////////////////////////////////////////
procedure ScanRow(ARowIdx,Start2C2GroupColIdx,End2C2GroupColIdx:Integer);
var
  i,j:  Integer;
  k:    Integer;
begin
  for i:=Start2C2GroupColIdx to End2C2GroupColIdx do
  begin
    for k:=gpRed to gpSpace do AIRowGroup[ARowIdx][i].Count[k]:=0;
    AIRowGroup[ARowIdx][i].Direction:=wdRow;
    AIRowGroup[ARowIdx][i].P1Row:=ARowIdx;
    AIRowGroup[ARowIdx][i].P1Col:=i;
    for j:=0 to 3 do
    begin
      Inc(AIRowGroup[ARowIdx][i].Count[AIBoard[ARowIdx][i+j]]);
      AIRowGroup[ARowIdx][i].P[j]:=@AIBoard[ARowIdx][i+j];
    end;
  end;
end;

////////////////////////////////////////////////////
//Scan columns                        COL
////////////////////////////////////////////////////
procedure ScanCol(AColIdx,Start2C2GroupRowIdx,End2C2GroupRowIdx:Integer);
var
  i,j:  Integer;
  k:    Integer;
begin
  for i:=Start2C2GroupRowIdx to End2C2GroupRowIdx do
  begin
    for k:=gpRed to gpSpace do AIColGroup[i][AColIdx].Count[k]:=0;
    AIColGroup[i][AColIdx].Direction:=wdColumn;
    AIColGroup[i][AColIdx].P1Row:=i;
    AIColGroup[i][AColIdx].P1Col:=AColIdx;
    for j:=0 to 3 do
    begin
      Inc(AIColGroup[i][AColIdx].Count[AIBoard[i+j][AColIdx]]);
      AIColGroup[i][AColIdx].P[j]:=@AIBoard[i+j][AColIdx];
    end;
  end;
end;
////////////////////////////////////////////////////
```

```
//                                                            DIAG1
////////////////////////////////////////////////////
procedure
ScanFirstDiag(Start2C2GroupRowIdx,Start2C2GroupColIdx,A2C2GroupCount:Integer);
var
  i,j:  Integer;
  k:    Integer;
begin
  for i:=0 to A2C2GroupCount-1 do
  begin
    for k:=gpRed to gpSpace do
AIFirstDiagGroup[Start2C2GroupRowIdx+i][Start2C2GroupColIdx+i].Count[k]:=0;

AIFirstDiagGroup[Start2C2GroupRowIdx+i][Start2C2GroupColIdx+i].Direction:=wdFirstDi
agonal;

AIFirstDiagGroup[Start2C2GroupRowIdx+i][Start2C2GroupColIdx+i].P1Row:=Start2C2Group
RowIdx+i;

AIFirstDiagGroup[Start2C2GroupRowIdx+i][Start2C2GroupColIdx+i].P1Col:=Start2C2Group
ColIdx+i;
    for j:=0 to 3 do
    begin

Inc(AIFirstDiagGroup[Start2C2GroupRowIdx+i][Start2C2GroupColIdx+i].Count[AIBoard[St
art2C2GroupRowIdx+i+j][Start2C2GroupColIdx+i+j]]);

AIFirstDiagGroup[Start2C2GroupRowIdx+i][Start2C2GroupColIdx+i].P[j]:=@AIBoard[Start
2C2GroupRowIdx+i+j][Start2C2GroupColIdx+i+j];
    end;
  end;
end;

////////////////////////////////////////////////////
//                                                            DIAG2
////////////////////////////////////////////////////
procedure
ScanSecDiag(Start2C2GroupRowIdx,Start2C2GroupColIdx,A2C2GroupCount:Integer);
var
  i,j:  Integer;
  k:    Integer;
begin
//here is tricky
//careful here: the rows indexes are reversed
  for i:=0 to A2C2GroupCount-1 do
  begin
    for k:=gpRed to gpSpace do
AISecDiagGroup[Start2C2GroupRowIdx+i][Start2C2GroupColIdx+i].Count[k]:=0;

AISecDiagGroup[Start2C2GroupRowIdx+i][Start2C2GroupColIdx+i].Direction:=wdSecondDia
gonal;
    AISecDiagGroup[Start2C2GroupRowIdx+i][Start2C2GroupColIdx+i].P1Row:=AIRowCount-
1-Start2C2GroupRowIdx-i;

AISecDiagGroup[Start2C2GroupRowIdx+i][Start2C2GroupColIdx+i].P1Col:=Start2C2GroupCo
lIdx+i;
    for j:=0 to 3 do
    begin
```

```
Inc(AISecDiagGroup[Start2C2GroupRowIdx+i][Start2C2GroupColIdx+i].Count[AIBoard[AIRo
wCount-1-Start2C2GroupRowIdx-i-j][Start2C2GroupColIdx+i+j]]);

AISecDiagGroup[Start2C2GroupRowIdx+i][Start2C2GroupColIdx+i].P[j]:=@AIBoard[AIRowCo
unt-1-Start2C2GroupRowIdx-i-j][Start2C2GroupColIdx+i+j];
    end;
  end;
end;

procedure BuildBoardMap;
var
  TempValue,
  i,j,k:  Integer;
  Start2C2GroupRowIdx,
  End2C2GroupRowIdx,
  Start2C2GroupColIdx,
  End2C2GroupColIdx: Integer;
  TopLeftIdx,
  The2C2GroupCount: Integer;
begin
//this is done only ONCE
  try
    for j:=0 to AIColCount-1 do
    begin
      AIColumnSortCounts[j]:=0;
      for i:=0 to AIRowCount-1 do
      begin
//assume you move at i,j; which 2c2 groups are affected by this move?
//procedure LastMoveScanRow(ARowMove,AColMove:Integer);
        Start2C2GroupColIdx:=Max(0,j-3);
        End2C2GroupColIdx:=Min(j,AIColCount-4);
        The2C2GroupCount:=End2C2GroupColIdx-Start2C2GroupColIdx+1;
        AIBoardMapCount[i][j]:=The2C2GroupCount;
        for k:=0 to The2C2GroupCount-1 do
        begin
          AIBoardMap[i][j][k]:=@AIRowGroup[i][Start2C2GroupColIdx+k];
          ScanRow(i,Start2C2GroupColIdx,End2C2GroupColIdx);
          if AIBoard[i][j]=gpSpace then Inc(AIColumnSortCounts[j]);
        end;
//procedure LastMoveScanCol(ARowMove,AColMove:Integer);
        Start2C2GroupRowIdx:=Max(0,i-3);
        End2C2GroupRowIdx:=Min(i,AIRowCount-4);
        The2C2GroupCount:=End2C2GroupRowIdx-Start2C2GroupRowIdx+1;
        for k:=0 to The2C2GroupCount-1 do
        begin

AIBoardMap[i][j][AIBoardMapCount[i][j]]:=@AIColGroup[Start2C2GroupRowIdx+k][j];
          Inc(AIBoardMapCount[i][j]);
          ScanCol(j,Start2C2GroupRowIdx,End2C2GroupRowIdx);
          if AIBoard[i][j]=gpSpace then Inc(AIColumnSortCounts[j]);
        end;
//procedure LastMoveScanFirstDiag(ARowMoveIdx,AColMoveIdx:Integer);
        TopLeftIdx:=Min(i,j);
        Start2C2GroupColIdx:=Max(j-TopLeftIdx,j-3);
        Start2C2GroupRowIdx:=Max(i-TopLeftIdx,i-3);
        End2C2GroupColIdx:=Min(j,AIColCount-4);
        End2C2GroupRowIdx:=Min(i,AIRowCount-4);
```

```
        The2C2GroupCount:=Min(End2C2GroupColIdx-
Start2C2GroupColIdx,End2C2GroupRowIdx-Start2C2GroupRowIdx)+1;
        if The2C2GroupCount>0 then
        begin
          for k:=0 to The2C2GroupCount-1 do
          begin

AIBoardMap[i][j][AIBoardMapCount[i][j]]:=@AIFirstDiagGroup[Start2C2GroupRowIdx+k][S
tart2C2GroupColIdx+k];
            Inc(AIBoardMapCount[i][j]);

ScanFirstDiag(Start2C2GroupRowIdx,Start2C2GroupColIdx,The2C2GroupCount);
            if AIBoard[i][j]=gpSpace then Inc(AIColumnSortCounts[j]);
          end;
        end;
//procedure LastMoveScanSecDiag(ARowMoveIdx,AColMoveIdx:Integer);
        TopLeftIdx:=Min(AIRowCount-1-i,j);
        Start2C2GroupColIdx:=Max(j-TopLeftIdx,j-3);
        Start2C2GroupRowIdx:=Max(AIRowCount-1-i-TopLeftIdx,AIRowCount-1-i-3);
        End2C2GroupColIdx:=Min(j,AIColCount-4);
        End2C2GroupRowIdx:=Min(AIRowCount-1-i,AIRowCount-4);
        The2C2GroupCount:=Min(End2C2GroupColIdx-
Start2C2GroupColIdx,End2C2GroupRowIdx-Start2C2GroupRowIdx)+1;
        if The2C2GroupCount>0 then
        begin
          for k:=0 to The2C2GroupCount-1 do
          begin

AIBoardMap[i][j][AIBoardMapCount[i][j]]:=@AISecDiagGroup[Start2C2GroupRowIdx+k][Sta
rt2C2GroupColIdx+k];
            Inc(AIBoardMapCount[i][j]);
            ScanSecDiag(Start2C2GroupRowIdx,Start2C2GroupColIdx,The2C2GroupCount);
            if AIBoard[i][j]=gpSpace then Inc(AIColumnSortCounts[j]);
          end;
        end;
      end;
    end;
//bubble sorting sorting the columns AIColumnSortCounts[j]
{$ifdef SORTING_ON}
  for j:=0 to AIColCount-2 do
    for k:=0 to AIColCount-2-j do
    begin
      if AIColumnSortCounts[k]<AIColumnSortCounts[k+1] then
      begin//swap the values
        TempValue:=AIColumnSortCounts[k];
        AIColumnSortCounts[k]:=AIColumnSortCounts[k+1];
        AIColumnSortCounts[k+1]:=TempValue;
        TempValue:=AIColumnSortIdx[k];
        AIColumnSortIdx[k]:=AIColumnSortIdx[k+1];
        AIColumnSortIdx[k+1]:=TempValue;
      end;
    end;
{$endif}
  except
    on E:Exception do ShowMessage('EXCEPTION during creating the board map:
'+E.Message);
  end;
end;
```

```
/////////////////////////////////////////-----------------------------------------
--------------------------
///////////ALPHA BETA
/////////////////////////////////////////////

//the move column and move piece define the node
function EvaluateMaxNodeChildren(Alpha,Beta:Real):Real;
var
  jc:      Integer;
  HeuristicValue:  Real;
begin
  Inc(CurrentDepth);
//for all children of this node do the evaluate MIN
  jc:=0;
  while jc<AIColCount do
  begin
//we check wether the move is valid
    if AINextRowIdx[AIColumnSortIdx[jc]]>=0 then
    begin
//get the max value of the red child
      HeuristicValue:=DoMove(gpRed,AIColumnSortIdx[jc],gpRed);
      if (HeuristicValue<INFINITY) and (CurrentDepth<PlyDepth) and
(AIMoveCount<AIMAX_MOVES) then HeuristicValue:=EvaluateMinNodeChildren(Alpha,Beta);
      if HeuristicValue>Alpha then Alpha:=HeuristicValue;
      UndoLastMove;
      if Alpha>=Beta then Break;
//get the max value of the green child
      HeuristicValue:=DoMove(gpRed,AIColumnSortIdx[jc],gpGreen);
      if (HeuristicValue<INFINITY) and (CurrentDepth<PlyDepth) and
(AIMoveCount<AIMAX_MOVES) then HeuristicValue:=EvaluateMinNodeChildren(Alpha,Beta);
      if HeuristicValue>Alpha then Alpha:=HeuristicValue;
      UndoLastMove;
      if Alpha>=Beta then Break;
    end;
    Inc(jc);
  end;
  Dec(CurrentDepth);
  Result:=Alpha;
//testing
{$ifdef AB_DEBUG}
  Stats.StatsForm.StatsMemo.Lines.Add('Level: '+IntToStr(CurrentDepth)+', tried
OPONNENT move: ('+PIECE_NAME[0][AMovePiece]+','+Format('[Col: %d], h=
%2.2f',[AMoveColumn+1,Result]));
  Stats.StatsForm.StatsMemo.Lines.Add('----------------------');
{$endif}
end;

//the move column and move piece define the node
function EvaluateMinNodeChildren(Alpha,Beta:Real):Real;
var
  jc:              Integer;
  HeuristicValue:  Real;
begin
//for all children of this node do the evaluate MAX
  jc:=0;
  while jc<AIColCount do
  begin
//we check wether the move is valid
    if AINextRowIdx[AIColumnSortIdx[jc]]>=0 then
```

```
    begin
//we get the min value of the black child
      HeuristicValue:=DoMove(gpBlack,AIColumnSortIdx[jc],gpBlack);
      if (HeuristicValue>-INFINITY) and (AIMoveCount<AIMAX_MOVES) then
HeuristicValue:=EvaluateMaxNodeChildren(Alpha,Beta);
      if HeuristicValue<Beta then Beta:=HeuristicValue;
      UndoLastMove;
      if Alpha>=Beta then Break;
//we get the min value of the green child
      HeuristicValue:=DoMove(gpBlack,AIColumnSortIdx[jc],gpGreen);
      if (HeuristicValue>-INFINITY) and (AIMoveCount<AIMAX_MOVES) then
HeuristicValue:=EvaluateMaxNodeChildren(Alpha,Beta);
      if HeuristicValue<Beta then Beta:=HeuristicValue;
      UndoLastMove;
      if Alpha>=Beta then Break;
    end;
    Inc(jc);
  end;
  Result:=Beta;
//testing
{$ifdef AB_DEBUG}
  Stats.StatsForm.StatsMemo.Lines.Add('Level: '+IntToStr(CurrentDepth)+', tried OWN
move: ('+PIECE_NAME[0][AMovePiece]+','+Format('[Col: %d], h=
%2.2f',[AMoveColumn+1,Result]));
  Stats.StatsForm.StatsMemo.Lines.Add('----------------------');
{$endif}
end;


function AlphaBeta:string;
var
  BestMoveColIdx:  Integer;
  BestMovePiece:  Integer;
  HeuristicValue,
  AlphaRoot:     Real;
  jc:              Integer;
  StartTime,
  EndTime:      TDateTime;
begin
  if AIMoveCount mod 2=0 then
  begin
    DefensiveStrategy:=FIRST_DEFENSIVE_STRATEGY
  end
  else
  begin
    DefensiveStrategy:=SECOND_DEFENSIVE_STRATEGY;
  end;
  NodesVisited:=0;
  PlyDepth:=MIN_PLY_DEPTH+AIMoveCount div (AIMAX_MOVES div 2);
  StartTime:=Time;
{$ifdef AB_DEBUG}
  Stats.StatsForm.StatsMemo.Lines.Add('It looks to me I am the
'+PLAYER_ORDER[AIMoveCount mod 2]+' player on a '+BOARD_TYPE[AIMAX_MOVES mod 2]+'
board.');
{$endif}
  AISavePointer:=0;
  CurrentDepth:=0;
//100* because when things are really bad, we still waht to find the best move
//the best of the worst
```

```
  AlphaRoot:=-INFINITY;
//beta is INFINITY
//this is the root node and has 2*AIColCount MIN children
//we need to get the maximum value of theirs
  BestMovePiece:=-1;
  BestMoveColIdx:=-1;
//  If AIMoveCount=15 then
//  begin
//    ShowMessage('Stop');
//  end;
  for jc:=0 to AIColCount-1 do
  begin
    if AINextRowIdx[AIColumnSortIdx[jc]]>=0 then
    begin
//if the board is full the the returned result is always infinity - not good when
you want to win at the end
//that's why we need to test for AIMAX_MOVES
      HeuristicValue:=DoMove(gpRed,AIColumnSortIdx[jc],gpRed);
//we do depth-first
      if (HeuristicValue<INFINITY) and (AIMoveCount<AIMAX_MOVES) then
HeuristicValue:=EvaluateMinNodeChildren(AlphaRoot,INFINITY);
      if HeuristicValue>AlphaRoot then
      begin
        AlphaRoot:=HeuristicValue;
        BestMoveColIdx:=AIColumnSortIdx[jc];
        BestMovePiece:=gpRed;
      end;
      UndoLastMove;
//this is a sure win, more  than one move away
      if (AlphaRoot>=INFINITY) then Break;
      HeuristicValue:=DoMove(gpRed,AIColumnSortIdx[jc],gpGreen);
//we do depth-first
      if (HeuristicValue>-INFINITY) and (HeuristicValue<INFINITY) and
(AIMoveCount<AIMAX_MOVES) then
HeuristicValue:=EvaluateMinNodeChildren(AlphaRoot,INFINITY);
      if HeuristicValue>AlphaRoot then
      begin
        AlphaRoot:=HeuristicValue;
        BestMoveColIdx:=AIColumnSortIdx[jc];
        BestMovePiece:=gpGreen;
      end;
      UndoLastMove;
      if (AlphaRoot>=INFINITY) then Break
    end;
  end;
//this is the losing gracefully part, we don't want to give up the game
  if BestMoveColIdx=-1 then
  begin
    if AIMoveCount<AIMAX_MOVES-1 then
    begin
//although we lose anyway find the best move, who knows
      ShowMessage('I think I am going to lose, but hey, I tried my best.');
      for jc:=0 to AIColCount-1 do
      begin
        if AINextRowIdx[AIColumnSortIdx[jc]]>=0 then
        begin
//try the green
          HeuristicValue:=DoMove(gpRed,AIColumnSortIdx[jc],gpGreen);
          if HeuristicValue>AlphaRoot then
```

```
          begin
            AlphaRoot:=HeuristicValue;
            BestMoveColIdx:=AIColumnSortIdx[jc];
            BestMovePiece:=gpGreen;
          end;
          UndoLastMove;
//try the red
          HeuristicValue:=DoMove(gpRed,AIColumnSortIdx[jc],gpRed);
          if HeuristicValue>AlphaRoot then
          begin
            AlphaRoot:=HeuristicValue;
            BestMoveColIdx:=AIColumnSortIdx[jc];
            BestMovePiece:=gpRed;
          end;
          UndoLastMove;
        end;
      end;
    end
    else ShowMessage('Illegal. shouldn''t have gotten here. The board is full. I am
making the default move.');
  end;
  if AlphaRoot>=INFINITY then ShowMessage('I''m quite sure I am going to win but
let''s just play some more, will ya?')
  else
  begin
    if AlphaRoot=0 then
    begin
//possible tie
      if RedScore>=INFINITY then ShowMessage('So, what do you think of that tie,
huh?');
      if (AIMoveCount>=AIMAX_MOVES-1) then ShowMessage('The board is full. I guess
we can call it a tie.');
    end;
  end;
//////////////////////////////////////////////////////////////
  EndTime:=Time;
{$ifdef SCORE_DEBUG}
  DoMove(gpRed,BestMoveColIdx,BestMovePiece);
  Stats.StatsForm.StatsMemo.Lines.Add(Format('Best move scores: Red: %2.2f, Black:
%2.2f, DEFENSE:%2.2f, Alpha:%2.2f, Nodes visited: %d, Ply Depth: %d, Movetime:
%2.2f]',

[RedScore,BlackScore,DefensiveStrategy,AlphaRoot,NodesVisited,PlyDepth,(Millisecond
sBetween(EndTime,StartTime)/1000)]));
  UndoLastMove;
  Stats.StatsForm.StatsMemo.Lines.Add('************************');
{$endif}
  Result:='('+IntToStr(BestMoveColIdx+1)+','+PIECE_CHAR[0][BestMovePiece]+')';
end;

//////////////////////////scanning procedures

procedure UndoLastMove;
var
  k: Integer;
  APiece: Integer;
begin
  try
//here we update the 2c2Groups information
```

```
    APiece:=AIBoard[AILastRowMoveIdx][AILastColMoveIdx];
    for k:=0 to AIBoardMapCount[AILastRowMoveIdx][AILastColMoveIdx]-1 do
    begin
      Inc(AIBoardMap[AILastRowMoveIdx][AILastColMoveIdx][k]^.Count[gpSpace]);
      Dec(AIBoardMap[AILastRowMoveIdx][AILastColMoveIdx][k]^.Count[APiece]);
    end;
    AIBoard[AILastRowMoveIdx][AILastColMoveIdx]:=gpSpace;
    Inc(AINextRowIdx[AILastColMoveIdx]);
    Dec(AIMoveCount);
    AILastColMoveIdx:=AISaveColMoveIdx[AISavePointer];
    AILastRowMoveIdx:=AISaveRowMoveIdx[AISavePointer];
    Dec(AISavePointer);
  except
    on E:Exception do ShowMessage('EXCEPTION during making the move: '+E.Message);
  end;
end;

function DoMove(APlayer,ACol,APiece:Integer):Real;
var
  ColumnSpaceGap,
  SpaceGap,
  i,j,k: Integer;
  Group: PGroupRec;
begin
  Result:=0.0;
  try
    Inc(NodesVisited);
//we do not check here whether the move is valid
    Inc(AISavePointer);
    AISaveColMoveIdx[AISavePointer]:=AILastColMoveIdx;
    AISaveRowMoveIdx[AISavePointer]:=AILastRowMoveIdx;
    AILastColMoveIdx:=ACol;
    AILastRowMoveIdx:=AINextRowIdx[AILastColMoveIdx];
    Dec(AINextRowIdx[AILastColMoveIdx]);
    Inc(AIMoveCount);
    AIBoard[AILastRowMoveIdx][AILastColMoveIdx]:=APiece;
/////
    RedScore:=0.0;
    BlackScore:=0.0;
//here we begin calculating the heuristics value
//here we update the 2c2Groups information
    for k:=0 to AIBoardMapCount[AILastRowMoveIdx][AILastColMoveIdx]-1 do
    begin
      Dec(AIBoardMap[AILastRowMoveIdx][AILastColMoveIdx][k]^.Count[gpSpace]);
      Inc(AIBoardMap[AILastRowMoveIdx][AILastColMoveIdx][k]^.Count[APiece]);
//--here find teh space gap based on the lasmove row col
    end;
//calculating the scores based on all the groups that are affected by this move
//the ones affected are the groupa which contain a square that is locate in the
move column
//here calculation of SpaceGap
    for i:=0 to AINextRowIdx[AILastColMoveIdx]+1 do
    begin
      for k:=0 to AIBoardMapCount[i][AILastColMoveIdx]-1 do
      begin
        Group:=AIBoardMap[i][AILastColMoveIdx][k];
        SpaceGap:=0;
        case Group^.Direction of
        0://wdRow
```

```
          for j:=0 to 3 do if Group^.P[j]^=gpSpace then
Inc(SpaceGap,AINextRowIdx[Group^.P1Col+j]-Group^.P1Row+1);
        1://wdColumn the space gap is actually the difference between the current
row value and theGroup row index of this 2c2 group
          begin
            ColumnSpaceGap:=AINextRowIdx[Group^.P1Col]-Group^.P1Row+1;
            if (ColumnSpaceGap>0) then Inc(SpaceGap,ColumnSpaceGap);
          end;
        2://wdFirstDiagonal
          for j:=0 to 3 do if Group^.P[j]^=gpSpace then
Inc(SpaceGap,AINextRowIdx[Group^.P1Col+j]-(Group^.P1Row+j)+1);
        3://wdSecondDiagonal
          for j:=0 to 3 do if Group^.P[j]^=gpSpace then
Inc(SpaceGap,AINextRowIdx[Group^.P1Col+j]-(Group^.P1Row-j)+1);
        end;
        if SpaceGap<0 then ShowMessage('Error calculating spacegap');
        Group^.SpaceGap:=SpaceGap;
//here calc score
        with Group^ do
        begin
          if (Count[gpRed]<3) and (Count[gpBlack]<3) and (Count[gpGreen]<3) and not
((Count[gpRed]>0) and (Count[gpBlack]>0)) then
          begin
            if (Count[gpRed]>0) then
            begin
              if (Count[gpRed]=2) and (Count[gpGreen]=2) then
RedScore:=RedScore+INFINITY
              else
              begin

RedScore:=RedScore+TERITORY_VALUE/(1.0+SPACE_GAP_ZOOM*(Group.SpaceGap-1.0));
//                if (Count[gpSpace]=1) and (Direction<>wdColumn) then
RedScore:=RedScore+HOT_SQUARE_VALUE/(1.0+SPACE_GAP_ZOOM*(Group.SpaceGap-1.0));
              end;
            end
            else
            begin
              if (Count[gpBlack]>0) then
              begin
                if (Count[gpBlack]=2) and (Count[gpGreen]=2) then
BlackScore:=BlackScore+INFINITY
                else
                begin

BlackScore:=BlackScore+DefensiveStrategy*TERITORY_VALUE/(1.0+SPACE_GAP_ZOOM*(Group.
SpaceGap-1.0));
//                  if (Count[gpSpace]=1) and (Direction<>wdColumn) then
BlackScore:=BlackScore+DefensiveStrategy*HOT_SQUARE_VALUE/(1.0+SPACE_GAP_ZOOM*(Grou
p.SpaceGap-1.0));
                end;
              end
              else
              begin

RedScore:=RedScore+TERITORY_VALUE/(1.0+SPACE_GAP_ZOOM*(Group.SpaceGap-2.0));

BlackScore:=BlackScore+DefensiveStrategy*TERITORY_VALUE/(1.0+SPACE_GAP_ZOOM*(Group.
SpaceGap-2.0));
              end;
```

```
            end;
          end;
        end;
      end;
    end;
//end score calculation
    if RedScore>=INFINITY then
    begin
      if BlackScore>=INFINITY then Result:=0.0
      else Result:=INFINITY+1;
    end
    else
    begin
      if BlackScore>=INFINITY then Result:=-INFINITY-1
      else Result:=RedScore-BlackScore;
    end
  except
    on E:Exception do ShowMessage('EXCEPTION during making the move: '+E.Message);
  end;
end;

function AIMove(ABoard:string):string;
begin
  AIBoardStr:=ABoard;
  CreateBoard;
  BuildBoardMap;
//here progressive deepening
  whle Expended nodes<MAX_EXPAND_NODE do Result:=AlphaBeta;
  Stats.StatsForm.StatsMemo.Lines.Add('----------------------');
  DestroyBoard;
end;

initialization
  NodesVisited:=0;
end.
```