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

```
int OthelloAI::minimax(Board *board, int depth, int player)
{
  int score;
  if (depth <= 0 || Game::isTerminal(board))
    return scoreBoardAdvanced(board,me);
  bool cantMove = true;
  if(player == me)
  {
    score = -2*INFINITY;
    for(int i=0;i<GRIDSIZE;i++)
    {
      for(int j=0;j<GRIDSIZE;j++)
      {
        if(Game::isLegalMove(i,j,board,player))
        {
          Board child (board, i, j, player);
          cantMove = false;
          score = max(score, minimax(&child, depth-1, otherPlayer(player)));
        }
      }
    }
    if(cantMove) // Child node is same node
      score = minimax(board, depth, otherPlayer(player));
  }else
  {
    score = 2*INFINITY;
    for(int i=0;i<GRIDSIZE;i++)
    {
      for(int j=0;j<GRIDSIZE;j++)
      {
        if(Game::isLegalMove(i,j,board,player))
        {
          Board child (board, i, j, player);
          cantMove = false;
          score = min(score, minimax(&child, depth-1, otherPlayer(player)));
        }
      }
    }
    if(cantMove) // Child node is same node
      score = minimax(board, depth, otherPlayer(player));
  }
  return score;
}
```

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

```
int OthelloAI::alphabeta(Board *board, int depth, int alpha, int beta, int player)
{
  if (depth <= 0 || Game::isTerminal(board))
    return scoreBoardAdvanced(board,me);
  bool cantMove = true;
  if(player == me)
  {
    // Queue children of state
    for(int i=0;i<GRIDSIZE*GRIDSIZE;i++)
    {
      int x = i%GRIDSIZE;
      int y = i/GRIDSIZE;
      if(Game::isLegalMove(x,y,board,player))
      {
        cantMove = false;
        alpha = max(alpha, alphabeta(&Board(board,x,y,player), depth-1, alpha, beta, otherPlayer(player)));
        if(beta <= alpha)
          return alpha;
      }
    }
    if(cantMove) // Child node is same node
      alpha = max(alpha, alphabeta(board, depth, alpha, beta, otherPlayer(player)));
    return alpha;
  }else
  {
    // Queue children of state
    for(int i=0;i<GRIDSIZE*GRIDSIZE;i++)
    {
      int x = i%GRIDSIZE;
      int y = i/GRIDSIZE;
      if(Game::isLegalMove(x,y,board,player))
      {
        cantMove = false;
        beta = min(beta, alphabeta(&Board(board,x,y,player), depth-1, alpha, beta, otherPlayer(player)));
        if(beta <= alpha)
          return beta;
      }
    }
    if(cantMove) // Child node is same node
      beta = min(beta, alphabeta(board, depth, alpha, beta, otherPlayer(player)));
    return beta;
  }
}
```

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

```
int OthelloAI::negascout(Board *board, int depth, int alpha, int beta, int player)
{
  if (depth <= 0 || Game::isTerminal(board))
  {
    int score = scoreBoardAdvanced(board, player);
    return score;
  }
  priority_queue<BoardOrder> children;
  bool cantMove = true;

```

```cpp
  // Queue children of state
  for(int i=0;i<GRIDSIZE*GRIDSIZE;i++)
  {
    int x = i%GRIDSIZE;
    int y = i/GRIDSIZE;
    if(Game::isLegalMove(x,y,board,player))
    {
      cantMove = false;
      Board *child = &Board(board,x,y,player);
      // Prioritise depending on whether it is min turn or
      // max turn
      int priority;
      if(player == me)
        priority = scoreBoardSimple(child,me);
      else
        priority = -scoreBoardSimple(child,me);
      children.push(BoardOrder (priority, *child));
    }
  }
  if(cantMove)
  {
    // Child is same node
    children.push(BoardOrder (0, *board));
    depth++; // Increment depth for consistency with other algorithms
  }

  bool firstChild = true;
  int b = beta; // scouting variable
  while(children.size() > 0)
  {
    Board child = (children.top().board);
    children.pop();
    int score = -negascout(&child, depth-1, -b, -alpha, otherPlayer(player));
    if(alpha < score && score < beta && !firstChild) // Perform full alpha-beta search
      score = -negascout(&child, depth - 1, -beta, -alpha, otherPlayer(player));
    firstChild = false;
    alpha = max(alpha, score);
    if(beta <= alpha)
      return alpha;
    b = alpha + 1;
  }

  return alpha;
}
```

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

```
int scoreBoardAdvanced(Board *board, int player)
{
  int scoreSimple = scoreBoardSimple(board,player);

  if(Game::isTerminal(board))
  {
    // player wins
    if(scoreSimple > 0)
      return INFINITY;
    // player loses
    if(scoreSimple < 0)
      return -INFINITY;
    // match was a draw (draw is more desirable
    // than a loss but still less desirable than
    // any other state)
    else
      return -INFINITY/2;
  }

  // Add 10 points for every square owned by player and
  // subtract 10 for every square owned by other player
  int score = 10*scoreSimple;

  // Add 5 points for every legal move and subtract
  // 5 for every legal move of opponent
  for(int i=0;i<GRIDSIZE;i++)
  {
    for(int j=0;j<GRIDSIZE;j++)
    {

      if(Game::isLegalMove(i,j,board,player))
      {
        score += 5;
      }
      if(Game::isLegalMove(i,j,board,otherPlayer(player)))
      {
        score -= 5;
      }
    }
  }

  // Add 50 points for every corner owned by player
  // and subtract 50 points for every corner owned
  // by other player
  int cornerBonus = 0;
  int corner = board->getSquare(0,0);
  if(corner == player)
    cornerBonus += 50;
  else if(corner == otherPlayer(player))
    cornerBonus -= 50;
  corner = board->getSquare(0,7);
  if(corner == player)
    cornerBonus += 50;
  else if(corner == otherPlayer(player))
    cornerBonus -= 50;
  corner = board->getSquare(7,0);
  if(corner == player)
    cornerBonus += 50;
  else if(corner == otherPlayer(player))
    cornerBonus -= 50;
  corner = board->getSquare(7,7);
  if(corner == player)
    cornerBonus += 50;
  else if(corner == otherPlayer(player))
    cornerBonus -= 50;
  return score + cornerBonus;
}
```

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

```
// Othello.cpp : Defines the entry point for the application.
//
#include "stdafx.h"
#include "Othello.h"
#include "Board.h"
#include "Game.h"


#include <stdlib.h>

using namespace std;

#define MAX_LOADSTRING 100
#define GRIDWIDTH 90
#define AI_AI_GAMES 1
// Global Variables:
HINSTANCE hInst;                        // current instance
TCHAR szTitle[MAX_LOADSTRING];                  // The title bar text
TCHAR szWindowClass[MAX_LOADSTRING];            // the main window class name
Game *game;                             // the game currently being played

int strategy1; // The strategy used by AI (defined in Game.h)
bool noplayer; // True if there is no human player (ie. AI vs. AI)
int strategy2; // Used only if noplayer is true
int humanPlayer;
int aiPlayer1;
int aiPlayer2;

```

```
// Forward declarations of functions included in this code module:
ATOM              MyRegisterClass(HINSTANCE hInstance);
BOOL              InitInstance(HINSTANCE, int);
LRESULT CALLBACK  WndProc(HWND, UINT, WPARAM, LPARAM);
INT_PTR CALLBACK  About(HWND, UINT, WPARAM, LPARAM);


// Draws the board in the window hdc using the global variable game.
void drawBoard(HDC);
void drawToken(HDC hdc, int x, int y, int player);

int APIENTRY _tWinMain(_In_ HINSTANCE hInstance,
                 _In_opt_ HINSTANCE hPrevInstance,
                 _In_ LPTSTR    lpCmdLine,
                 _In_ int       nCmdShow)
{
  UNREFERENCED_PARAMETER(hPrevInstance);
  UNREFERENCED_PARAMETER(lpCmdLine);

  // TODO: Place code here.
  MSG msg;
  HACCEL hAccelTable;

  // Initialize global strings
  LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
  LoadString(hInstance, IDC_OTHELLO, szWindowClass, MAX_LOADSTRING);
  MyRegisterClass(hInstance);

  // Perform application initialization:
  if (!InitInstance (hInstance, nCmdShow))
  {
    return FALSE;
  }

  hAccelTable = LoadAccelerators(hInstance, MAKEINTRESOURCE(IDC_OTHELLO));

  // Main message loop:
  while (GetMessage(&msg, NULL, 0, 0))
  {
    if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
    {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
    }
  }

  return (int) msg.wParam;
}



//
//  FUNCTION: MyRegisterClass()
```

```
//
//  PURPOSE: Registers the window class.
//
ATOM MyRegisterClass(HINSTANCE hInstance)
{
  WNDCLASSEX wcex;

  wcex.cbSize = sizeof(WNDCLASSEX);

  wcex.style          = CS_HREDRAW | CS_VREDRAW;
  wcex.lpfnWndProc    = WndProc;
  wcex.cbClsExtra     = 0;
  wcex.cbWndExtra     = 0;
  wcex.hInstance      = hInstance;
  wcex.hIcon          = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_OTHELLO));
  wcex.hCursor        = LoadCursor(NULL, IDC_HAND);
  wcex.hbrBackground  = (HBRUSH) CreateSolidBrush(RGB(0, 62, 0));
  wcex.lpszMenuName = MAKEINTRESOURCE(IDC_OTHELLO);
  wcex.lpszClassName  = szWindowClass;
  wcex.hIconSm        = LoadIcon(wcex.hInstance, MAKEINTRESOURCE(IDI_SMALL));

  return RegisterClassEx(&wcex);
}

//
//   FUNCTION: InitInstance(HINSTANCE, int)
//
//   PURPOSE: Saves instance handle and creates main window
//
//   COMMENTS:
//
//        In this function, we save the instance handle in a global variable and
//        create and display the main program window.
//
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
  HWND hWnd;
  game = new(Game);
  strategy1 = AI_STRATEGY_B;
  strategy2 = AI_STRATEGY_A;
  humanPlayer = BLACK;
  aiPlayer1 = BLACK;
  aiPlayer2 = WHITE;
  noplayer = false;


  hInst = hInstance; // Store instance handle in our global variable

  hWnd = CreateWindow(szWindowClass, szTitle, WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX,
     10, 10, GRIDSIZE*GRIDWIDTH + 17, GRIDSIZE*GRIDWIDTH + 60, NULL, NULL, hInstance, NULL);

  if (!hWnd)
  {
    return FALSE;
  }

  ShowWindow(hWnd, nCmdShow);
  UpdateWindow(hWnd);

  return TRUE;
}

//
//  FUNCTION: WndProc(HWND, UINT, WPARAM, LPARAM)
//
//  PURPOSE:  Processes messages for the main window.
//
//  WM_COMMAND  - process the application menu
//  WM_PAINT  - Paint the main window
//  WM_DESTROY  - post a quit message and return
//
//
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
  int wmId, wmEvent;
  PAINTSTRUCT ps;
  HDC hdc;

  switch (message)
  {
  case WM_COMMAND:
    wmId    = LOWORD(wParam);
    wmEvent = HIWORD(wParam);

    // Parse the menu selections:
    switch (wmId)
    {
    case IDM_ABOUT:
      DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd, About);
      break;
    case IDM_EXIT:
      DestroyWindow(hWnd);
      break;
    case IDM_AIFIRST:
      delete game;
      game = new(Game);
      noplayer = false;
      game->makeMoveAI(strategy2);
      humanPlayer = WHITE;
      aiPlayer2 = BLACK;
      InvalidateRect(hWnd, 0, TRUE);
      break;
    case IDM_AISECOND:
      delete game;
      game = new(Game);
      noplayer = false;
      humanPlayer = BLACK;
      aiPlayer2 = WHITE;
      InvalidateRect(hWnd, 0, TRUE);
      break;
    case IDM_STRATA:
      strategy2 = AI_STRATEGY_A;
      InvalidateRect(hWnd, 0, TRUE);
      break;
    case IDM_STRATB:
      strategy2 = AI_STRATEGY_B;
      InvalidateRect(hWnd, 0, TRUE);
      break;
    case IDM_STRATC:
      strategy2 = AI_STRATEGY_C;
      InvalidateRect(hWnd, 0, TRUE);
      break;
    case IDM_STRATD:
      strategy2 = AI_STRATEGY_D;
      InvalidateRect(hWnd, 0, TRUE);
      break;
    case IDM_STRATE:
      strategy2 = AI_STRATEGY_E;
      InvalidateRect(hWnd, 0, TRUE);
      break;
    case IDM_NOPLAYER:
      delete game;
      game = new(Game);
      strategy1 = AI_STRATEGY_D;
      strategy2 = AI_STRATEGY_B;
      aiPlayer2 = BLACK;
      aiPlayer1 = WHITE;
      noplayer = true;
      InvalidateRect(hWnd, 0, TRUE);
      break;
    case IDM_LEARNING:
      delete game;
      game = new(Game);
      noplayer = true;
      aiPlayer1 = BLACK;
      aiPlayer2 = WHITE;
      strategy1 = AI_STRATEGY_D;
      strategy2 = AI_STRATEGY_E;
      InvalidateRect(hWnd, 0, TRUE);
      break;
    default:
      return DefWindowProc(hWnd, message, wParam, lParam);
    }
    break;
  case WM_LBUTTONDOWN:
   {
      if(noplayer)
      {
         for(int counter = 0;counter < AI_AI_GAMES; counter++)
         {
            delete game;
            game = new(Game);
            while(!game->isOverGame())
            {
               if(game->getCurrentPlayer() == aiPlayer1)
               {
                  game->makeMoveAI(strategy1);
               }else
               {
                  game->makeMoveAI(strategy2);
               }
            }
            // Swap players
            int temp = aiPlayer1;
            aiPlayer1 = aiPlayer2;
            aiPlayer2 = temp;
         }
         int winner = game->whoWins();
         if(winner == WHITE)
            ::MessageBox(hWnd, _T(”The winner is white!”), _T(”Game over”),
                 MB_OK | MB_ICONEXCLAMATION);
         else if (winner == BLACK)
            ::MessageBox(hWnd, _T(”The winner is black!”), _T(”Game over”),
                 MB_OK | MB_ICONEXCLAMATION);
         else
            ::MessageBox(hWnd, _T(”It was a draw!”), _T(”Game over”), MB_OK |
                   MB_ICONEXCLAMATION);
         InvalidateRect(hWnd, 0, TRUE);
         break;
      }else
      {
         int x,y;
         x = LOWORD(lParam)/GRIDWIDTH;
         y = HIWORD(lParam)/GRIDWIDTH;
         if (game->isLegalMove(x,y)) {
            game->makeMove(x,y);
            while(!game->isOverGame())
            {
               if(game->getCurrentPlayer() == aiPlayer2)
                  game->makeMoveAI(strategy2);
               else
                  break;
            }
            if(game->isOverGame())
            {
               int winner = game->whoWins();
               if(winner == WHITE)
```

```
              ::MessageBox(hWnd, _T("The winner is white!"), _T("Game over"), MB_OK | MB_ICONEXCLAMATION);
            else if (winner == BLACK)
              ::MessageBox(hWnd, _T("The winner is black!"), _T("Game over"), MB_OK | MB_ICONEXCLAMATION);
            else
              ::MessageBox(hWnd, _T("It was a draw!"), _T("Game over"), MB_OK | MB_ICONEXCLAMATION);
          }

        } else {
          // Invalid move
        }
        // Repaint the window after the update
        InvalidateRect(hWnd, 0, TRUE);
        break;
      }
    }
  case WM_PAINT:
    hdc = BeginPaint(hWnd, &ps);
    // Draw Grid
    drawBoard(hdc);
    EndPaint(hWnd, &ps);
    break;
  case WM_DESTROY:
    PostQuitMessage(0);
    break;
  default:
    return DefWindowProc(hWnd, message, wParam, lParam);
  }
  return 0;
}

// Message handler for about box.
INT_PTR CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
  UNREFERENCED_PARAMETER(lParam);
  switch (message)
  {
  case WM_INITDIALOG:
    return (INT_PTR)TRUE;

  case WM_COMMAND:
    if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
    {
      EndDialog(hDlg, LOWORD(wParam));
      return (INT_PTR)TRUE;
    }
    break;
  }
  return (INT_PTR)FALSE;
}
```

```

void drawBoard(HDC hdc)
{
  HPEN hPenOld;
  // Draw the board lines
  HPEN hLinePen;
  COLORREF qLineColor;
  qLineColor = RGB(0, 0, 0);
  hLinePen = CreatePen(PS_SOLID, 2, qLineColor);
  hPenOld = (HPEN)SelectObject(hdc, hLinePen);


  for (int i = GRIDWIDTH; i < GRIDSIZE*GRIDWIDTH; i += GRIDWIDTH) {
    MoveToEx(hdc, i, 0, NULL);
    LineTo(hdc, i, GRIDSIZE*GRIDWIDTH);
  }


  for (int i = GRIDWIDTH; i < GRIDSIZE*GRIDWIDTH; i += GRIDWIDTH) {
    MoveToEx(hdc, 0, i, NULL);
    LineTo(hdc, GRIDSIZE*GRIDWIDTH, i);
  }

  Board currentBoard = game->getBoard();
  for (int i=0; i<GRIDSIZE;i++)
  {
    for(int j=0; j<GRIDSIZE;j++)
    {
      int square = currentBoard.getSquare(i,j);
      if(square != EMPTY)
      {
        drawToken(hdc,i,j,square);
      }
    }
  }

  SelectObject(hdc, hPenOld);
  DeleteObject(hLinePen);
}

void drawToken(HDC hdc, int x, int y, int player) {
  HPEN hPenOld;
  HPEN hLinePen;
  HBRUSH hBrushOld;
  HBRUSH hBrush;
  COLORREF qLineColor;

  if(player == WHITE)
  {
    qLineColor = RGB(255, 255, 255);
  }else if(player == BLACK)
  {
    qLineColor = RGB(0, 0, 0);
  }else
  {
    return;
  }

  const int penThickness = 2;
  hLinePen = CreatePen(PS_SOLID, penThickness, RGB(0, 0, 0));
  hBrush = CreateSolidBrush(qLineColor);
  hPenOld = (HPEN)SelectObject(hdc, hLinePen);
  hBrushOld = (HBRUSH)SelectObject(hdc, hBrush);

  // Get bounds
  const int x0  = x*GRIDWIDTH + 2*penThickness;
  const int x1  = (x + 1)*GRIDWIDTH - 2*penThickness;
  const int y0  = y*GRIDWIDTH + 2*penThickness;
  const int y1  = (y + 1)*GRIDWIDTH - 2*penThickness;

  Ellipse(hdc, x0, y0, x1, y1);

  SelectObject(hdc, hPenOld);
  DeleteObject(hLinePen);
  SelectObject(hdc, hBrushOld);
  DeleteObject(hBrush);
}
```

3.1.2 Resource.h

```
//{{NO_DEPENDENCIES}}
// Microsoft Visual C++ generated include file.
// Used by Othello.rc
//

#define IDS_APP_TITLE          103

#define IDR_MAINFRAME          128
#define IDD_OTHELLO_DIALOG         102
#define IDD_ABOUTBOX           103
#define IDM_ABOUT          104
#define IDM_EXIT           105
#define IDM_AIFIRST            110
#define IDM_AISECOND           111
#define IDM_STRATA             201
#define IDM_STRATB             202
#define IDM_STRATC             203
#define IDM_STRATD             209
#define IDM_STRATE             210
#define IDM_NOPLAYER           204
#define IDM_LEARNING           205
#define IDI_OTHELLO            107
#define IDI_SMALL          108
#define IDC_OTHELLO            109

```

```
#define IDC_MYICON            2
#ifndef IDC_STATIC
#define IDC_STATIC            −1
#endif
// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS

#define _APS_NO_MFC               130
#define _APS_NEXT_RESOURCE_VALUE    129
#define _APS_NEXT_COMMAND_VALUE     32771
#define _APS_NEXT_CONTROL_VALUE     1000
#define _APS_NEXT_SYMED_VALUE     110
#endif
#endif
```

3.1.3 Othello.h

```
#pragma once

#include ”resource.h”
```

3.2 Command Line Interface

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

3.2.1 othelloMain.cpp

```
#include<iostream>
#include ”Game.h”
#include ”Board.h”


using namespace std;

#define BOARD_THICKNESS 3

// Prints the board to stdout
void printBoard(Game ∗game);

int main(void)
{
  int aiStrategy;
  int humanPlayer;
  int aiPlayer;
  Game ∗game = new(Game);
  while(true)
  {
    cout << "Select strategy type by typing a number and pressing enter" << endl;
    cout << "1: Minimax" << endl;
    cout << "2: Minimax with Alpha-Beta pruning" << endl;
    cout << "3: Minimax with Alpha-Beta pruning and Book Moves" << endl;
    cout << "4: Random moves" << endl;
    int choice;
    cin >> choice;
    switch (choice)
    {
    case 1:
      aiStrategy = AI_STRATEGY_A;
      break;
    case 2:
      aiStrategy = AI_STRATEGY_B;
      break;
    case 3:
      aiStrategy = AI_STRATEGY_C;
      break;
    case 4:
      aiStrategy = AI_STRATEGY_D;
      break;
    default:
      cout << "Invalid selection. Terminating application.";
      return 0;
      break;
    }
    cout << "Select who plays first" << endl;
    cout << "1: You play first" << endl;
    cout << "2: Computer plays first" << endl;
    cin >> choice;
    switch (choice)
    {
    case 1:
      humanPlayer = BLACK;
      aiPlayer = WHITE;
      break;
    case 2:
      aiPlayer = BLACK;
      humanPlayer = WHITE;
      break;
    default:
      cout << "Invalid selection. Terminating application.";
      return 0;
      break;
    }
    while(!game->isOverGame())
    {
      printBoard(game);
      if(game->getCurrentPlayer() == humanPlayer) // Human turn to move
      {
        int x,y;
        char xChar, yChar;
        cout << "Enter a move (eg. A5) then press enter..." << endl;
        cin >> xChar >> yChar;
        x = xChar - 'A';
        y = yChar - '1';
        if(x>=0 && x<GRIDSIZE && y>=0 && y<GRIDSIZE)
        {
          if(game->isLegalMove(x,y))
            game->makeMove(x,y);
          else
            cout << "Illegal move attempted" << endl;
        }else
        {
          cout << "Invalid selection!" << endl;
        }
      }else // Computer turn to move
      {
        cout << "AI is moving..." << endl;
        game->makeMoveAI(aiStrategy);
      }
    }
    if(game->whoWins() == BLACK)
      cout << "Game over. The winner is black!" << endl;
    else if(game->whoWins() == WHITE)
      cout << "Game over. The winner is white!" << endl;
    else
      cout << "Game over. It was a draw!" << endl;
  }
  return 0;
}

void printBoard(Game *game)
{
  cout << "    ";
  for(int i=0; i<GRIDSIZE; i++)
  {
    char letter = 'A' + i;
    for(int j=0; j<BOARD_THICKNESS+1; j++){cout << letter;}
    cout << " ";
  }
  cout << endl;
  cout << "    ";
  for (int i=0; i<GRIDSIZE; i++)
  {
    for(int j=0; j<BOARD_THICKNESS+1; j++) {cout << "-";}
    cout << " ";
  }
  cout << endl;
  Board currentBoard = game->getBoard();
  for(int y=0; y<GRIDSIZE; y++)
  {
    for(int i=0; i<BOARD_THICKNESS; i++)
    {
      cout << (y+1) << " |";
```

```
            for(int x=0; x<GRIDSIZE; x++)
            {
                if(currentBoard.getSquare(x,y) == BLACK)
                {
                    for(int j=0; j<BOARD_THICKNESS+1; j++){cout << "B";}
                    cout << "|";
                }else if(currentBoard.getSquare(x,y) == WHITE)
                {
                    for(int j=0; j<BOARD_THICKNESS+1; j++){cout << "W";}
                    cout << "|";
                }
                else
                {
                    for(int j=0; j<BOARD_THICKNESS+1; j++){cout << " ";}
                    cout << "|";
                }
            }
            cout << endl;
        }
        cout << "    ";
        for (int i=0; i<GRIDSIZE; i++)
        {
            for(int j=0; j<BOARD_THICKNESS+1; j++) {cout << "-";}
            cout << " ";
        }
        cout << endl;
    }
    cout << endl;
}
```

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

```
#pragma once
#include "stdafx.h"
#include "Board.h"
#include "othelloAI.h"

#define AI_STRATEGY_A 777
#define AI_STRATEGY_B 778
#define AI_STRATEGY_C 779
#define AI_STRATEGY_D 780
#define AI_STRATEGY_E 781


```

```
class Game
{
  Board *board;
  int currentPlayer;
  OthelloAI *ai;
  bool canCurrentPlayerMove();

public:
  Game(void);
  ~Game(void);

  //
  // FUNCTION: getBoard()
  //
  // PURPOSE: returns a copy of the Board for this game
  Board getBoard();

  // FUNCTION getCurrentPlayer()
  //
  // RETURN: the next player to make a move in the game.
  // NOTE: currentPlayer is updated immediately if a player's
  // turn must be skipped. Thus this will not return the a
  // player that cannot move (except if a game is finished).
  int getCurrentPlayer();

  //
  // FUNCTION: isLegalMove(int x, int y)
  //
  // PURPOSE: Determines if a move in the (x,y) square is legal for the current player.
  bool isLegalMove(int x, int y);

  //
  // FUNCTION: isLegalMove(int x, int y)
  //
  // PURPOSE: Determines if a move in the (x,y) square is legal for the current player in Board someBoard.
  static bool isLegalMove(int x, int y, Board *someBoard, int playerToMove);

  //
  // FUNCTION: isLegalMove(int x, int y)
  //
  // PURPOSE: Determines if a move in the (x,y) square is legal
  // for the current player in the current board of game.
  bool isLegalMove(int x, int y, int player);

  //
  // FUNCTION: makeMove(int x, int y)
  //
  // PURPOSE: makes a move in square (x,y) if it is legal and it is the users turn. This function does nothing otherwise.
  void makeMove(int x, int y);

  //
  // FUNCTION: makeMoveAI()
  //
  // PURPOSE: Causes the AI to make a move in the current game
  void makeMoveAI(int strategy);

  // FUNCTION: Used to determine if a game has ended
  // RETURN: true if the game is at a terminal stage and false otherwise
  bool isOverGame();

  // FUNCTION: isOverGame(Board *someBoard)
  // PURPOSE: Used to determine if a board 'someBoard' is terminal (ie.
  // nobody can move)
  // RETURN: true if someBoard is at a terminal stage and false otherwise
  static bool isTerminal(Board *someBoard);

  // FUNCTION: Used to determine the winner of a game at its terminal stage
  int whoWins();
};
```

3.3.2 Game.cpp

```
#include "stdafx.h"
#include "Game.h"
#include "OthelloAI.h"
#include <vector>

using std::vector;

// RETURN: WHITE if player is BLACK and BLACK otherwise
inline int otherPlayer(int player);
// Initialises the board to the starting position for the Othello game
void initialiseOthello(Board *board);

Game::Game(void)
{
  currentPlayer = BLACK;
  board = new(Board);
  initialiseOthello(board);
  ai = new(OthelloAI);
}


Game::~Game(void)
{
  delete board;
  delete ai;
}

int Game::getCurrentPlayer()
{
  return currentPlayer;
}

bool Game::isLegalMove(int x, int y)
{
  if(board->getSquare(x,y) != EMPTY)
    return false;
  if(x+2<GRIDSIZE)
  {
    if(board->getSquare(x+2,y) == currentPlayer && board->getSquare(x+1,y) == otherPlayer(currentPlayer))
      return true;
    if(y+2<GRIDSIZE)
    {
      if(board->getSquare(x+2,y+2) == currentPlayer && board->getSquare(x+1,y+1) == otherPlayer(currentPlayer))
        return true;
    }
    if(y-2>=0)
    {
      if(board->getSquare(x+2,y-2) == currentPlayer && board->getSquare(x+1,y-1) == otherPlayer(currentPlayer))
        return true;
    }
  }
  if(x-2>=0)
  {
    if(board->getSquare(x-2,y) == currentPlayer && board->getSquare(x-1,y) == otherPlayer(currentPlayer))
      return true;
    if(y+2<GRIDSIZE)
    {
      if(board->getSquare(x-2,y+2) == currentPlayer && board->getSquare(x-1,y+1) == otherPlayer(currentPlayer))
        return true;
    }
    if(y-2>=0)
    {
      if(board->getSquare(x-2,y-2) == currentPlayer && board->getSquare(x-1,y-1) == otherPlayer(currentPlayer))
        return true;
    }
  }
  if(y+2<GRIDSIZE)
  {
    if(board->getSquare(x,y+2) == currentPlayer && board->getSquare(x,y+1) == otherPlayer(currentPlayer))
      return true;
  }
  if(y-2>=0)
  {
    if(board->getSquare(x,y-2) == currentPlayer && board->getSquare(x,y-1) == otherPlayer(currentPlayer))
```

```
      return true;
  }

  return false;
}


Board Game::getBoard()
{
  return *board;
}

void Game::makeMove(int x, int y)
{
  if(isLegalMove(x,y))
  {
    board->insertToken(x,y,currentPlayer);
    currentPlayer = otherPlayer(currentPlayer);
    if(isOverGame())
    {
      // Do nothing
    }
    if(!canCurrentPlayerMove())
    {
      currentPlayer = otherPlayer(currentPlayer);
    }
  }
}

void Game::makeMoveAI(int strategy)
{
  if(isOverGame())
    return;
  ai->setPlayer(currentPlayer);
  while(true)
  {
    int aiMove = ai->getMove(board, strategy);
    int aiMoveX = aiMove%GRIDSIZE;
    int aiMoveY = aiMove/GRIDSIZE;
    if(isLegalMove(aiMoveX,aiMoveY))
    {
      board->insertToken(aiMoveX,aiMoveY,currentPlayer);
      currentPlayer = otherPlayer(currentPlayer);
      break;
    }else
    {
    }
  }
  if(isOverGame())
  {
    // Do nothing
  }else if(!canCurrentPlayerMove())
  {
    currentPlayer = otherPlayer(currentPlayer);
  }
}



inline int otherPlayer(int player)
{
  if(player == BLACK)
    return WHITE;
  else
    return BLACK;
}

void initialiseOthello(Board *board)
{
  delete board;
  board = new(Board);
  board->changeToken(3,3,WHITE);
  board->changeToken(4,4,WHITE);
  board->changeToken(3,4,BLACK);
  board->changeToken(4,3,BLACK);
}

bool Game::isLegalMove(int x, int y, Board *someBoard, int playerToMove)
{
  if(someBoard->getSquare(x,y) != EMPTY)
    return false;
  if(x+2<GRIDSIZE)
  {
    if(someBoard->getSquare(x+2,y) == playerToMove && someBoard->getSquare(x+1,y) == otherPlayer(playerToMove))
      return true;
    if(y+2<GRIDSIZE)
    {
      if(someBoard->getSquare(x+2,y+2) == playerToMove && someBoard->getSquare(x+1,y+1) == otherPlayer(playerToMove))
        return true;
    }
    if(y-2>=0)
    {
      if(someBoard->getSquare(x+2,y-2) == playerToMove && someBoard->getSquare(x+1,y-1) == otherPlayer(playerToMove))
        return true;
    }
  }
  if(x-2>=0)
  {
    if(someBoard->getSquare(x-2,y) == playerToMove && someBoard->getSquare(x-1,y) == otherPlayer(playerToMove))
      return true;
    if(y+2<GRIDSIZE)
    {
      if(someBoard->getSquare(x-2,y+2) == playerToMove && someBoard->getSquare(x-1,y+1) == otherPlayer(playerToMove))
        return true;
    }
    if(y-2>=0)
    {
      if(someBoard->getSquare(x-2,y-2) == playerToMove && someBoard->getSquare(x-1,y-1) == otherPlayer(playerToMove))
        return true;
    }
  }
  if(y+2<GRIDSIZE)
  {
    if(someBoard->getSquare(x,y+2) == playerToMove && someBoard->getSquare(x,y+1) == otherPlayer(playerToMove))
      return true;
  }
  if(y-2>=0)
  {
    if(someBoard->getSquare(x,y-2) == playerToMove && someBoard->getSquare(x,y-1) == otherPlayer(playerToMove))
      return true;
  }
  return false;
}

bool Game::isLegalMove(int x, int y, int player)
{
  if(board->getSquare(x,y) != EMPTY)
    return false;
  if(x+2<GRIDSIZE)
  {
    if(board->getSquare(x+2,y) == player && board->getSquare(x+1,y) == otherPlayer(player))
      return true;
    if(y+2<GRIDSIZE)
    {
      if(board->getSquare(x+2,y+2) == player && board->getSquare(x+1,y+1) == otherPlayer(player))
        return true;
    }
    if(y-2>=0)
    {
      if(board->getSquare(x+2,y-2) == player && board->getSquare(x+1,y-1) == otherPlayer(player))
        return true;
    }
  }
  if(x-2>=0)
  {
    if(board->getSquare(x-2,y) == player && board->getSquare(x-1,y) == otherPlayer(player))
      return true;
    if(y+2<GRIDSIZE)
    {
      if(board->getSquare(x-2,y+2) == player && board->getSquare(x-1,y+1) == otherPlayer(player))
        return true;
    }
    if(y-2>=0)
    {
      if(board->getSquare(x-2,y-2) == player && board->getSquare(x-1,y-1) == otherPlayer(player))
        return true;
    }
  }
  if(y+2<GRIDSIZE)
  {
    if(board->getSquare(x,y+2) == player && board->getSquare(x,y+1) == otherPlayer(player))
      return true;
  }
  if(y-2>=0)
  {
    if(board->getSquare(x,y-2) == player && board->getSquare(x,y-1) == otherPlayer(player))
      return true;
  }
  return false;
}

bool Game::canCurrentPlayerMove()
{
  for(int i=0;i<GRIDSIZE;i++)
    for(int j=0;j<GRIDSIZE;j++)
      if(this->isLegalMove(i,j,currentPlayer))
        return true;
  return false;
}


bool Game::isOverGame()
{
  for(int i=0;i<GRIDSIZE;i++)
    for(int j=0;j<GRIDSIZE;j++)
      if(this->isLegalMove(i,j,WHITE) || this->isLegalMove(i,j,BLACK))
        return false;
  return true;
}

bool Game::isTerminal(Board *someBoard)
{
  for(int i=0;i<GRIDSIZE;i++)
    for(int j=0;j<GRIDSIZE;j++)
      if(isLegalMove(i,j,someBoard,WHITE) || isLegalMove(i,j,someBoard,BLACK))
```

```
        return false;
  return true;
}

int Game::whoWins()
{
  int white = 0;
  int black = 0;
  for(int i=0;i<GRIDSIZE;i++)
    for(int j=0;j<GRIDSIZE;j++)
      if(board->getSquare(i,j) == WHITE)
        white++;
      else if(board->getSquare(i,j) == BLACK)
        black++;
  if(white > black)
    return WHITE;
  if(black > white)
    return BLACK;
  return EMPTY;
}
```

3.3.3 Board.h

```
#pragma once
#include "stdafx.h"

#define EMPTY 0
#define BLACK 1
#define WHITE -1
#define GRIDSIZE 8

class Board
{
private:
  int boardArray[GRIDSIZE*GRIDSIZE];
public:

  // Constructs a board with all cells initialised to EMPTY
  Board(void);

  // FUNCTION: Used to create a new board with one token added.
  //
  // RETURN: A new board with all squares the same as board
  // but square (x,y) set to player
  Board(Board *board, int x, int y, int player);

  ~Board(void);

  // Gets the value at square (x,y) in the board
  int getSquare(int x, int y);

  // Inserts a token of colour 'colour' into the board at square (x,y)
  // and also makes the flips that occur in othello game after such
  // an insert
  void insertToken(int x, int y, int colour);

  // Same as insertToken except it makes no flips.
  void changeToken(int x, int y, int colour);

  void changeBoard(Board *board, int x, int y, int player);
};
```

3.3.4 Board.cpp

```
#include "stdafx.h"
#include "Board.h"

// Flips all the pieces when a token of type currentPlayer is inserted into
// square (x,y).
void makeFlips(int x, int y, int insertedColour, Board *board);

// RETURN: WHITE if player is BLACK and BLACK otherwise
inline int otherColour(int colour);

Board::Board(void)
{
  for(int i=0;i<GRIDSIZE;i++)
  {
    for(int j=0;j<GRIDSIZE;j++)
    {
      boardArray[i*GRIDSIZE+j] = EMPTY;
    }
  }
}

Board::Board(Board *board, int x, int y, int player)
{
  for(int i = 0; i < GRIDSIZE*GRIDSIZE; i++)
  {
    this->boardArray[i] = board->boardArray[i];
  }
  this->insertToken(x,y,player);
}

Board::~Board(void)
{
}

void Board::insertToken(int x, int y, int colour)
{
  boardArray[x + y*GRIDSIZE] = colour;
  makeFlips(x,y,colour,this);
}

```

```
void Board:: changeToken(int x, int y, int colour)
{
  boardArray[x + y*GRIDSIZE] = colour;
}

int Board::getSquare(int x, int y)
{
  return boardArray[x + y*GRIDSIZE];
}


void makeFlips(int x, int y, int insertedColour, Board *board)
{
  if(x+2<GRIDSIZE)
  {
    if(board->getSquare(x+2,y) == insertedColour && board->getSquare(x+1,y) == otherColour(insertedColour))
      board->changeToken(x+1,y,insertedColour);
    if(y+2<GRIDSIZE)
    {
      if(board->getSquare(x+2,y+2) == insertedColour && board->getSquare(x+1,y+1) == otherColour(insertedColour))
        board->changeToken(x+1,y+1,insertedColour);
    }
    if(y-2>=0)
    {
      if(board->getSquare(x+2,y-2) == insertedColour && board->getSquare(x+1,y-1) == otherColour(insertedColour))
        board->changeToken(x+1,y-1,insertedColour);
    }
  }
  if(x-2>=0)
  {
    if(board->getSquare(x-2,y) == insertedColour && board->getSquare(x-1,y) == otherColour(insertedColour))
      board->changeToken(x-1,y,insertedColour);
    if(y+2<GRIDSIZE)
    {
      if(board->getSquare(x-2,y+2) == insertedColour && board->getSquare(x-1,y+1) == otherColour(insertedColour))
        board->changeToken(x-1,y+1,insertedColour);
    }
    if(y-2>=0)
    {
      if(board->getSquare(x-2,y-2) == insertedColour && board->getSquare(x-1,y-1) == otherColour(insertedColour))
        board->changeToken(x-1,y-1,insertedColour);
    }
  }
  if(y+2<GRIDSIZE)
  {
    if(board->getSquare(x,y+2) == insertedColour && board->getSquare(x,y+1) == otherColour(insertedColour))
```

```
      board->changeToken(x,y+1,insertedColour);
  }
  if(y-2>=0)
  {
    if(board->getSquare(x,y-2) == insertedColour && board->getSquare(x,y-1) == otherColour(insertedColour))
      board->changeToken(x,y-1,insertedColour);
  }
}

inline int otherColour(int colour)
{
  if(colour == BLACK)
    return WHITE;
  else
    return BLACK;
}
```

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

```
#pragma once
#include "stdafx.h"
#include "Board.h"
#include "State.h"
#include <vector>
#include <hash_set>
#include "StateMove.h"
#include <exception>
#include <map>
#include <queue>

using std::vector; using std::exception; using std::multimap; using std::queue;using namespace std;
class OthelloAI
{
private:
  int me;


  multimap<State, int> book;

  //hash_set<StateMove> book;

  // Returns the first legal move found
```

```
int firstLegalMove(Board *board);

// FUNCTION: Determines a minimax score
//
// RETURN: The score for the node board using minimax algorithm to a depth of depth
int minimax(Board *board, int depth, int player);

// FUNCTION: Uses minimax search method to find a move
//
// RETURN: The move decided by minimax search algorithm
// to be the best (in the format (x+y*GRIDSIZE))
int minimaxSearch(Board *board);

// FUNCTION: Determines a negamax score
//
// RETURN: The score for the node board using negamax algorithm to a depth of depth
int negamax(Board *board, int depth, int playerToMove);

// FUNCTION: Uses negamax search method to find a move
//
// RETURN: The move decided by negamax search algorithm
// to be the best (in the format (x+y*GRIDSIZE))
int negamaxSearch(Board *board);

// PURPOSE: Uses minimax search with alpha beta pruning
// method and scouting optimization to find a best move
//
// RETURN: The move decided by minimax search algorithm
// to be the best (in the format 'x+y*GRIDSIZE')
int negascoutSearch(Board *board);

// FUNCTION: score(Board *board, int depth, int alpha, int beta, int player)
//
// PURPOSE: Determines a minimax score using alpha beta pruning and scouting
//
// RETURN: The score for the node board using minimax algorithm to a depth of depth
int negascout(Board *board, int depth, int alpha, int beta, int player);

// PURPOSE: Uses minimax search with alpha beta pruning
// method to find a move
//
// RETURN: The move decided by minimax search algorithm
// to be the best (in the format 'x+y*GRIDSIZE')
int alphabetaSearch(Board *board);

// PURPOSE: ONLY USE FOR LEARNING. Uses minimax search
// with alpha beta pruning and scouting to find a move with a
// very deep search tree.
```

```
  //
  // RETURN: The move decided by minimax search algorithm
  // to be the best (in the format 'x+y*GRIDSIZE')
  int deepSearch(Board *board);

  // FUNCTION: alphabeta(Board *board, int depth, int alpha, int beta, int player)
  //
  // PURPOSE: Determines a minimax score using alpha beta pruning
  //
  // RETURN: The score for the node board using minimax algorithm to a depth of depth
  int alphabeta(Board *board, int depth, int alpha, int beta, int player);



  // PURPOSE: Opens the move book and stores in 'book' variable
  // to be read in the findInMoveBook(Board *board, int playerToMove) function
  void openBook();

  // FUNCTION: findMoveInBook(Board *board, int playerToMove)
  // PURPOSE: Used to find the move to to make in the current state
  // defined by board and playerToMove.
  // RETURN: The move to make as defined in the book
  // THROWS: stateNotInBookException
  int findMoveInBook(Board *board, int playerToMove);


public:

  // Sets the 'me' field of the OthelloAI to player
  void setPlayer(int player);

  // Default AI with me set to BLACK
  OthelloAI(void);
  // AI with me to player
  OthelloAI(int player);
  ~OthelloAI(void);

  // Used to find the move chosen by the AI for the currentBoard
  // using the strategy 'strategy'
  //
  // RETURN: A move in the format (x+y*GRIDSIZE)
  int getMove(Board *currentBoard, int strategy);

};
```

3.4.2 OthelloAI.cpp

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

```
#include "Board.h"
#include "Game.h"
#include "StateMove.h"
#include <time.h>
#include <queue>

#include <stdlib.h>

using namespace std;

#define INFINITY 999999
#define MINIMAX_DEPTH 4
#define ALPHABETA_DEPTH 5
#define DEEP_DEPTH 10
#define SCOUT_DEPTH 8

#define BOOK "book_moves.dat"

// RETURN: WHITE if player is BLACK and BLACK otherwise
inline int otherPlayer(int player);

// FUNCTION: scoreBoardAdvanced(Board *board, player)
// PURPOSE: An advanced heuristic function for scoring
// a Board board from the perspective of player.
int scoreBoardAdvanced(Board *board, int player);

// PURPOSE: Used as a simple heuristic scoring function
//
// RETURN: Sum over all tokens beloning to 'me' minus the sum over all tokens belonging to other player
int scoreBoardSimple(Board *board, int player);

// FUNCTION: randomLegalMove(Board *board)
// RETURN: a random legal move in the Board
// referenced by 'board' for player 'me'
int randomLegalMove(Board *board, int me);


// FUNCTION: appendToBook(StateMove sm)
// PURPOSE: Saves the state move sm in the book at the end
void appendToBook(StateMove sm);

// RETURN: number of possible moves for player 'me' in
// the Board 'board'
int possibleMoves(Board *board, int me);

class stateNotInBookException: public exception
{
  virtual const char* what() const throw()
  {
    return "State not in book";
  }
} StateNotInBookException;
```

```

// BoardOrder class is used only for
// expanding children in good order in
// negascout algorithm
class BoardOrder
{
  friend bool operator<(BoardOrder a, BoardOrder b);
public:
  ~BoardOrder(void){}
  BoardOrder(void){}
  BoardOrder(int priority, Board board)
  {
    this->priority = priority;
    this->board = board;
  }
  int priority;
  Board board;
};

// Redefine less than operator for use in
// priority queue
bool operator <(BoardOrder a, BoardOrder b)
{
  return a.priority < b.priority;
}

OthelloAI::OthelloAI(void)
{
  openBook();
  me = BLACK;
}

OthelloAI::OthelloAI(int player)
{
  openBook();
  me = player;
}

void OthelloAI::setPlayer(int player)
{
  me = player;
}

OthelloAI::~OthelloAI(void)
{

}

int OthelloAI::getMove(Board *currentBoard, int strategy)
{
  int move = 0;
  switch(strategy)
  {
```

```
  case AI_STRATEGY_A:
    move = minimaxSearch(currentBoard);
  case AI_STRATEGY_B:
    move = alphabetaSearch(currentBoard);
    break;
  case AI_STRATEGY_C:
    try // IF: current state is in book
    {
      move = findMoveInBook(currentBoard,me);
    }catch (stateNotInBookException e) // ELSE:
    {
      move = negascoutSearch(currentBoard);
    }
    break;
  case AI_STRATEGY_D: // Random moving player
    move =  randomLegalMove(currentBoard,me);
    break;
  case AI_STRATEGY_E: // Learning phase
    try // IF: current state is in book
    {
      move = findMoveInBook(currentBoard,me);
    }catch (stateNotInBookException e) // ELSE:
    {
      // Make a move using minimax with deep alphabeta search
      move = deepSearch(currentBoard);
      // Record that move in book
      appendToBook(StateMove(currentBoard, me, move));
    }
    break;
  }
  return move;
}

int OthelloAI::negascoutSearch(Board *board)
{
  int bestMoveScore = -2*INFINITY;
  int bestMove;
  int score;

  for (int move=0; move < GRIDSIZE*GRIDSIZE; move++)
  {
    if(Game::isLegalMove(move%GRIDSIZE,move/GRIDSIZE,board,me))
    {
      Board moveBoard (board, move%GRIDSIZE, move/GRIDSIZE, me);
      score = -negascout(&moveBoard, SCOUT_DEPTH, -2*INFINITY, -
          bestMoveScore, otherPlayer(me));
      if(score > bestMoveScore)
      {
        bestMove = move;
        bestMoveScore = score;
      }
    }
  }
  return bestMove;
}

int OthelloAI::negascout(Board *board, int depth, int alpha, int beta, int player)
{
  if (depth <= 0 || Game::isTerminal(board))
  {
    int score = scoreBoardAdvanced(board,player);
    return score;
  }
  priority_queue<BoardOrder> children;
  bool cantMove = true;

  // Queue children of state
  for(int i=0;i<GRIDSIZE*GRIDSIZE;i++)
  {
    int x = i%GRIDSIZE;
    int y = i/GRIDSIZE;
    if(Game::isLegalMove(x,y,board,player))
    {
      cantMove = false;
      Board child (board,x,y,player);
      // Prioritise depending on whether it is min turn or
      // max turn
      int priority;
      if(player == me)
        priority = scoreBoardSimple(&child,me);
      else
        priority = -scoreBoardSimple(&child,me);
      children.push(BoardOrder (priority, child));
    }
  }
  if(cantMove)
  {
    // Child is same node
    children.push(BoardOrder (0, *board));
    depth++; // Increment depth for consistency with other algorithms
  }

  bool firstChild = true;
  int b = beta; // scouting variable
  while(children.size() > 0)
  {
    Board child = (children.top().board);
    children.pop();
    int score = -negascout(&child, depth-1, -b, -alpha, otherPlayer(player));
    if(alpha < score && score < beta && !firstChild) // Perform full alpha-beta search
      score = -negascout(&child, depth - 1, -beta, -alpha, otherPlayer(player));
    firstChild = false;
    alpha = max(alpha, score);
    if(beta <= alpha)
      return alpha;
    b = alpha + 1;
  }

  return alpha;
}

int OthelloAI::minimaxSearch(Board *board)
{
  int bestMoveScore = -2*INFINITY;
  int bestMove;
  int score;
  for (int move=0; move < GRIDSIZE*GRIDSIZE; move++)
  {
    if(Game::isLegalMove(move%GRIDSIZE,move/GRIDSIZE,board,me))
    {
      Board moveBoard (board, move%GRIDSIZE, move/GRIDSIZE, me);
      score = minimax(&moveBoard, MINIMAX_DEPTH, otherPlayer(me));
      if(score > bestMoveScore)
      {
        bestMove = move;
        bestMoveScore = score;
      }
    }
  }
  return bestMove;
}

int OthelloAI::minimax(Board *board, int depth, int player)
{
  int score;
  if (depth <= 0 || Game::isTerminal(board))
    return scoreBoardAdvanced(board,me);
  bool cantMove = true;
  if(player == me)
  {
    score = -2*INFINITY;
    for(int i=0;i<GRIDSIZE;i++)
    {
      for(int j=0;j<GRIDSIZE;j++)
      {
        if(Game::isLegalMove(i,j,board,player))
        {
          Board child (board, i, j, player);
          cantMove = false;
          score = max(score, minimax(&child, depth-1, otherPlayer(player)));
        }
      }
    }
    if(cantMove) // Child node is same node
```

```
      score = minimax(board, depth, otherPlayer(player));
  }else
  {
    score = 2*INFINITY;
    for(int i=0;i<GRIDSIZE;i++)
    {
      for(int j=0;j<GRIDSIZE;j++)
      {
        if(Game::isLegalMove(i,j,board,player))
        {
          Board child (board, i, j, player);
          cantMove = false;
          score = min(score, minimax(&child, depth-1, otherPlayer(player)));
        }
      }
    }
    if(cantMove) // Child node is same node
      score = minimax(board, depth, otherPlayer(player));
  }
  return score;
}

int OthelloAI::alphabetaSearch(Board *board)
{
  int bestMoveScore = -2*INFINITY;
  int bestMove;
  int alphabetaScore;
  for (int move=0; move < GRIDSIZE*GRIDSIZE; move++)
  {
    if(Game::isLegalMove(move%GRIDSIZE,move/GRIDSIZE,board,me))
    {
      Board moveBoard (board, move%GRIDSIZE, move/GRIDSIZE, me);
      alphabetaScore = alphabeta(&moveBoard, ALPHABETA_DEPTH, bestMoveScore, +INFINITY, otherPlayer(me));
      if(alphabetaScore > bestMoveScore)
      {
        bestMove = move;
        bestMoveScore = alphabetaScore;
      }
    }
  }
  return bestMove;
}

int OthelloAI::alphabeta(Board *board, int depth, int alpha, int beta, int player)
{
  if (depth <= 0 || Game::isTerminal(board))
    return scoreBoardAdvanced(board,me);
  bool cantMove = true;
  if(player == me)
  {
    // Queue children of state
    for(int i=0;i<GRIDSIZE*GRIDSIZE;i++)
    {
      int x = i%GRIDSIZE;
      int y = i/GRIDSIZE;
      if(Game::isLegalMove(x,y,board,player))
      {
        cantMove = false;
        alpha = max(alpha, alphabeta(&Board(board,x,y,player), depth-1, alpha, beta, otherPlayer(player)));
        if(beta <= alpha)
          return alpha;
      }
    }
    if(cantMove) // Child node is same node
      alpha = max(alpha, alphabeta(board, depth, alpha, beta, otherPlayer(player)));
    return alpha;
  }else
  {
    // Queue children of state
    for(int i=0;i<GRIDSIZE*GRIDSIZE;i++)
    {
      int x = i%GRIDSIZE;
      int y = i/GRIDSIZE;
      if(Game::isLegalMove(x,y,board,player))
      {
        cantMove = false;
        beta = min(beta, alphabeta(&Board(board,x,y,player), depth-1, alpha, beta, otherPlayer(player)));
        if(beta <= alpha)
          return beta;
      }
    }
    if(cantMove) // Child node is same node
      beta = min(beta, alphabeta(board, depth, alpha, beta, otherPlayer(player)));
    return beta;
  }
}

int OthelloAI::negamaxSearch(Board *board)
{
  int bestMoveScore = -2*INFINITY;
  int bestMove;
  int score;
  for (int move=0; move < GRIDSIZE*GRIDSIZE; move++)
  {
    if(Game::isLegalMove(move%GRIDSIZE,move/GRIDSIZE,board,me))
    {
      Board moveBoard (board, move%GRIDSIZE, move/GRIDSIZE, me);
      score = negamax(&moveBoard, MINIMAX_DEPTH, otherPlayer(me));
      if(score > bestMoveScore)
      {
        bestMove = move;
        bestMoveScore = score;
      }
    }
  }
  return bestMove;
}

int OthelloAI::negamax(Board *board, int depth, int playerToMove)
{
  Board *child;
  if(depth <= 0 || Game::isTerminal(board))
    return scoreBoardAdvanced(board, otherPlayer(playerToMove));
  int alpha = INFINITY;
  bool cantMove = true;
  for(int i=0; i<GRIDSIZE; i++)
  {
    for(int j=0; j<GRIDSIZE; j++)
    {
      if(Game::isLegalMove(i, j, board, playerToMove))
      {
        Board child (board, i, j, playerToMove);
        cantMove = false;
        alpha = min(alpha, -negamax(&child, depth-1, otherPlayer(playerToMove)));
      }
    }
  }
  if(cantMove) // child is the same as board if you skip a turn
    alpha = min(alpha, -negamax(board, depth, otherPlayer(playerToMove)));
  return alpha;
}

int randomLegalMove(Board *board, int me)
{
  srand(time(NULL));
  while(true)
  {
    int move = rand()%64;
    if (Game::isLegalMove(move%GRIDSIZE, move/GRIDSIZE, board, me))
      return move;
  }
}

int OthelloAI::firstLegalMove(Board *board)
{
  for(int i=0;i<GRIDSIZE;i++)
    for(int j=0;j<GRIDSIZE;j++)
      if(Game::isLegalMove(i,j,board,me))
        return (i+j*GRIDSIZE);
}

```

```
int scoreBoardSimple(Board *board, int player)
{
  int score = 0;
  for(int i=0;i<GRIDSIZE;i++)
  {
    for(int j=0;j<GRIDSIZE;j++)
    {
      if(board->getSquare(i,j) == player)
        score++;
      else if(board->getSquare(i,j) == otherPlayer(player))
        score--;
    }

  }
  return score;
}

int scoreBoardAdvanced(Board *board, int player)
{
  int scoreSimple = scoreBoardSimple(board,player);

  if(Game::isTerminal(board))
  {
    // player wins
    if(scoreSimple > 0)
      return INFINITY;
    // player loses
    if(scoreSimple < 0)
      return -INFINITY;
    // match was a draw (draw is more desirable
    // than a loss but still less desirable than
    // any other state)
    else
      return -INFINITY/2;
  }

  // Add 10 points for every square owned by player and
  // subtract 10 for every square owned by other player
  int score = 10*scoreSimple;

  // Add 5 points for every legal move and subtract
  // 5 for every legal move of opponent
  for(int i=0;i<GRIDSIZE;i++)
  {
    for(int j=0;j<GRIDSIZE;j++)
    {

      if(Game::isLegalMove(i,j,board,player))
      {
        score += 5;
      }
      if(Game::isLegalMove(i,j,board,otherPlayer(player)))
      {
        score -= 5;
      }
    }
  }

  // Add 50 points for every corner owned by player
  // and subtract 50 points for every corner owned
  // by other player
  int cornerBonus = 0;
  int corner = board->getSquare(0,0);
  if(corner == player)
    cornerBonus += 50;
  else if(corner == otherPlayer(player))
    cornerBonus -= 50;
  corner = board->getSquare(0,7);
  if(corner == player)
    cornerBonus += 50;
  else if(corner == otherPlayer(player))
    cornerBonus -= 50;
  corner = board->getSquare(7,0);
  if(corner == player)
    cornerBonus += 50;
  else if(corner == otherPlayer(player))
    cornerBonus -= 50;
  corner = board->getSquare(7,7);
  if(corner == player)
    cornerBonus += 50;
  else if(corner == otherPlayer(player))
    cornerBonus -= 50;
  return score + cornerBonus;
}

inline int otherPlayer(int player)
{

  if(player == BLACK)
    return WHITE;
  else
    return BLACK;
}


void appendToBook(StateMove sm)
{
  ofstream bookfile;
  bookfile.open(BOOK, ofstream::out | ofstream::app);
  bookfile << sm.stateFirstHalf << " " << sm.stateSecondHalf << " " << sm.player << " " << sm.move << endl;
  bookfile.close();
}

int OthelloAI::findMoveInBook(Board *board, int playerToMove)
{
  int move = 0;
  if(book.count(State (board, playerToMove)) > 0)
    move = book.lower_bound(State (board, playerToMove))->second;
  else
    throw StateNotInBookException;
  return move;
}

void OthelloAI::openBook()
{
  unsigned long long firstHalf, secondHalf;
  int player, move;
  ifstream bookfile (BOOK);
  if (bookfile.is_open())
  {
    while ( bookfile.good() )
    {
      bookfile >> firstHalf >> secondHalf >> player >> move;
      book.insert(pair<State, int> (State (firstHalf, secondHalf, player), move));
    }
    bookfile.close();
  }
}

int OthelloAI::deepSearch(Board *board)
{
  int bestMoveScore = -2*INFINITY;
  int bestMove;
  int score;
  for (int move=0; move < GRIDSIZE*GRIDSIZE; move++)
  {
    if(Game::isLegalMove(move%GRIDSIZE,move/GRIDSIZE,board,me))
    {
      Board moveBoard (board, move%GRIDSIZE, move/GRIDSIZE, me);
      score = -negascout(&moveBoard, DEEP_DEPTH, -2*INFINITY, -bestMoveScore, otherPlayer(me));
      if(score > bestMoveScore)
      {
        bestMove = move;
        bestMoveScore = score;
      }
    }
  }
  return bestMove;
}

int possibleMoves(Board *board, int me)
{
  int moves;
  for(int i=0;i<GRIDSIZE;i++)
  {
    for(int j=0;j<GRIDSIZE;j++)
    {
      if(Game::isLegalMove(i,j,board,me))
        moves++;
    }
  }
  return moves;
}
```

3.4.3 State.h

```
#pragma once
#include "stdafx.h"
#include "Board.h"


class State
{
  friend bool operator<(State a, State b);
  friend bool operator==(State a, State b);
public:
  State(void);

  // RETURN: true if these states are equal and false otherwise
  bool isEqual(State *other);

  // RETURN: true if the state is the same as that defined by
  // the parameters
  bool isEqual(Board *board, int player);

  // Construct a state defined by the parameters
  State(Board *board, int player);

  // Construct a state explicitly from the field values
  State(unsigned long long first, unsigned long long second, unsigned long long player);

  ~State(void);

  // RETURN: The hash code for the state
  unsigned long long getHash() const;

  unsigned long long hash;
  unsigned long long firstHalf;
  unsigned long long movingPlayer;
  unsigned long long secondHalf;
};
```

3.4.4 State.cpp

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

```
#include "State.h"

// RETURN: x to the power i
inline unsigned long long power(int x, int i)
{
  unsigned long long answer = 1;
  for(int counter=0;counter<i;counter++) answer *= x;
  return answer;
}

// RETURN: Unique modulo 3 value of square
inline unsigned long long value(int square);

State::State(void)
{
}


State::~State(void)
{
}

bool State::isEqual(Board *board, int player)
{
  unsigned long long first = 0;
  unsigned long long second = 0;
  for(unsigned long long i=0; i<32 ;i++)
  {
    first += power(3,i)*value(board->getSquare(i%GRIDSIZE,i/GRIDSIZE));
  }
  for(unsigned long long i=32; i<64 ;i++)
  {
    second += power(3,i)*value(board->getSquare(i%GRIDSIZE,i/GRIDSIZE));
  }
  return (this->firstHalf == first && this->secondHalf == second && this->
      movingPlayer == player);
}

bool State::isEqual(State *other)
{
  return (this->firstHalf == other->firstHalf && this->secondHalf == other
      ->secondHalf && this->movingPlayer == other->movingPlayer);
}

State::State(unsigned long long first, unsigned long long second, unsigned
    long long player)
{
  firstHalf = first;
  secondHalf = second;
  movingPlayer = player;
  hash = getHash();
}

```

```cpp
State::State(Board *board, int player)
{
  firstHalf = 0;
  secondHalf = 0;
  for(unsigned long long i=0; i<32 ;i++)
  {
    firstHalf += power(3,i)*value(board->getSquare(i%GRIDSIZE,i/GRIDSIZE));
  }
  for(unsigned long long i=32; i<64 ;i++)
  {
    secondHalf += power(3,i)*value(board->getSquare(i%GRIDSIZE,i/GRIDSIZE));
  }
  movingPlayer = player;
  hash = getHash();
}

unsigned long long State::getHash() const
{
  return firstHalf*secondHalf + movingPlayer;
}

inline unsigned long long value(int square)
{
  if(square == WHITE)
    return 2;
  if(square == BLACK)
    return 1;
  return 0;
}

// Overwrite comparison operator 'less' for two states
bool operator<(State a, State b)
{
  return a.getHash() < b.getHash();
}

bool operator==(const State a, const State b)
{
   return (a.firstHalf == b.firstHalf && a.secondHalf == b.secondHalf && a.movingPlayer == b.movingPlayer);
}
```

3.4.5 StateMove.h

```cpp
#pragma once
#include "stdafx.h"
#include "Board.h"


class StateMove
{
```

```
public:
  StateMove(void);

  // Constructs a StateMove by explicitly specifying all its field values
  StateMove(unsigned long long firstHalf, unsigned long long secondHalf, int playerToMove, int moveMade);

  // Constructs a StateMove with its properties defined by the parameters
  StateMove(Board *board, int movingPlayer, int moveTo);

  ˜StateMove(void);

  unsigned long long stateFirstHalf;
  unsigned long long stateSecondHalf;
  int player;
  // A move (in the format 'x+y*GRIDSIZE')
  int move;

  // RETURN: true iff this state and move is equal to other state and move
  bool isEqual(StateMove *other);

  // RETURN: true iff this state is equal to the state defined by
  // board and movingPlayer
  bool isEqualState(Board *board, int movingPlayer);
};
```

3.4.6 StateMove.cpp

```
#include ”stdafx.h”
#include ”StateMove.h”
#include ”Board.h”

// RETURN: x to the power i
inline unsigned long long power(int x, int i)
{
  unsigned long long answer = 1;
  for(int counter=0;counter<i;counter++) answer *= x;
  return answer;
}

inline unsigned long long value(int square);

StateMove::StateMove(Board *board, int movingPlayer, int moveTo)
{
  stateFirstHalf = 0;
  stateSecondHalf = 0;
  for(unsigned long long i=0; i<32 ;i++)
  {
    stateFirstHalf += power(3,i)*value(board->getSquare(i%GRIDSIZE,i/GRIDSIZE));
  }
  for(unsigned long long i=32; i<64 ;i++)
  {
    stateSecondHalf += power(3,i)*value(board->getSquare(i%GRIDSIZE,i/GRIDSIZE));
  }
  player = movingPlayer;
  move = moveTo;
}

StateMove::StateMove(unsigned long long firstHalf, unsigned long long secondHalf, int playerToMove, int moveMade)
{
  stateFirstHalf = firstHalf;
  stateSecondHalf = secondHalf;
  player = playerToMove;
  move = moveMade;
}

StateMove::StateMove(void)
{

}


StateMove::~StateMove(void)
{
}


bool StateMove::isEqual(StateMove *other)
{
  return (this->stateFirstHalf == other->stateFirstHalf && this->stateSecondHalf == other->stateSecondHalf&& this->move == other->move);
}

bool StateMove::isEqualState(Board *board, int movingPlayer)
{
  unsigned long long newStateFirstHalf, newStateSecondHalf;

  newStateFirstHalf = 0;
  newStateSecondHalf = 0;

  for(unsigned long long i=0; i<32 ; i++)
  {
    newStateFirstHalf += power(3,i)*value(board->getSquare(i%GRIDSIZE,i/GRIDSIZE));
  }
  for(unsigned long long i=32; i<64 ; i++)
  {
    newStateSecondHalf += power(3,i)*value(board->getSquare(i%GRIDSIZE,i/GRIDSIZE));
  }
  return (newStateFirstHalf == stateFirstHalf && newStateSecondHalf == stateSecondHalf && movingPlayer == player);
}

inline unsigned long long value(int square)
{
  if(square == WHITE)
    return 2;
  if(square == BLACK)
    return 1;
  return 0;
}
```

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:

```
cd Othello\Othello
g++ -fpermissive *.cpp
.\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:

```
cd othelloCmd/othelloCmd
g++ -fpermissive *.cpp
./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.