Artificial Intelligence in the Game of Othello

Dylan Griffith
(420122140/dgri9968)
School of Physics,
University of Sydney,
New South Wales,
Australia.
dyl.griffith@gmail.com

April 19, 2012

Contents

Chapter 1

Strategies

1.1 Strategy A

1.1.1 Description

It was determined that the minimax algorithm was only really able to reach a depth of 4 before the wait time to make a move became too long for an enjoyable gaming experience. The key method for testing this was by implementing also the negamax algorithm which can be shown to be equivalent to minimax when the evaluation function is antisymmetric with respect to change of player (ie. $\text{Evaluation}(\text{State}, \text{Player1}) = -\text{Evaluation}(\text{State}, \text{Player2})$). The move chosen by minimax and negamax (and the state evaluations) were compared across several different games to ensure that these were giving the same result every move.

In order to make sure the algorithms were function correctly the depth was reduced to 1 and some examples were calculated by hand and these were compared with the results of the algorithm. Finally more support for the correctness of this algorithm was derived from the fact that it played very well against a human player and defeated a random moving player every time it was tested (in over 100 rounds).

1.1.2 Code

```
1  int OthelloAI::minimax(Board *board, int depth, int player)
2  {
3    int score;
4    if (depth <= 0 || Game::isTerminal(board))
5      return scoreBoardAdvanced(board,me);
6    bool cantMove = true;
7    if(player == me)
8    {
9      score = -2*INFINITY;
10     for(int i=0;i<GRIDSIZE;i++)
11     {
12       for(int j=0;j<GRIDSIZE;j++)
13       {
```

```
14          if(Game::isLegalMove(i,j,board,player))
15          {
16            Board child (board, i, j, player);
17            cantMove = false;
18            score = max(score, minimax(&child, depth-1, otherPlayer(player)))
                  ;
19          }
20        }
21      }
22      if(cantMove) // Child node is same node
23        score = minimax(board, depth, otherPlayer(player));
24    }else
25    {
26      score = 2*INFINITY;
27      for(int i=0;i<GRIDSIZE;i++)
28      {
29        for(int j=0;j<GRIDSIZE;j++)
30        {
31          if(Game::isLegalMove(i,j,board,player))
32          {
33            Board child (board, i, j, player);
34            cantMove = false;
35            score = min(score, minimax(&child, depth-1, otherPlayer(player)))
                  ;
36          }
37        }
38      }
39      if(cantMove) // Child node is same node
40        score = minimax(board, depth, otherPlayer(player));
41    }
42    return score;
43  }
```

1.2 Strategy B

1.2.1 Description

In the minimax with alpha-beta pruning the depth was increased to 5 and it still allowed for enjoyable gaming. Testing was similar to that of Strategy A. Firstly it was noted that alpha-beta pruning should return the same best move as minimax without the pruning. Thus, many trials to compare this algorithm with the two from the previous section were undertaken and all tests passed. It was also checked against simple examples to ensure that it was pruning the correct branches of the search tree and that it gave the correct result and these tests all passed. Finally it was shown to beat a human player every time and also beat a random moving player every time across many trial matches.

1.2.2 Code

```
1  int OthelloAI::alphabeta(Board *board, int depth, int alpha, int beta, int
       player)
2  {
3    if (depth <= 0 || Game::isTerminal(board))
4      return scoreBoardAdvanced(board,me);
5    bool cantMove = true;
6    if(player == me)
7    {
8      // Queue children of state
9      for(int i=0;i<GRIDSIZE*GRIDSIZE;i++)
10     {
11       int x = i%GRIDSIZE;
12       int y = i/GRIDSIZE;
13       if(Game::isLegalMove(x,y,board,player))
14       {
15         cantMove = false;
16         alpha = max(alpha, alphabeta(&Board(board,x,y,player), depth-1,
               alpha, beta, otherPlayer(player)));
17         if(beta <= alpha)
18           return alpha;
19       }
20     }
21     if(cantMove) // Child node is same node
22       alpha = max(alpha, alphabeta(board, depth, alpha, beta, otherPlayer(
             player)));
23     return alpha;
24   }else
25   {
26     // Queue children of state
27     for(int i=0;i<GRIDSIZE*GRIDSIZE;i++)
28     {
29       int x = i%GRIDSIZE;
30       int y = i/GRIDSIZE;
31       if(Game::isLegalMove(x,y,board,player))
32       {
33         cantMove = false;
34         beta = min(beta, alphabeta(&Board(board,x,y,player), depth-1, alpha
               , beta, otherPlayer(player)));
35         if(beta <= alpha)
36           return beta;
37       }
38     }
39     if(cantMove) // Child node is same node
40       beta = min(beta, alphabeta(board, depth, alpha, beta, otherPlayer(
             player)));
41     return beta;
42   }
43 }
```

1.3 Strategy C

1.3.1 Description

The key feature in this third strategy was the idea of 'Book Moves'. Certain States of the game tree were stored in a book which is loaded into the AI as the game is initialised. These moves were stored in a hash table to allow for constant time lookups by the AI during run time. The way the "Book" was constructed was by allowing the AI to play many games, while searching the game tree at a very high value of depth ≥ 11, against a random moving AI and output the chosen move to the Book. This 'learning phase' took place over several hours running multiple of instances of the program.

Finally a technique known as "Scouting" was used to improve the speed of the minimax algorithm even further. The basic principle involves firstly expanding the children of a node in order based on some priority (with the emphasis on expanding more promising children first) and then perform a so called 'Null Window' search on all but the first expanded child. This "Null Window", which involves setting $\alpha - \beta = 1$ in order to quickly show that the child can not give a better result than the previous child. If the test fails then the algorithm reverts to usual Minimax with Alpha-Beta pruning. After several trials the depth was able to be increased to 8 and still allow for snappy move determination. The priority of the nodes was calculated using a simple evaluation on the children states which is the number of tokens owned by the scoring player minus those owned by the opposing player.

The following three tools were used for testing this strategy for correctness:

- The algorithm always chose the same path as normal Minimax with Alpha-Beta pruning (and gave the same scores for the same nodes)
- The algorithm always made a legal move choice (ie. there were no illegal moves written to the book)
- The algorithm outperformed (in terms of speed) Minimax with Alpha-Beta pruning when the ordering was sensible and was approximately equivalent when there was no ordering

1.3.2 Code

```
1  int OthelloAI::negascout(Board *board, int depth, int alpha, int beta, int
       player)
2  {
3    if (depth <= 0 || Game::isTerminal(board))
4    {
5      int score = scoreBoardAdvanced(board,player);
6      return score;
7    }
8    priority_queue<BoardOrder> children;
9    bool cantMove = true;
10
```

```
11    // Queue children of state
12    for(int i=0;i<GRIDSIZE*GRIDSIZE;i++)
13    {
14      int x = i%GRIDSIZE;
15      int y = i/GRIDSIZE;
16      if(Game::isLegalMove(x,y,board,player))
17      {
18        cantMove = false;
19        Board *child = &Board(board,x,y,player);
20        // Prioritise depending on whether it is min turn or
21        // max turn
22        int priority;
23        if(player == me)
24          priority = scoreBoardSimple(child,me);
25        else
26          priority = -scoreBoardSimple(child,me);
27        children.push(BoardOrder (priority, *child));
28      }
29    }
30    if(cantMove)
31    {
32      // Child is same node
33      children.push(BoardOrder (0, *board));
34      depth++; // Increment depth for consistency with other algorithms
35    }
36
37    bool firstChild = true;
38    int b = beta; // scouting variable
39    while(children.size() > 0)
40    {
41      Board child = (children.top().board);
42      children.pop();
43      int score = -negascout(&child, depth-1, -b, -alpha, otherPlayer(player)
          );
44      if(alpha < score && score < beta && !firstChild) // Perform full alpha-
          beta search
45        score = -negascout(&child, depth - 1, -beta, -alpha, otherPlayer(
            player));
46      firstChild = false;
47      alpha = max(alpha, score);
48      if(beta <= alpha)
49        return alpha;
50      b = alpha + 1;
51    }
52
53    return alpha;
54  }
```

1.4 Evaluation Function

1.4.1 Description

Initially the evaluation function was the simple one described in the previous section. Modifications were first made by noting that maximising the number of moves a player could make and minimising the possible moves of the opponent meant that the opponent was more likely to have to skip a turn and this can give quite an advantage. Also in the game of Othello the corner pieces are more valuable given that they cannot be taken by an opponent. Thus a corner bonus was also added. The advanced evaluation function is summarised in the following:

- 10 points are given to every square owned by the player
- 10 points are taken off for every square owned by the other player
- 5 points is given for every possible legal move
- 5 points are taken off for every possible legal move of the opponent
- 50 points are added for every corner piece
- 50 points removed for every corner piece owned by the opponent

Initial tests involving the AI vs. AI with one using the simple evaluation and the other using the advanced evaluation showed the advanced one to be superior.

1.4.2 Code

```
1  int scoreBoardAdvanced(Board *board, int player)
2  {
3    int scoreSimple = scoreBoardSimple(board,player);
4
5    if(Game::isTerminal(board))
6    {
7      // player wins
8      if(scoreSimple > 0)
9        return INFINITY;
10     // player loses
11     if(scoreSimple < 0)
12       return -INFINITY;
13     // match was a draw (draw is more desirable
14     // than a loss but still less desirable than
15     // any other state)
16     else
17       return -INFINITY/2;
18   }
19
20   // Add 10 points for every square owned by player and
21   // subtract 10 for every square owned by other player
22   int score = 10*scoreSimple;
```

```
23
24    // Add 5 points for every legal move and subtract
25    // 5 for every legal move of opponent
26    for(int i=0;i<GRIDSIZE;i++)
27    {
28      for(int j=0;j<GRIDSIZE;j++)
29      {
30
31        if(Game::isLegalMove(i,j,board,player))
32        {
33          score += 5;
34        }
35        if(Game::isLegalMove(i,j,board,otherPlayer(player)))
36        {
37          score -= 5;
38        }
39      }
40    }
41
42    // Add 50 points for every corner owned by player
43    // and subtract 50 points for every corner owned
44    // by other player
45    int cornerBonus = 0;
46    int corner = board->getSquare(0,0);
47    if(corner == player)
48      cornerBonus += 50;
49    else if(corner == otherPlayer(player))
50      cornerBonus -= 50;
51    corner = board->getSquare(0,7);
52    if(corner == player)
53      cornerBonus += 50;
54    else if(corner == otherPlayer(player))
55      cornerBonus -= 50;
56    corner = board->getSquare(7,0);
57    if(corner == player)
58      cornerBonus += 50;
59    else if(corner == otherPlayer(player))
60      cornerBonus -= 50;
61    corner = board->getSquare(7,7);
62    if(corner == player)
63      cornerBonus += 50;
64    else if(corner == otherPlayer(player))
65      cornerBonus -= 50;
66    return score + cornerBonus;
67  }
```

Chapter 2

Results

2.1 Human vs. AI

In order to test the different strategy a person was made to play 10 games in a row against each strategy. It was found that the AI beat the human player every time in every strategy. This is mostly due to the fact that computers are much better at playing these types of games than humans.

Some notes taken when playing against the AI player are given below.

Notes:

- Mostly the AI can return moves very quickly but in some pathological cases where there are many possible moves each with a similar outcome (eg. near the final move for the winning player) the AI takes some time to move
- Winning against the AI even when it is searching a depth of 2 is very difficult
- The AI did not have any element of randomness and thus the same games were played often (and certainly the first few moves of each game was fairly consistent)

2.2 Ai vs. AI

Since the MiniMax and MiniMax with Alpha-Beta pruning play equivalently only strategy B and C were compared against eachother. It was found that the Strategy C was superior when moving first and when moving second playing against strategy B. This confirms that the improvements made do indeed create a 'smarter' AI.

2.3 Speed

In order to compare the speed of each of the algorithm each one was made to play 10 games against a random moving player and the times are recorded in the table below.

2.4 Memory

Using the same method to compare speeds the memory used during the trials were recorded. The differences were negligible with the application using typically $\sim 1200K$ memory for all strategies.

2.5 Conclusions

2.5.1 Strengths and Weaknesses

Strategy B was shown to be faster than Strategy A for depths exceeding 4, however for low depths Strategy B was slower due to the additional operations need for Alpha-Beta pruning. All of the strengths of Minimax are inherint in both Strategy A and Strategy B, but Strategy B has the added strength of being able to prune less promising branches from the search tree. There are certain cases where this is very useful (eg. when some branches are clearly worse than the best one found), but in some cases it offers barely any performance improvement (eg. when all branches show similar promise) and has additional overheads which are mostly wasteful.

The scouting algorithm part of Strategy C encompasses all of the strengths of Minimax with Alpha-Beta pruning and improves on it by using some information about the children nodes and expanding in the order which will allow for greatest pruning. Strategy C has some additional operations compared with B, but at high depths (depth > 7) it computes much faster than B. There may be cases where Strategy C runs very slowly, but during testing no cases were found. Strategy C relies heavily on information about board states and without a good understanding of how to play the game well it would not be able to achieve what it does.

The book moves in Strategy C allowed for searching a great depth of the game tree in advanced such that the first few moves in a game could be selected quickly and would be sure to be very good moves. This feature required that the program be run in a 'Learning Phase' for a long time before it would actually be useful.

2.5.2 Reflections

Testing and debugging these algorithms was the greatest challenge of this assignment. When it is clear that the AI is not making correct decisions it is often very hard to find the cause of the error. This is due to the fact that the AI relies on several things behaving correctly including the back end of the Othello game and a sensible evaluation function.

This assignment made very clear that computers are much better at playing simple board games, such as Othello, than are humans. The Minimax algorithm is a very powerful tool in adversarial zero sum games.

2.5.3 Future research

This program is by far not the perfect Othello AI. There are many improvements that could be made to different aspects. Some improvements that could potentially be promising include:

- modifying the evaluation function based on a more thorough study of Othello tactics
- making a learning evaluation function where the evaluation value would be based on previously played games (eg. wins/losses)
- modifying the prioritisation evaluation used in the scouting algorithm such that previously pruned branches of the search tree are evaluated last

Chapter 3

Complete Code

3.1 Graphical User Interface

The GUI was completely controlled by the code from the Othello.cpp and Othello.h files. Also there are some definitions in Resource.h.

3.1.1 Othello.cpp

```
1  // Othello.cpp : Defines the entry point for the application.
2  //
3  #include "stdafx.h"
4  #include "Othello.h"
5  #include "Board.h"
6  #include "Game.h"
7
8
9  #include <stdlib.h>
10
11 using namespace std;
12
13 #define MAX_LOADSTRING 100
14 #define GRIDWIDTH 90
15 #define AI_AI_GAMES 1
16 // Global Variables:
17 HINSTANCE hInst;                    // current instance
18 TCHAR szTitle[MAX_LOADSTRING];           // The title bar text
19 TCHAR szWindowClass[MAX_LOADSTRING];       // the main window class name
20 Game *game;                        // the game currently being played
21
22 int strategy1; // The strategy used by AI (defined in Game.h)
23 bool noplayer; // True if there is no human player (ie. AI vs. AI)
24 int strategy2; // Used only if noplayer is true
25 int humanPlayer;
26 int aiPlayer1;
27 int aiPlayer2;
28
```

```
29 // Forward declarations of functions included in this code module:
30 ATOM            MyRegisterClass(HINSTANCE hInstance);
31 BOOL            InitInstance(HINSTANCE, int);
32 LRESULT CALLBACK   WndProc(HWND, UINT, WPARAM, LPARAM);
33 INT_PTR CALLBACK   About(HWND, UINT, WPARAM, LPARAM);
34
35
36 // Draws the board in the window hdc using the global variable game.
37 void drawBoard(HDC);
38 void drawToken(HDC hdc, int x, int y, int player);
39
40 int APIENTRY _tWinMain(_In_ HINSTANCE hInstance,
41                        _In_opt_ HINSTANCE hPrevInstance,
42                        _In_ LPTSTR    lpCmdLine,
43                        _In_ int       nCmdShow)
44 {
45   UNREFERENCED_PARAMETER(hPrevInstance);
46   UNREFERENCED_PARAMETER(lpCmdLine);
47
48   // TODO: Place code here.
49   MSG msg;
50   HACCEL hAccelTable;
51
52   // Initialize global strings
53   LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
54   LoadString(hInstance, IDC_OTHELLO, szWindowClass, MAX_LOADSTRING);
55   MyRegisterClass(hInstance);
56
57   // Perform application initialization:
58   if (!InitInstance (hInstance, nCmdShow))
59   {
60     return FALSE;
61   }
62
63   hAccelTable = LoadAccelerators(hInstance, MAKEINTRESOURCE(IDC_OTHELLO));
64
65   // Main message loop:
66   while (GetMessage(&msg, NULL, 0, 0))
67   {
68     if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
69     {
70       TranslateMessage(&msg);
71       DispatchMessage(&msg);
72     }
73   }
74
75   return (int) msg.wParam;
76 }
77
78
79
80 //
81 //  FUNCTION: MyRegisterClass()
```

```
82 //
83 //  PURPOSE: Registers the window class.
84 //
85 ATOM MyRegisterClass(HINSTANCE hInstance)
86 {
87   WNDCLASSEX wcex;
88
89   wcex.cbSize = sizeof(WNDCLASSEX);
90
91   wcex.style         = CS_HREDRAW | CS_VREDRAW;
92   wcex.lpfnWndProc   = WndProc;
93   wcex.cbClsExtra    = 0;
94   wcex.cbWndExtra    = 0;
95   wcex.hInstance     = hInstance;
96   wcex.hIcon         = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_OTHELLO));
97   wcex.hCursor       = LoadCursor(NULL, IDC_HAND);
98   wcex.hbrBackground  = (HBRUSH) CreateSolidBrush(RGB(0, 62, 0));
99   wcex.lpszMenuName = MAKEINTRESOURCE(IDC_OTHELLO);
100  wcex.lpszClassName  = szWindowClass;
101  wcex.hIconSm       = LoadIcon(wcex.hInstance, MAKEINTRESOURCE(IDI_SMALL));
102
103  return RegisterClassEx(&wcex);
104 }
105
106 //
107 //   FUNCTION: InitInstance(HINSTANCE, int)
108 //
109 //   PURPOSE: Saves instance handle and creates main window
110 //
111 //   COMMENTS:
112 //
113 //        In this function, we save the instance handle in a global
        variable and
114 //        create and display the main program window.
115 //
116 BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
117 {
118   HWND hWnd;
119   game = new(Game);
120   strategy1 = AI_STRATEGY_B;
121   strategy2 = AI_STRATEGY_A;
122   humanPlayer = BLACK;
123   aiPlayer1 = BLACK;
124   aiPlayer2 = WHITE;
125   noplayer = false;
126
127
128   hInst = hInstance; // Store instance handle in our global variable
129
130   hWnd = CreateWindow(szWindowClass, szTitle, WS_CAPTION | WS_SYSMENU |
        WS_MINIMIZEBOX,
131     10, 10, GRIDSIZE*GRIDWIDTH + 17, GRIDSIZE*GRIDWIDTH + 60, NULL, NULL,
          hInstance, NULL);
```

```
132
133     if (!hWnd)
134     {
135        return FALSE;
136     }
137
138     ShowWindow(hWnd, nCmdShow);
139     UpdateWindow(hWnd);
140
141     return TRUE;
142  }
143
144  //
145  //  FUNCTION: WndProc(HWND, UINT, WPARAM, LPARAM)
146  //
147  //  PURPOSE:  Processes messages for the main window.
148  //
149  //  WM_COMMAND - process the application menu
150  //  WM_PAINT - Paint the main window
151  //  WM_DESTROY - post a quit message and return
152  //
153  //
154  LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
       lParam)
155  {
156     int wmId, wmEvent;
157     PAINTSTRUCT ps;
158     HDC hdc;
159
160     switch (message)
161     {
162     case WM_COMMAND:
163        wmId    = LOWORD(wParam);
164        wmEvent = HIWORD(wParam);
165
166        // Parse the menu selections:
167        switch (wmId)
168        {
169        case IDM_ABOUT:
170           DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd, About);
171           break;
172        case IDM_EXIT:
173           DestroyWindow(hWnd);
174           break;
175        case IDM_AIFIRST:
176           delete game;
177           game = new(Game);
178           noplayer = false;
179           game->makeMoveAI(strategy2);
180           humanPlayer = WHITE;
181           aiPlayer2 = BLACK;
182           InvalidateRect(hWnd, 0, TRUE);
183           break;
```

```
184       case IDM_AISECOND:
185         delete game;
186         game = new(Game);
187         noplayer = false;
188         humanPlayer = BLACK;
189         aiPlayer2 = WHITE;
190         InvalidateRect(hWnd, 0, TRUE);
191         break;
192       case IDM_STRATA:
193         strategy2 = AI_STRATEGY_A;
194         InvalidateRect(hWnd, 0, TRUE);
195         break;
196       case IDM_STRATB:
197         strategy2 = AI_STRATEGY_B;
198         InvalidateRect(hWnd, 0, TRUE);
199         break;
200       case IDM_STRATC:
201         strategy2 = AI_STRATEGY_C;
202         InvalidateRect(hWnd, 0, TRUE);
203         break;
204       case IDM_STRATD:
205         strategy2 = AI_STRATEGY_D;
206         InvalidateRect(hWnd, 0, TRUE);
207         break;
208       case IDM_STRATE:
209         strategy2 = AI_STRATEGY_E;
210         InvalidateRect(hWnd, 0, TRUE);
211         break;
212       case IDM_NOPLAYER:
213         delete game;
214         game = new(Game);
215         strategy1 = AI_STRATEGY_D;
216         strategy2 = AI_STRATEGY_B;
217         aiPlayer2 = BLACK;
218         aiPlayer1 = WHITE;
219         noplayer = true;
220         InvalidateRect(hWnd, 0, TRUE);
221         break;
222       case IDM_LEARNING:
223         delete game;
224         game = new(Game);
225         noplayer = true;
226         aiPlayer1 = BLACK;
227         aiPlayer2 = WHITE;
228         strategy1 = AI_STRATEGY_D;
229         strategy2 = AI_STRATEGY_E;
230         InvalidateRect(hWnd, 0, TRUE);
231         break;
232       default:
233         return DefWindowProc(hWnd, message, wParam, lParam);
234       }
235       break;
236     case WM_LBUTTONDOWN:
```

```
237     {
238       if(noplayer)
239       {
240         for(int counter = 0;counter < AI_AI_GAMES; counter++)
241         {
242           delete game;
243           game = new(Game);
244           while(!game->isOverGame())
245           {
246             if(game->getCurrentPlayer() == aiPlayer1)
247             {
248               game->makeMoveAI(strategy1);
249             }else
250             {
251               game->makeMoveAI(strategy2);
252             }
253           }
254           // Swap players
255           int temp = aiPlayer1;
256           aiPlayer1 = aiPlayer2;
257           aiPlayer2 = temp;
258         }
259         int winner = game->whoWins();
260         if(winner == WHITE)
261           ::MessageBox(hWnd, _T("The winner is white!"), _T("Game over"),
                MB_OK | MB_ICONEXCLAMATION);
262         else if (winner == BLACK)
263           ::MessageBox(hWnd, _T("The winner is black!"), _T("Game over"),
                MB_OK | MB_ICONEXCLAMATION);
264         else
265           ::MessageBox(hWnd, _T("It was a draw!"), _T("Game over"), MB_OK |
                MB_ICONEXCLAMATION);
266         InvalidateRect(hWnd, 0, TRUE);
267         break;
268       }else
269       {
270         int x,y;
271         x = LOWORD(lParam)/GRIDWIDTH;
272         y = HIWORD(lParam)/GRIDWIDTH;
273         if (game->isLegalMove(x,y)) {
274           game->makeMove(x,y);
275           while(!game->isOverGame())
276           {
277             if(game->getCurrentPlayer() == aiPlayer2)
278               game->makeMoveAI(strategy2);
279             else
280               break;
281           }
282           if(game->isOverGame())
283           {
284             int winner = game->whoWins();
285             if(winner == WHITE)
```

```
286                 ::MessageBox(hWnd, _T("The winner is white!"), _T("Game over"
                        ), MB_OK | MB_ICONEXCLAMATION);
287               else if (winner == BLACK)
288                 ::MessageBox(hWnd, _T("The winner is black!"), _T("Game over"
                        ), MB_OK | MB_ICONEXCLAMATION);
289               else
290                 ::MessageBox(hWnd, _T("It was a draw!"), _T("Game over"),
                        MB_OK | MB_ICONEXCLAMATION);
291             }
292
293           } else {
294             // Invalid move
295           }
296           // Repaint the window after the update
297           InvalidateRect(hWnd, 0, TRUE);
298           break;
299         }
300       }
301     case WM_PAINT:
302       hdc = BeginPaint(hWnd, &ps);
303       // Draw Grid
304       drawBoard(hdc);
305       EndPaint(hWnd, &ps);
306       break;
307     case WM_DESTROY:
308       PostQuitMessage(0);
309       break;
310     default:
311       return DefWindowProc(hWnd, message, wParam, lParam);
312     }
313     return 0;
314 }
315
316 // Message handler for about box.
317 INT_PTR CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM
        lParam)
318 {
319     UNREFERENCED_PARAMETER(lParam);
320     switch (message)
321     {
322     case WM_INITDIALOG:
323       return (INT_PTR)TRUE;
324
325     case WM_COMMAND:
326       if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
327       {
328         EndDialog(hDlg, LOWORD(wParam));
329         return (INT_PTR)TRUE;
330       }
331       break;
332     }
333     return (INT_PTR)FALSE;
334 }
```

```
335
336 void drawBoard(HDC hdc)
337 {
338   HPEN hPenOld;
339   // Draw the board lines
340   HPEN hLinePen;
341   COLORREF qLineColor;
342   qLineColor = RGB(0, 0, 0);
343   hLinePen = CreatePen(PS_SOLID, 2, qLineColor);
344   hPenOld = (HPEN)SelectObject(hdc, hLinePen);
345
346
347   for (int i = GRIDWIDTH; i < GRIDSIZE*GRIDWIDTH; i += GRIDWIDTH) {
348     MoveToEx(hdc, i, 0, NULL);
349     LineTo(hdc, i, GRIDSIZE*GRIDWIDTH);
350   }
351
352
353   for (int i = GRIDWIDTH; i < GRIDSIZE*GRIDWIDTH; i += GRIDWIDTH) {
354     MoveToEx(hdc, 0, i, NULL);
355     LineTo(hdc, GRIDSIZE*GRIDWIDTH, i);
356   }
357
358   Board currentBoard = game->getBoard();
359   for (int i=0; i<GRIDSIZE;i++)
360   {
361     for(int j=0; j<GRIDSIZE;j++)
362     {
363       int square = currentBoard.getSquare(i,j);
364       if(square != EMPTY)
365       {
366         drawToken(hdc,i,j,square);
367       }
368     }
369   }
370
371   SelectObject(hdc, hPenOld);
372   DeleteObject(hLinePen);
373 }
374
375 void drawToken(HDC hdc, int x, int y, int player) {
376   HPEN hPenOld;
377   HPEN hLinePen;
378   HBRUSH hBrushOld;
379   HBRUSH hBrush;
380   COLORREF qLineColor;
381
382   if(player == WHITE)
383   {
384     qLineColor = RGB(255, 255, 255);
385   }else if(player == BLACK)
386   {
387     qLineColor = RGB(0, 0, 0);
```

```
388     }else
389     {
390       return;
391     }
392
393     const int penThickness = 2;
394     hLinePen = CreatePen(PS_SOLID, penThickness, RGB(0, 0, 0));
395     hBrush = CreateSolidBrush(qLineColor);
396     hPenOld = (HPEN)SelectObject(hdc, hLinePen);
397     hBrushOld = (HBRUSH)SelectObject(hdc, hBrush);
398
399     // Get bounds
400     const int x0  = x*GRIDWIDTH + 2*penThickness;
401     const int x1  = (x + 1)*GRIDWIDTH - 2*penThickness;
402     const int y0  = y*GRIDWIDTH + 2*penThickness;
403     const int y1  = (y + 1)*GRIDWIDTH - 2*penThickness;
404
405     Ellipse(hdc, x0, y0, x1, y1);
406
407     SelectObject(hdc, hPenOld);
408     DeleteObject(hLinePen);
409     SelectObject(hdc, hBrushOld);
410     DeleteObject(hBrush);
411 }
```

3.1.2 Resource.h

```
1  //{{NO_DEPENDENCIES}}
2  // Microsoft Visual C++ generated include file.
3  // Used by Othello.rc
4  //
5
6  #define IDS_APP_TITLE       103
7
8  #define IDR_MAINFRAME       128
9  #define IDD_OTHELLO_DIALOG      102
10 #define IDD_ABOUTBOX        103
11 #define IDM_ABOUT           104
12 #define IDM_EXIT            105
13 #define IDM_AIFIRST         110
14 #define IDM_AISECOND        111
15 #define IDM_STRATA          201
16 #define IDM_STRATB          202
17 #define IDM_STRATC          203
18 #define IDM_STRATD          209
19 #define IDM_STRATE          210
20 #define IDM_NOPLAYER        204
21 #define IDM_LEARNING        205
22 #define IDI_OTHELLO         107
23 #define IDI_SMALL           108
24 #define IDC_OTHELLO         109
25
```

```
26 #define IDC_MYICON              2
27 #ifndef IDC_STATIC
28 #define IDC_STATIC              -1
29 #endif
30 // Next default values for new objects
31 //
32 #ifdef APSTUDIO_INVOKED
33 #ifndef APSTUDIO_READONLY_SYMBOLS
34
35 #define _APS_NO_MFC             130
36 #define _APS_NEXT_RESOURCE_VALUE    129
37 #define _APS_NEXT_COMMAND_VALUE     32771
38 #define _APS_NEXT_CONTROL_VALUE     1000
39 #define _APS_NEXT_SYMED_VALUE     110
40 #endif
41 #endif
```

3.1.3 Othello.h

```
1 #pragma once
2
3 #include "resource.h"
```

3.2 Command Line Interface

The command line interface is controlled by the othelloMain.cpp file.

3.2.1 othelloMain.cpp

```
1 #include<iostream>
2 #include "Game.h"
3 #include "Board.h"
4
5
6 using namespace std;
7
8 #define BOARD_THICKNESS 3
9
10 // Prints the board to stdout
11 void printBoard(Game *game);
12
13 int main(void)
14 {
15    int aiStrategy;
16    int humanPlayer;
17    int aiPlayer;
18    Game *game = new(Game);
19    while(true)
20    {
```

```
21      cout << "Select strategy type by typing a number and pressing enter" <<
            endl;
22      cout << "1: Minimax" << endl;
23      cout << "2: Minimax with Alpha-Beta pruning" << endl;
24      cout << "3: Minimax with Alpha-Beta pruning and Book Moves" << endl;
25      cout << "4: Random moves" << endl;
26      int choice;
27      cin >> choice;
28      switch (choice)
29      {
30      case 1:
31        aiStrategy = AI_STRATEGY_A;
32        break;
33      case 2:
34        aiStrategy = AI_STRATEGY_B;
35        break;
36      case 3:
37        aiStrategy = AI_STRATEGY_C;
38        break;
39      case 4:
40        aiStrategy = AI_STRATEGY_D;
41        break;
42      default:
43        cout << "Invalid selection. Terminating application.";
44        return 0;
45        break;
46      }
47      cout << "Select who plays first" << endl;
48      cout << "1: You play first" << endl;
49      cout << "2: Computer plays first" << endl;
50      cin >> choice;
51      switch (choice)
52      {
53      case 1:
54        humanPlayer = BLACK;
55        aiPlayer = WHITE;
56        break;
57      case 2:
58        aiPlayer = BLACK;
59        humanPlayer = WHITE;
60        break;
61      default:
62        cout << "Invalid selection. Terminating application.";
63        return 0;
64        break;
65      }
66      while(!game->isOverGame())
67      {
68        printBoard(game);
69        if(game->getCurrentPlayer() == humanPlayer) // Human turn to move
70        {
71          int x,y;
72          char xChar, yChar;
```

```
73          cout << "Enter a move (eg. A5) then press enter..." << endl;
74          cin >> xChar >> yChar;
75          x = xChar - 'A';
76          y = yChar - '1';
77          if(x>=0 && x<GRIDSIZE && y>=0 && y<GRIDSIZE)
78          {
79            if(game->isLegalMove(x,y))
80              game->makeMove(x,y);
81            else
82              cout << "Illegal move attempted" << endl;
83          }else
84          {
85            cout << "Invalid selection!" << endl;
86          }
87        }else // Computer turn to move
88        {
89          cout << "AI is moving..." << endl;
90          game->makeMoveAI(aiStrategy);
91        }
92      }
93      if(game->whoWins() == BLACK)
94        cout << "Game over. The winner is black!" << endl;
95      else if(game->whoWins() == WHITE)
96        cout << "Game over. The winner is white!" << endl;
97      else
98        cout << "Game over. It was a draw!" << endl;
99    }
100   return 0;
101 }
102
103 void printBoard(Game *game)
104 {
105   cout << "   ";
106   for(int i=0; i<GRIDSIZE; i++)
107   {
108     char letter = 'A' + i;
109     for(int j=0; j<BOARD_THICKNESS+1; j++){cout << letter;}
110     cout << " ";
111   }
112   cout << endl;
113   cout << "   ";
114   for (int i=0; i<GRIDSIZE; i++)
115   {
116     for(int j=0; j<BOARD_THICKNESS+1; j++) {cout << "-";}
117     cout << " ";
118   }
119   cout << endl;
120   Board currentBoard = game->getBoard();
121   for(int y=0; y<GRIDSIZE; y++)
122   {
123     for(int i=0; i<BOARD_THICKNESS; i++)
124     {
125       cout << (y+1) << " |";
```

```
126         for(int x=0; x<GRIDSIZE; x++)
127         {
128           if(currentBoard.getSquare(x,y) == BLACK)
129           {
130             for(int j=0; j<BOARD_THICKNESS+1; j++){cout << "B";}
131             cout << "|";
132           }else if(currentBoard.getSquare(x,y) == WHITE)
133           {
134             for(int j=0; j<BOARD_THICKNESS+1; j++){cout << "W";}
135             cout << "|";
136           }
137           else
138           {
139             for(int j=0; j<BOARD_THICKNESS+1; j++){cout << " ";}
140             cout << "|";
141           }
142         }
143         cout << endl;
144       }
145       cout << "    ";
146       for (int i=0; i<GRIDSIZE; i++)
147       {
148         for(int j=0; j<BOARD_THICKNESS+1; j++) {cout << "-";}
149         cout << " ";
150       }
151       cout << endl;
152     }
153     cout << endl;
154 }
```

3.3 Othello Game Back End

The back end was centrally controlled by the Game class defined in Game.h and Game.cpp. It made use of the Board class defined in Board.h and Board.cpp in order to maintain the game state.

3.3.1 Game.h

```
1  #pragma once
2  #include "stdafx.h"
3  #include "Board.h"
4  #include "othelloAI.h"
5
6  #define AI_STRATEGY_A 777
7  #define AI_STRATEGY_B 778
8  #define AI_STRATEGY_C 779
9  #define AI_STRATEGY_D 780
10 #define AI_STRATEGY_E 781
11
12
```

```
13 class Game
14 {
15   Board *board;
16   int currentPlayer;
17   OthelloAI *ai;
18   bool canCurrentPlayerMove();
19
20 public:
21   Game(void);
22   ~Game(void);
23
24   //
25   // FUNCTION: getBoard()
26   //
27   // PURPOSE: returns a copy of the Board for this game
28   Board getBoard();
29
30   // FUNCTION getCurrentPlayer()
31   //
32   // RETURN: the next player to make a move in the game.
33   // NOTE: currentPlayer is updated immediately if a player's
34   // turn must be skipped. Thus this will not return the a
35   // player that cannot move (except if a game is finished).
36   int getCurrentPlayer();
37
38   //
39   // FUNCTION: isLegalMove(int x, int y)
40   //
41   // PURPOSE: Determines if a move in the (x,y) square is legal for the
         current player.
42   bool isLegalMove(int x, int y);
43
44   //
45   // FUNCTION: isLegalMove(int x, int y)
46   //
47   // PURPOSE: Determines if a move in the (x,y) square is legal for the
         current player in Board someBoard.
48   static bool isLegalMove(int x, int y, Board *someBoard, int playerToMove)
         ;
49
50   //
51   // FUNCTION: isLegalMove(int x, int y)
52   //
53   // PURPOSE: Determines if a move in the (x,y) square is legal
54   // for the current player in the current board of game.
55   bool isLegalMove(int x, int y, int player);
56
57   //
58   // FUNCTION: makeMove(int x, int y)
59   //
60   // PURPOSE: makes a move in square (x,y) if it is legal and it is the
         users turn. This function does nothing otherwise.
61   void makeMove(int x, int y);
```

```
62
63    //
64    // FUNCTION: makeMoveAI()
65    //
66    // PURPOSE: Causes the AI to make a move in the current game
67    void makeMoveAI(int strategy);
68
69    // FUNCTION: Used to determine if a game has ended
70    // RETURN: true if the game is at a terminal stage and false otherwise
71    bool isOverGame();
72
73    // FUNCTION: isOverGame(Board *someBoard)
74    // PURPOSE: Used to determine if a board 'someBoard' is terminal (ie.
75    // nobody can move)
76    // RETURN: true if someBoard is at a terminal stage and false otherwise
77    static bool isTerminal(Board *someBoard);
78
79    // FUNCTION: Used to determine the winner of a game at its terminal stage
80    int whoWins();
81 };
```

3.3.2 Game.cpp

```
1 #include "stdafx.h"
2 #include "Game.h"
3 #include "OthelloAI.h"
4 #include <vector>
5
6 using std::vector;
7
8 // RETURN: WHITE if player is BLACK and BLACK otherwise
9 inline int otherPlayer(int player);
10 // Initialises the board to the starting position for the Othello game
11 void initialiseOthello(Board *board);
12
13 Game::Game(void)
14 {
15    currentPlayer = BLACK;
16    board = new(Board);
17    initialiseOthello(board);
18    ai = new(OthelloAI);
19 }
20
21
22 Game::~Game(void)
23 {
24    delete board;
25    delete ai;
26 }
27
28 int Game::getCurrentPlayer()
29 {
```

```
30    return currentPlayer;
31 }
32
33 bool Game::isLegalMove(int x, int y)
34 {
35    if(board->getSquare(x,y) != EMPTY)
36      return false;
37    if(x+2<GRIDSIZE)
38    {
39      if(board->getSquare(x+2,y) == currentPlayer && board->getSquare(x+1,y)
          == otherPlayer(currentPlayer))
40        return true;
41      if(y+2<GRIDSIZE)
42      {
43        if(board->getSquare(x+2,y+2) == currentPlayer && board->getSquare(x
            +1,y+1) == otherPlayer(currentPlayer))
44          return true;
45      }
46      if(y-2>=0)
47      {
48        if(board->getSquare(x+2,y-2) == currentPlayer && board->getSquare(x
            +1,y-1) == otherPlayer(currentPlayer))
49          return true;
50      }
51    }
52    if(x-2>=0)
53    {
54      if(board->getSquare(x-2,y) == currentPlayer && board->getSquare(x-1,y)
          == otherPlayer(currentPlayer))
55        return true;
56      if(y+2<GRIDSIZE)
57      {
58        if(board->getSquare(x-2,y+2) == currentPlayer && board->getSquare(x
            -1,y+1) == otherPlayer(currentPlayer))
59          return true;
60      }
61      if(y-2>=0)
62      {
63        if(board->getSquare(x-2,y-2) == currentPlayer && board->getSquare(x
            -1,y-1) == otherPlayer(currentPlayer))
64          return true;
65      }
66    }
67    if(y+2<GRIDSIZE)
68    {
69      if(board->getSquare(x,y+2) == currentPlayer && board->getSquare(x,y+1)
          == otherPlayer(currentPlayer))
70        return true;
71    }
72    if(y-2>=0)
73    {
74      if(board->getSquare(x,y-2) == currentPlayer && board->getSquare(x,y-1)
          == otherPlayer(currentPlayer))
```

```
75          return true;
76      }
77
78      return false;
79  }
80
81
82  Board Game::getBoard()
83  {
84      return *board;
85  }
86
87  void Game::makeMove(int x, int y)
88  {
89      if(isLegalMove(x,y))
90      {
91          board->insertToken(x,y,currentPlayer);
92          currentPlayer = otherPlayer(currentPlayer);
93          if(isOverGame())
94          {
95              // Do nothing
96          }
97          if(!canCurrentPlayerMove())
98          {
99              currentPlayer = otherPlayer(currentPlayer);
100         }
101     }
102 }
103
104 void Game::makeMoveAI(int strategy)
105 {
106     if(isOverGame())
107         return;
108     ai->setPlayer(currentPlayer);
109     while(true)
110     {
111         int aiMove = ai->getMove(board, strategy);
112         int aiMoveX = aiMove%GRIDSIZE;
113         int aiMoveY = aiMove/GRIDSIZE;
114         if(isLegalMove(aiMoveX,aiMoveY))
115         {
116             board->insertToken(aiMoveX,aiMoveY,currentPlayer);
117             currentPlayer = otherPlayer(currentPlayer);
118             break;
119         }else
120         {
121         }
122     }
123     if(isOverGame())
124     {
125         // Do nothing
126     }else if(!canCurrentPlayerMove())
127     {
```

```
128       currentPlayer = otherPlayer(currentPlayer);
129     }
130 }
131
132
133
134 inline int otherPlayer(int player)
135 {
136     if(player == BLACK)
137       return WHITE;
138     else
139       return BLACK;
140 }
141
142 void initialiseOthello(Board *board)
143 {
144     delete board;
145     board = new(Board);
146     board->changeToken(3,3,WHITE);
147     board->changeToken(4,4,WHITE);
148     board->changeToken(3,4,BLACK);
149     board->changeToken(4,3,BLACK);
150 }
151
152 bool Game::isLegalMove(int x, int y, Board *someBoard, int playerToMove)
153 {
154     if(someBoard->getSquare(x,y) != EMPTY)
155       return false;
156     if(x+2<GRIDSIZE)
157     {
158       if(someBoard->getSquare(x+2,y) == playerToMove && someBoard->getSquare(
              x+1,y) == otherPlayer(playerToMove))
159         return true;
160       if(y+2<GRIDSIZE)
161       {
162         if(someBoard->getSquare(x+2,y+2) == playerToMove && someBoard->
                getSquare(x+1,y+1) == otherPlayer(playerToMove))
163           return true;
164       }
165       if(y-2>=0)
166       {
167         if(someBoard->getSquare(x+2,y-2) == playerToMove && someBoard->
                getSquare(x+1,y-1) == otherPlayer(playerToMove))
168           return true;
169       }
170     }
171     if(x-2>=0)
172     {
173       if(someBoard->getSquare(x-2,y) == playerToMove && someBoard->getSquare(
              x-1,y) == otherPlayer(playerToMove))
174         return true;
175       if(y+2<GRIDSIZE)
176       {
```

```
177         if(someBoard->getSquare(x-2,y+2) == playerToMove && someBoard->
                getSquare(x-1,y+1) == otherPlayer(playerToMove))
178           return true;
179       }
180       if(y-2>=0)
181       {
182         if(someBoard->getSquare(x-2,y-2) == playerToMove && someBoard->
                getSquare(x-1,y-1) == otherPlayer(playerToMove))
183           return true;
184       }
185     }
186     if(y+2<GRIDSIZE)
187     {
188       if(someBoard->getSquare(x,y+2) == playerToMove && someBoard->getSquare(
              x,y+1) == otherPlayer(playerToMove))
189         return true;
190     }
191     if(y-2>=0)
192     {
193       if(someBoard->getSquare(x,y-2) == playerToMove && someBoard->getSquare(
              x,y-1) == otherPlayer(playerToMove))
194         return true;
195     }
196     return false;
197 }
198
199 bool Game::isLegalMove(int x, int y, int player)
200 {
201     if(board->getSquare(x,y) != EMPTY)
202       return false;
203     if(x+2<GRIDSIZE)
204     {
205       if(board->getSquare(x+2,y) == player && board->getSquare(x+1,y) ==
              otherPlayer(player))
206         return true;
207       if(y+2<GRIDSIZE)
208       {
209         if(board->getSquare(x+2,y+2) == player && board->getSquare(x+1,y+1)
                == otherPlayer(player))
210           return true;
211       }
212       if(y-2>=0)
213       {
214         if(board->getSquare(x+2,y-2) == player && board->getSquare(x+1,y-1)
                == otherPlayer(player))
215           return true;
216       }
217     }
218     if(x-2>=0)
219     {
220       if(board->getSquare(x-2,y) == player && board->getSquare(x-1,y) ==
              otherPlayer(player))
221         return true;
```

```
222       if(y+2<GRIDSIZE)
223       {
224         if(board->getSquare(x-2,y+2) == player && board->getSquare(x-1,y+1)
              == otherPlayer(player))
225           return true;
226       }
227       if(y-2>=0)
228       {
229         if(board->getSquare(x-2,y-2) == player && board->getSquare(x-1,y-1)
              == otherPlayer(player))
230           return true;
231       }
232     }
233     if(y+2<GRIDSIZE)
234     {
235       if(board->getSquare(x,y+2) == player && board->getSquare(x,y+1) ==
            otherPlayer(player))
236         return true;
237     }
238     if(y-2>=0)
239     {
240       if(board->getSquare(x,y-2) == player && board->getSquare(x,y-1) ==
            otherPlayer(player))
241         return true;
242     }
243     return false;
244 }
245
246 bool Game::canCurrentPlayerMove()
247 {
248   for(int i=0;i<GRIDSIZE;i++)
249     for(int j=0;j<GRIDSIZE;j++)
250       if(this->isLegalMove(i,j,currentPlayer))
251         return true;
252   return false;
253 }
254
255
256 bool Game::isOverGame()
257 {
258   for(int i=0;i<GRIDSIZE;i++)
259     for(int j=0;j<GRIDSIZE;j++)
260       if(this->isLegalMove(i,j,WHITE) || this->isLegalMove(i,j,BLACK))
261         return false;
262   return true;
263 }
264
265 bool Game::isTerminal(Board *someBoard)
266 {
267   for(int i=0;i<GRIDSIZE;i++)
268     for(int j=0;j<GRIDSIZE;j++)
269       if(isLegalMove(i,j,someBoard,WHITE) || isLegalMove(i,j,someBoard,
            BLACK))
```

```
270             return false;
271     return true;
272 }
273
274 int Game::whoWins()
275 {
276     int white = 0;
277     int black = 0;
278     for(int i=0;i<GRIDSIZE;i++)
279       for(int j=0;j<GRIDSIZE;j++)
280         if(board->getSquare(i,j) == WHITE)
281           white++;
282         else if(board->getSquare(i,j) == BLACK)
283           black++;
284     if(white > black)
285       return WHITE;
286     if(black > white)
287       return BLACK;
288     return EMPTY;
289 }
```

3.3.3 Board.h

```
1  #pragma once
2  #include "stdafx.h"
3
4  #define EMPTY 0
5  #define BLACK 1
6  #define WHITE -1
7  #define GRIDSIZE 8
8
9  class Board
10 {
11 private:
12     int boardArray[GRIDSIZE*GRIDSIZE];
13 public:
14
15     // Constructs a board with all cells initialised to EMPTY
16     Board(void);
17
18     // FUNCTION: Used to create a new board with one token added.
19     //
20     // RETURN: A new board with all squares the same as board
21     // but square (x,y) set to player
22     Board(Board *board, int x, int y, int player);
23
24     ~Board(void);
25
26     // Gets the value at square (x,y) in the board
27     int getSquare(int x, int y);
28
29     // Inserts a token of colour 'colour' into the board at square (x,y)
```

```
30    // and also makes the flips that occur in othello game after such
31    // an insert
32    void insertToken(int x, int y, int colour);
33
34    // Same as insertToken except it makes no flips.
35    void changeToken(int x, int y, int colour);
36
37    void changeBoard(Board *board, int x, int y, int player);
38 };
```

3.3.4 Board.cpp

```
1  #include "stdafx.h"
2  #include "Board.h"
3
4  // Flips all the pieces when a token of type currentPlayer is inserted into
5  // square (x,y).
6  void makeFlips(int x, int y, int insertedColour, Board *board);
7
8  // RETURN: WHITE if player is BLACK and BLACK otherwise
9  inline int otherColour(int colour);
10
11 Board::Board(void)
12 {
13    for(int i=0;i<GRIDSIZE;i++)
14    {
15       for(int j=0;j<GRIDSIZE;j++)
16       {
17          boardArray[i*GRIDSIZE+j] = EMPTY;
18       }
19    }
20 }
21
22 Board::Board(Board *board, int x, int y, int player)
23 {
24    for(int i = 0; i < GRIDSIZE*GRIDSIZE; i++)
25    {
26       this->boardArray[i] = board->boardArray[i];
27    }
28    this->insertToken(x,y,player);
29 }
30
31 Board::~Board(void)
32 {
33 }
34
35 void Board::insertToken(int x, int y, int colour)
36 {
37    boardArray[x + y*GRIDSIZE] = colour;
38    makeFlips(x,y,colour,this);
39 }
40
```

```
41 void Board:: changeToken(int x, int y, int colour)
42 {
43   boardArray[x + y*GRIDSIZE] = colour;
44 }
45
46 int Board::getSquare(int x, int y)
47 {
48   return boardArray[x + y*GRIDSIZE];
49 }
50
51
52 void makeFlips(int x, int y, int insertedColour, Board *board)
53 {
54   if(x+2<GRIDSIZE)
55   {
56     if(board->getSquare(x+2,y) == insertedColour && board->getSquare(x+1,y)
          == otherColour(insertedColour))
57       board->changeToken(x+1,y,insertedColour);
58     if(y+2<GRIDSIZE)
59     {
60       if(board->getSquare(x+2,y+2) == insertedColour && board->getSquare(x
            +1,y+1) == otherColour(insertedColour))
61         board->changeToken(x+1,y+1,insertedColour);
62     }
63     if(y-2>=0)
64     {
65       if(board->getSquare(x+2,y-2) == insertedColour && board->getSquare(x
            +1,y-1) == otherColour(insertedColour))
66         board->changeToken(x+1,y-1,insertedColour);
67     }
68   }
69   if(x-2>=0)
70   {
71     if(board->getSquare(x-2,y) == insertedColour && board->getSquare(x-1,y)
          == otherColour(insertedColour))
72       board->changeToken(x-1,y,insertedColour);
73     if(y+2<GRIDSIZE)
74     {
75       if(board->getSquare(x-2,y+2) == insertedColour && board->getSquare(x
            -1,y+1) == otherColour(insertedColour))
76         board->changeToken(x-1,y+1,insertedColour);
77     }
78     if(y-2>=0)
79     {
80       if(board->getSquare(x-2,y-2) == insertedColour && board->getSquare(x
            -1,y-1) == otherColour(insertedColour))
81         board->changeToken(x-1,y-1,insertedColour);
82     }
83   }
84   if(y+2<GRIDSIZE)
85   {
86     if(board->getSquare(x,y+2) == insertedColour && board->getSquare(x,y+1)
          == otherColour(insertedColour))
```

```
87          board->changeToken(x,y+1,insertedColour);
88      }
89      if(y-2>=0)
90      {
91          if(board->getSquare(x,y-2) == insertedColour && board->getSquare(x,y-1)
                == otherColour(insertedColour))
92              board->changeToken(x,y-1,insertedColour);
93      }
94  }
95
96  inline int otherColour(int colour)
97  {
98      if(colour == BLACK)
99          return WHITE;
100     else
101         return BLACK;
102 }
```

3.4 Artificial Intelligence

The artificial intelligence was controlled by the OthelloAI class defined in OthelloAI.h and Othello.cpp it also made use of some simple class defined in State.h/cpp and StateMove.h/cpp.

3.4.1 OthelloAI.h

```
1  #pragma once
2  #include "stdafx.h"
3  #include "Board.h"
4  #include "State.h"
5  #include <vector>
6  #include <hash_set>
7  #include "StateMove.h"
8  #include <exception>
9  #include <map>
10 #include <queue>
11
12 using std::vector; using std::exception; using std::multimap; using std::
       queue;using namespace std;
13 class OthelloAI
14 {
15 private:
16     int me;
17
18
19     multimap<State, int> book;
20
21     //hash_set<StateMove> book;
22
23     // Returns the first legal move found
```

```
24  int firstLegalMove(Board *board);
25
26  // FUNCTION: Determines a minimax score
27  //
28  // RETURN: The score for the node board using minimax algorithm to a
        depth of depth
29  int minimax(Board *board, int depth, int player);
30
31  // FUNCTION: Uses minimax search method to find a move
32  //
33  // RETURN: The move decided by minimax search algorithm
34  // to be the best (in the format (x+y*GRIDSIZE))
35  int minimaxSearch(Board *board);
36
37  // FUNCTION: Determines a negamax score
38  //
39  // RETURN: The score for the node board using negamax algorithm to a
        depth of depth
40  int negamax(Board *board, int depth, int playerToMove);
41
42  // FUNCTION: Uses negamax search method to find a move
43  //
44  // RETURN: The move decided by negamax search algorithm
45  // to be the best (in the format (x+y*GRIDSIZE))
46  int negamaxSearch(Board *board);
47
48  // PURPOSE: Uses minimax search with alpha beta pruning
49  // method and scouting optimization to find a best move
50  //
51  // RETURN: The move decided by minimax search algorithm
52  // to be the best (in the format 'x+y*GRIDSIZE')
53  int negascoutSearch(Board *board);
54
55  // FUNCTION: score(Board *board, int depth, int alpha, int beta, int
        player)
56  //
57  // PURPOSE: Determines a minimax score using alpha beta pruning and
        scouting
58  //
59  // RETURN: The score for the node board using minimax algorithm to a
        depth of depth
60  int negascout(Board *board, int depth, int alpha, int beta, int player);
61
62  // PURPOSE: Uses minimax search with alpha beta pruning
63  // method to find a move
64  //
65  // RETURN: The move decided by minimax search algorithm
66  // to be the best (in the format 'x+y*GRIDSIZE')
67  int alphabetaSearch(Board *board);
68
69  // PURPOSE: ONLY USE FOR LEARNING. Uses minimax search
70  // with alpha beta pruning and scouting to find a move with a
71  // very deep search tree.
```

```
72    //
73    // RETURN: The move decided by minimax search algorithm
74    // to be the best (in the format 'x+y*GRIDSIZE')
75    int deepSearch(Board *board);
76
77    // FUNCTION: alphabeta(Board *board, int depth, int alpha, int beta, int
          player)
78    //
79    // PURPOSE: Determines a minimax score using alpha beta pruning
80    //
81    // RETURN: The score for the node board using minimax algorithm to a
          depth of depth
82    int alphabeta(Board *board, int depth, int alpha, int beta, int player);
83
84
85
86    // PURPOSE: Opens the move book and stores in 'book' variable
87    // to be read in the findInMoveBook(Board *board, int playerToMove)
          function
88    void openBook();
89
90    // FUNCTION: findMoveInBook(Board *board, int playerToMove)
91    // PURPOSE: Used to find the move to to make in the current state
92    // defined by board and playerToMove.
93    // RETURN: The move to make as defined in the book
94    // THROWS: stateNotInBookException
95    int findMoveInBook(Board *board, int playerToMove);
96
97
98  public:
99
100   // Sets the 'me' field of the OthelloAI to player
101   void setPlayer(int player);
102
103   // Default AI with me set to BLACK
104   OthelloAI(void);
105   // AI with me to player
106   OthelloAI(int player);
107   ~OthelloAI(void);
108
109   // Used to find the move chosen by the AI for the currentBoard
110   // using the strategy 'strategy'
111   //
112   // RETURN: A move in the format (x+y*GRIDSIZE)
113   int getMove(Board *currentBoard, int strategy);
114
115 };
```

3.4.2 OthelloAI.cpp

```
1 #include "stdafx.h"
2 #include "OthelloAI.h"
```

```
3 #include "Board.h"
4 #include "Game.h"
5 #include "StateMove.h"
6 #include <time.h>
7 #include <queue>
8
9 #include <stdlib.h>
10
11 using namespace std;
12
13 #define INFINITY 999999
14 #define MINIMAX_DEPTH 4
15 #define ALPHABETA_DEPTH 5
16 #define DEEP_DEPTH 10
17 #define SCOUT_DEPTH 8
18
19 #define BOOK "book_moves.dat"
20
21 // RETURN: WHITE if player is BLACK and BLACK otherwise
22 inline int otherPlayer(int player);
23
24 // FUNCTION: scoreBoardAdvanced(Board *board, player)
25 // PURPOSE: An advanced heuristic function for scoring
26 // a Board board from the perspective of player.
27 int scoreBoardAdvanced(Board *board, int player);
28
29 // PURPOSE: Used as a simple heuristic scoring function
30 //
31 // RETURN: Sum over all tokens beloning to 'me' minus the sum over all
      tokens belonging to other player
32 int scoreBoardSimple(Board *board, int player);
33
34 // FUNCTION: randomLegalMove(Board *board)
35 // RETURN: a random legal move in the Board
36 // referenced by 'board' for player 'me'
37 int randomLegalMove(Board *board, int me);
38
39
40 // FUNCTION: appendToBook(StateMove sm)
41 // PURPOSE: Saves the state move sm in the book at the end
42 void appendToBook(StateMove sm);
43
44 // RETURN: number of possible moves for player 'me' in
45 // the Board 'board'
46 int possibleMoves(Board *board, int me);
47
48 class stateNotInBookException: public exception
49 {
50   virtual const char* what() const throw()
51   {
52     return "State not in book";
53   }
54 } StateNotInBookException;
```

```
55
56 // BoardOrder class is used only for
57 // expanding children in good order in
58 // negascout algorithm
59 class BoardOrder
60 {
61   friend bool operator<(BoardOrder a, BoardOrder b);
62 public:
63   ~BoardOrder(void){}
64   BoardOrder(void){}
65   BoardOrder(int priority, Board board)
66   {
67     this->priority = priority;
68     this->board = board;
69   }
70   int priority;
71   Board board;
72 };
73
74 // Redefine less than operator for use in
75 // priority queue
76 bool operator <(BoardOrder a, BoardOrder b)
77 {
78   return a.priority < b.priority;
79 }
80
81 OthelloAI::OthelloAI(void)
82 {
83   openBook();
84   me = BLACK;
85 }
86
87 OthelloAI::OthelloAI(int player)
88 {
89   openBook();
90   me = player;
91 }
92
93 void OthelloAI::setPlayer(int player)
94 {
95   me = player;
96 }
97
98 OthelloAI::~OthelloAI(void)
99 {
100
101 }
102
103 int OthelloAI::getMove(Board *currentBoard, int strategy)
104 {
105   int move = 0;
106   switch(strategy)
107   {
```

```
108     case AI_STRATEGY_A:
109       move = minimaxSearch(currentBoard);
110     case AI_STRATEGY_B:
111       move = alphabetaSearch(currentBoard);
112       break;
113     case AI_STRATEGY_C:
114       try // IF: current state is in book
115       {
116         move = findMoveInBook(currentBoard,me);
117       }catch (stateNotInBookException e) // ELSE:
118       {
119         move = negascoutSearch(currentBoard);
120       }
121       break;
122     case AI_STRATEGY_D: // Random moving player
123       move =  randomLegalMove(currentBoard,me);
124       break;
125     case AI_STRATEGY_E: // Learning phase
126       try // IF: current state is in book
127       {
128         move = findMoveInBook(currentBoard,me);
129       }catch (stateNotInBookException e) // ELSE:
130       {
131         // Make a move using minimax with deep alphabeta search
132         move = deepSearch(currentBoard);
133         // Record that move in book
134         appendToBook(StateMove(currentBoard, me, move));
135       }
136       break;
137     }
138     return move;
139 }
140
141 int OthelloAI::negascoutSearch(Board *board)
142 {
143     int bestMoveScore = -2*INFINITY;
144     int bestMove;
145     int score;
146
147     for (int move=0; move < GRIDSIZE*GRIDSIZE; move++)
148     {
149       if(Game::isLegalMove(move%GRIDSIZE,move/GRIDSIZE,board,me))
150       {
151         Board moveBoard (board, move%GRIDSIZE, move/GRIDSIZE, me);
152         score = -negascout(&moveBoard, SCOUT_DEPTH, -2*INFINITY, -
              bestMoveScore, otherPlayer(me));
153         if(score > bestMoveScore)
154         {
155           bestMove = move;
156           bestMoveScore = score;
157         }
158       }
159     }
```

```
160     return bestMove;
161 }
162
163 int OthelloAI::negascout(Board *board, int depth, int alpha, int beta, int
        player)
164 {
165     if (depth <= 0 || Game::isTerminal(board))
166     {
167         int score = scoreBoardAdvanced(board,player);
168         return score;
169     }
170     priority_queue<BoardOrder> children;
171     bool cantMove = true;
172
173     // Queue children of state
174     for(int i=0;i<GRIDSIZE*GRIDSIZE;i++)
175     {
176         int x = i%GRIDSIZE;
177         int y = i/GRIDSIZE;
178         if(Game::isLegalMove(x,y,board,player))
179         {
180             cantMove = false;
181             Board child (board,x,y,player);
182             // Prioritise depending on whether it is min turn or
183             // max turn
184             int priority;
185             if(player == me)
186                 priority = scoreBoardSimple(&child,me);
187             else
188                 priority = -scoreBoardSimple(&child,me);
189             children.push(BoardOrder (priority, child));
190         }
191     }
192     if(cantMove)
193     {
194         // Child is same node
195         children.push(BoardOrder (0, *board));
196         depth++; // Increment depth for consistency with other algorithms
197     }
198
199     bool firstChild = true;
200     int b = beta; // scouting variable
201     while(children.size() > 0)
202     {
203         Board child = (children.top().board);
204         children.pop();
205         int score = -negascout(&child, depth-1, -b, -alpha, otherPlayer(player)
                );
206         if(alpha < score && score < beta && !firstChild) // Perform full alpha-
                beta search
207             score = -negascout(&child, depth - 1, -beta, -alpha, otherPlayer(
                    player));
208         firstChild = false;
```

```
209       alpha = max(alpha, score);
210       if(beta <= alpha)
211         return alpha;
212       b = alpha + 1;
213     }
214
215     return alpha;
216   }
217
218   int OthelloAI::minimaxSearch(Board *board)
219   {
220     int bestMoveScore = -2*INFINITY;
221     int bestMove;
222     int score;
223     for (int move=0; move < GRIDSIZE*GRIDSIZE; move++)
224     {
225       if(Game::isLegalMove(move%GRIDSIZE,move/GRIDSIZE,board,me))
226       {
227         Board moveBoard (board, move%GRIDSIZE, move/GRIDSIZE, me);
228         score = minimax(&moveBoard, MINIMAX_DEPTH, otherPlayer(me));
229         if(score > bestMoveScore)
230         {
231           bestMove = move;
232           bestMoveScore = score;
233         }
234       }
235     }
236     return bestMove;
237   }
238
239   int OthelloAI::minimax(Board *board, int depth, int player)
240   {
241     int score;
242     if (depth <= 0 || Game::isTerminal(board))
243       return scoreBoardAdvanced(board,me);
244     bool cantMove = true;
245     if(player == me)
246     {
247       score = -2*INFINITY;
248       for(int i=0;i<GRIDSIZE;i++)
249       {
250         for(int j=0;j<GRIDSIZE;j++)
251         {
252           if(Game::isLegalMove(i,j,board,player))
253           {
254             Board child (board, i, j, player);
255             cantMove = false;
256             score = max(score, minimax(&child, depth-1, otherPlayer(player)))
                      ;
257           }
258         }
259       }
260       if(cantMove) // Child node is same node
```

```
261         score = minimax(board, depth, otherPlayer(player));
262     }else
263     {
264       score = 2*INFINITY;
265       for(int i=0;i<GRIDSIZE;i++)
266       {
267         for(int j=0;j<GRIDSIZE;j++)
268         {
269           if(Game::isLegalMove(i,j,board,player))
270           {
271             Board child (board, i, j, player);
272             cantMove = false;
273             score = min(score, minimax(&child, depth-1, otherPlayer(player)))
                  ;
274           }
275         }
276       }
277       if(cantMove) // Child node is same node
278         score = minimax(board, depth, otherPlayer(player));
279     }
280     return score;
281 }
282
283 int OthelloAI::alphabetaSearch(Board *board)
284 {
285     int bestMoveScore = -2*INFINITY;
286     int bestMove;
287     int alphabetaScore;
288     for (int move=0; move < GRIDSIZE*GRIDSIZE; move++)
289     {
290       if(Game::isLegalMove(move%GRIDSIZE,move/GRIDSIZE,board,me))
291       {
292         Board moveBoard (board, move%GRIDSIZE, move/GRIDSIZE, me);
293         alphabetaScore = alphabeta(&moveBoard, ALPHABETA_DEPTH, bestMoveScore
              , +INFINITY, otherPlayer(me));
294         if(alphabetaScore > bestMoveScore)
295         {
296           bestMove = move;
297           bestMoveScore = alphabetaScore;
298         }
299       }
300     }
301     return bestMove;
302 }
303
304 int OthelloAI::alphabeta(Board *board, int depth, int alpha, int beta, int
      player)
305 {
306     if (depth <= 0 || Game::isTerminal(board))
307       return scoreBoardAdvanced(board,me);
308     bool cantMove = true;
309     if(player == me)
310     {
```

```
311       // Queue children of state
312       for(int i=0;i<GRIDSIZE*GRIDSIZE;i++)
313       {
314         int x = i%GRIDSIZE;
315         int y = i/GRIDSIZE;
316         if(Game::isLegalMove(x,y,board,player))
317         {
318           cantMove = false;
319           alpha = max(alpha, alphabeta(&Board(board,x,y,player), depth-1,
                alpha, beta, otherPlayer(player)));
320           if(beta <= alpha)
321             return alpha;
322         }
323       }
324       if(cantMove) // Child node is same node
325         alpha = max(alpha, alphabeta(board, depth, alpha, beta, otherPlayer(
              player)));
326       return alpha;
327     }else
328     {
329       // Queue children of state
330       for(int i=0;i<GRIDSIZE*GRIDSIZE;i++)
331       {
332         int x = i%GRIDSIZE;
333         int y = i/GRIDSIZE;
334         if(Game::isLegalMove(x,y,board,player))
335         {
336           cantMove = false;
337           beta = min(beta, alphabeta(&Board(board,x,y,player), depth-1, alpha
                , beta, otherPlayer(player)));
338           if(beta <= alpha)
339             return beta;
340         }
341       }
342       if(cantMove) // Child node is same node
343         beta = min(beta, alphabeta(board, depth, alpha, beta, otherPlayer(
              player)));
344       return beta;
345     }
346 }
347
348 int OthelloAI::negamaxSearch(Board *board)
349 {
350   int bestMoveScore = -2*INFINITY;
351   int bestMove;
352   int score;
353   for (int move=0; move < GRIDSIZE*GRIDSIZE; move++)
354   {
355     if(Game::isLegalMove(move%GRIDSIZE,move/GRIDSIZE,board,me))
356     {
357       Board moveBoard (board, move%GRIDSIZE, move/GRIDSIZE, me);
358       score = negamax(&moveBoard, MINIMAX_DEPTH, otherPlayer(me));
359       if(score > bestMoveScore)
```

```
360         {
361           bestMove = move;
362           bestMoveScore = score;
363         }
364       }
365     }
366     return bestMove;
367 }
368
369 int OthelloAI::negamax(Board *board, int depth, int playerToMove)
370 {
371     Board *child;
372     if(depth <= 0 || Game::isTerminal(board))
373       return scoreBoardAdvanced(board,otherPlayer(playerToMove));
374     int alpha = INFINITY;
375     bool cantMove = true;
376     for(int i=0; i<GRIDSIZE;i++)
377     {
378       for(int j=0; j<GRIDSIZE;j++)
379       {
380         if(Game::isLegalMove(i,j,board,playerToMove))
381         {
382           Board child (board, i, j, playerToMove);
383           cantMove = false;
384           alpha = min(alpha, -negamax(&child, depth-1, otherPlayer(
                playerToMove)));
385         }
386       }
387     }
388     if(cantMove) // child is the same as board if you skip a turn
389       alpha = min(alpha, -negamax(board, depth, otherPlayer(playerToMove)));
390     return alpha;
391 }
392
393 int randomLegalMove(Board *board, int me)
394 {
395     srand(time(NULL));
396     while(true)
397     {
398       int move = rand()%64;
399       if (Game::isLegalMove(move%GRIDSIZE, move/GRIDSIZE, board, me))
400         return move;
401     }
402 }
403
404 int OthelloAI::firstLegalMove(Board *board)
405 {
406     for(int i=0;i<GRIDSIZE;i++)
407       for(int j=0;j<GRIDSIZE;j++)
408         if(Game::isLegalMove(i,j,board,me))
409           return (i+j*GRIDSIZE);
410 }
411
```

```
412 int scoreBoardSimple(Board *board, int player)
413 {
414   int score = 0;
415   for(int i=0;i<GRIDSIZE;i++)
416   {
417     for(int j=0;j<GRIDSIZE;j++)
418     {
419       if(board->getSquare(i,j) == player)
420         score++;
421       else if(board->getSquare(i,j) == otherPlayer(player))
422         score--;
423     }
424
425   }
426   return score;
427 }
428
429 int scoreBoardAdvanced(Board *board, int player)
430 {
431   int scoreSimple = scoreBoardSimple(board,player);
432
433   if(Game::isTerminal(board))
434   {
435     // player wins
436     if(scoreSimple > 0)
437       return INFINITY;
438     // player loses
439     if(scoreSimple < 0)
440       return -INFINITY;
441     // match was a draw (draw is more desirable
442     // than a loss but still less desirable than
443     // any other state)
444     else
445       return -INFINITY/2;
446   }
447
448   // Add 10 points for every square owned by player and
449   // subtract 10 for every square owned by other player
450   int score = 10*scoreSimple;
451
452   // Add 5 points for every legal move and subtract
453   // 5 for every legal move of opponent
454   for(int i=0;i<GRIDSIZE;i++)
455   {
456     for(int j=0;j<GRIDSIZE;j++)
457     {
458
459       if(Game::isLegalMove(i,j,board,player))
460       {
461         score += 5;
462       }
463       if(Game::isLegalMove(i,j,board,otherPlayer(player)))
464       {
```

```
465           score -= 5;
466         }
467       }
468     }
469
470     // Add 50 points for every corner owned by player
471     // and subtract 50 points for every corner owned
472     // by other player
473     int cornerBonus = 0;
474     int corner = board->getSquare(0,0);
475     if(corner == player)
476       cornerBonus += 50;
477     else if(corner == otherPlayer(player))
478       cornerBonus -= 50;
479     corner = board->getSquare(0,7);
480     if(corner == player)
481       cornerBonus += 50;
482     else if(corner == otherPlayer(player))
483       cornerBonus -= 50;
484     corner = board->getSquare(7,0);
485     if(corner == player)
486       cornerBonus += 50;
487     else if(corner == otherPlayer(player))
488       cornerBonus -= 50;
489     corner = board->getSquare(7,7);
490     if(corner == player)
491       cornerBonus += 50;
492     else if(corner == otherPlayer(player))
493       cornerBonus -= 50;
494     return score + cornerBonus;
495 }
496
497 inline int otherPlayer(int player)
498 {
499
500     if(player == BLACK)
501       return WHITE;
502     else
503       return BLACK;
504 }
505
506
507 void appendToBook(StateMove sm)
508 {
509     ofstream bookfile;
510     bookfile.open(BOOK, ofstream::out | ofstream::app);
511     bookfile << sm.stateFirstHalf << " " << sm.stateSecondHalf << " " << sm.
          player << " " << sm.move << endl;
512     bookfile.close();
513 }
514
515 int OthelloAI::findMoveInBook(Board *board, int playerToMove)
516 {
```

```
517   int move = 0;
518   if(book.count(State (board, playerToMove)) > 0)
519     move = book.lower_bound(State (board, playerToMove))->second;
520   else
521     throw StateNotInBookException;
522   return move;
523 }
524
525 void OthelloAI::openBook()
526 {
527   unsigned long long firstHalf, secondHalf;
528   int player, move;
529   ifstream bookfile (BOOK);
530   if (bookfile.is_open())
531   {
532     while ( bookfile.good() )
533     {
534       bookfile >> firstHalf >> secondHalf >> player >> move;
535       book.insert(pair<State, int> (State (firstHalf, secondHalf, player),
              move));
536     }
537     bookfile.close();
538   }
539 }
540
541 int OthelloAI::deepSearch(Board *board)
542 {
543   int bestMoveScore = -2*INFINITY;
544   int bestMove;
545   int score;
546   for (int move=0; move < GRIDSIZE*GRIDSIZE; move++)
547   {
548     if(Game::isLegalMove(move%GRIDSIZE,move/GRIDSIZE,board,me))
549     {
550       Board moveBoard (board, move%GRIDSIZE, move/GRIDSIZE, me);
551       score = -negascout(&moveBoard, DEEP_DEPTH, -2*INFINITY, -
              bestMoveScore, otherPlayer(me));
552       if(score > bestMoveScore)
553       {
554         bestMove = move;
555         bestMoveScore = score;
556       }
557     }
558   }
559   return bestMove;
560 }
561
562 int possibleMoves(Board *board, int me)
563 {
564   int moves;
565   for(int i=0;i<GRIDSIZE;i++)
566   {
567     for(int j=0;j<GRIDSIZE;j++)
```

```
568     {
569       if(Game::isLegalMove(i,j,board,me))
570         moves++;
571     }
572   }
573   return moves;
574 }
```

3.4.3 State.h

```
1 #pragma once
2 #include "stdafx.h"
3 #include "Board.h"
4
5
6 class State
7 {
8   friend bool operator<(State a, State b);
9   friend bool operator==(State a, State b);
10 public:
11   State(void);
12
13   // RETURN: true if these states are equal and false otherwise
14   bool isEqual(State *other);
15
16   // RETURN: true if the state is the same as that defined by
17   // the parameters
18   bool isEqual(Board *board, int player);
19
20   // Construct a state defined by the parameters
21   State(Board *board, int player);
22
23   // Construct a state explicitly from the field values
24   State(unsigned long long first, unsigned long long second, unsigned long
       long player);
25
26   ~State(void);
27
28   // RETURN: The hash code for the state
29   unsigned long long getHash() const;
30
31   unsigned long long hash;
32   unsigned long long firstHalf;
33   unsigned long long movingPlayer;
34   unsigned long long secondHalf;
35 };
```

3.4.4 State.cpp

```
1 #include "stdafx.h"
```

```
2 #include "State.h"
3
4 // RETURN: x to the power i
5 inline unsigned long long power(int x, int i)
6 {
7   unsigned long long answer = 1;
8   for(int counter=0;counter<i;counter++) answer *= x;
9   return answer;
10 }
11
12 // RETURN: Unique modulo 3 value of square
13 inline unsigned long long value(int square);
14
15 State::State(void)
16 {
17 }
18
19
20 State::~State(void)
21 {
22 }
23
24 bool State::isEqual(Board *board, int player)
25 {
26   unsigned long long first = 0;
27   unsigned long long second = 0;
28   for(unsigned long long i=0; i<32 ;i++)
29   {
30     first += power(3,i)*value(board->getSquare(i%GRIDSIZE,i/GRIDSIZE));
31   }
32   for(unsigned long long i=32; i<64 ;i++)
33   {
34     second += power(3,i)*value(board->getSquare(i%GRIDSIZE,i/GRIDSIZE));
35   }
36   return (this->firstHalf == first && this->secondHalf == second && this->
       movingPlayer == player);
37 }
38
39 bool State::isEqual(State *other)
40 {
41   return (this->firstHalf == other->firstHalf && this->secondHalf == other
       ->secondHalf && this->movingPlayer == other->movingPlayer);
42 }
43
44 State::State(unsigned long long first, unsigned long long second, unsigned
     long long player)
45 {
46   firstHalf = first;
47   secondHalf = second;
48   movingPlayer = player;
49   hash = getHash();
50 }
51
```

```
52 State::State(Board *board, int player)
53 {
54   firstHalf = 0;
55   secondHalf = 0;
56   for(unsigned long long i=0; i<32 ;i++)
57   {
58     firstHalf += power(3,i)*value(board->getSquare(i%GRIDSIZE,i/GRIDSIZE));
59   }
60   for(unsigned long long i=32; i<64 ;i++)
61   {
62     secondHalf += power(3,i)*value(board->getSquare(i%GRIDSIZE,i/GRIDSIZE))
         ;
63   }
64   movingPlayer = player;
65   hash = getHash();
66 }
67
68 unsigned long long State::getHash() const
69 {
70   return firstHalf*secondHalf + movingPlayer;
71 }
72
73 inline unsigned long long value(int square)
74 {
75   if(square == WHITE)
76     return 2;
77   if(square == BLACK)
78     return 1;
79   return 0;
80 }
81
82 // Overwrite comparison operator 'less' for two states
83 bool operator<(State a, State b)
84 {
85   return a.getHash() < b.getHash();
86 }
87
88 bool operator==(const State a, const State b)
89 {
90    return (a.firstHalf == b.firstHalf && a.secondHalf == b.secondHalf && a.
       movingPlayer == b.movingPlayer);
91 }
```

3.4.5 StateMove.h

```
1 #pragma once
2 #include "stdafx.h"
3 #include "Board.h"
4
5
6 class StateMove
7 {
```

```
8  public:
9    StateMove(void);
10
11   // Constructs a StateMove by explicitly specifying all its field values
12   StateMove(unsigned long long firstHalf, unsigned long long secondHalf,
         int playerToMove, int moveMade);
13
14   // Constructs a StateMove with its properties defined by the parameters
15   StateMove(Board *board, int movingPlayer, int moveTo);
16
17   ~StateMove(void);
18
19   unsigned long long stateFirstHalf;
20   unsigned long long stateSecondHalf;
21   int player;
22   // A move (in the format 'x+y*GRIDSIZE')
23   int move;
24
25   // RETURN: true iff this state and move is equal to other state and move
26   bool isEqual(StateMove *other);
27
28   // RETURN: true iff this state is equal to the state defined by
29   // board and movingPlayer
30   bool isEqualState(Board *board, int movingPlayer);
31 };
```

3.4.6 StateMove.cpp

```
1  #include "stdafx.h"
2  #include "StateMove.h"
3  #include "Board.h"
4
5  // RETURN: x to the power i
6  inline unsigned long long power(int x, int i)
7  {
8    unsigned long long answer = 1;
9    for(int counter=0;counter<i;counter++) answer *= x;
10   return answer;
11 }
12
13 inline unsigned long long value(int square);
14
15 StateMove::StateMove(Board *board, int movingPlayer, int moveTo)
16 {
17   stateFirstHalf = 0;
18   stateSecondHalf = 0;
19   for(unsigned long long i=0; i<32 ;i++)
20   {
21     stateFirstHalf += power(3,i)*value(board->getSquare(i%GRIDSIZE,i/
         GRIDSIZE));
22   }
23   for(unsigned long long i=32; i<64 ;i++)
```

```
24    {
25      stateSecondHalf += power(3,i)*value(board->getSquare(i%GRIDSIZE,i/
            GRIDSIZE));
26    }
27    player = movingPlayer;
28    move = moveTo;
29  }
30
31  StateMove::StateMove(unsigned long long firstHalf, unsigned long long
        secondHalf, int playerToMove, int moveMade)
32  {
33    stateFirstHalf = firstHalf;
34    stateSecondHalf = secondHalf;
35    player = playerToMove;
36    move = moveMade;
37  }
38
39  StateMove::StateMove(void)
40  {
41
42  }
43
44
45  StateMove::~StateMove(void)
46  {
47  }
48
49
50  bool StateMove::isEqual(StateMove *other)
51  {
52    return (this->stateFirstHalf == other->stateFirstHalf && this->
          stateSecondHalf == other->stateSecondHalf&& this->move == other->move
          );
53  }
54
55  bool StateMove::isEqualState(Board *board, int movingPlayer)
56  {
57    unsigned long long newStateFirstHalf, newStateSecondHalf;
58
59    newStateFirstHalf = 0;
60    newStateSecondHalf = 0;
61
62    for(unsigned long long i=0; i<32 ;i++)
63    {
64      newStateFirstHalf += power(3,i)*value(board->getSquare(i%GRIDSIZE,i/
            GRIDSIZE));
65    }
66    for(unsigned long long i=32; i<64 ;i++)
67    {
68      newStateSecondHalf += power(3,i)*value(board->getSquare(i%GRIDSIZE,i/
            GRIDSIZE));
69    }
```

```
70    return (newStateFirstHalf == stateFirstHalf && newStateSecondHalf ==
          stateSecondHalf && movingPlayer == player);
71  }
72
73  inline unsigned long long value(int square)
74  {
75    if(square == WHITE)
76      return 2;
77    if(square == BLACK)
78      return 1;
79    return 0;
80  }
```

Appendix A

Instructions: Compiling and Running

These programs were developed using Microsoft Visual Studio. The zipped folder contains two folders one named 'Othello' and the other 'othelloCmd'. These contain all the same source files except that 'Othello' uses a Win32 based graphical user interface, while 'othelloCmd' uses only C++ standard library functions and runs through the command line. Both projects can be opened as solutions (using the .sln file in the root folder) in Visual Studio and compiled through the 'Build' menu. Also both projects have the **latest versions pre-compiled as windows executable files (.exe) in the folders titled 'Release'**. If neither of these are possible then read the following section for further instructions.

A.1 Alternatives

If the appropriate Win32 libraries are installed but Visual Studio is not it should be possible to compile the GUI version by compiling all source files in the directory 'Othello\Othello'. Try something like:

```
1 cd Othello\Othello
2 g++ -fpermissive *.cpp
3 .\a.out
```

If it is still not possible to use the GUI version then try compiling the command line version from source code. On a linux machine this can be done using the following commands:

```
1 cd othelloCmd/othelloCmd
2 g++ -fpermissive *.cpp
3 ./a.out
```

Note: the '-fpermissive' flag must be used to avoid some error that did not occur in Visual Studio. This error does not cause any issues to the actual game.

Appendix B

GUI: Using the interface

Note: The black player always moves first

Important: When you click in a square to make a legal move the screen will NOT update until it is your turn to move again. There may be some delay (up to potentially 1 minute, but this would be quite extreme) before the AI chooses its move especially in the case where you had to skip a turn (because you could not move). This game has been tested a great deal and there do not appear to be any bugs so be patient when waiting for the screen to update.

When playing in the GUI version you can restart a game at any time by selecting 'AI plays first' or 'Human plays first' in the 'Restart options' menu. You may also change the strategy of the opponent at any time by selecting one of the strategies from the 'Strategy options' menu at any point while playing.