Artificial Intelligence

Winter 2016-2017

Reversi

Students:
Ilya Holtz: 319236931
Volodymyr Dzhuranyuk: 323792796

Lecturer: Prof. Larry Manevitz

Submission date: 04.01.2017

Source code and executable:
- On the usb drive in the attached envelope
- On Github public repository:
  https://github.com/holtzilya2008/reversiAI

Overview

This document Explains the solution to the Reversi game project, homework 1 of this semester. The main discussion will focus on the choice of the Heuristic function for the engine of the program. Also we will describe the most important part of our code in the context of Artificial Intelligence such as: The Implementation of the MiniMax algorithm and the Alpha-Beta pruning algorithm.

In addition, we will provide a small installation and user guide to the program. We will present the language and technologies used for building the solution, and we will give a brief description of the original source code of the GUI, witch was reused and modified in our solution.

Document structure

1. The problem - Brief descriprion

The main goal of this project is to implement the Reversi game, with a board 12x12, in a way that we could play competitive game with the computer, and the computer could play against itself. Therefore, for the game to be competitive, the computer must have a good strategy.

We build a small AI, that will have a good enough strategy to play against his opponent. In the AI course we learned so far about two searching algorithms (to go through all the options and to “think” several turns forward) and also how to use heuristics for making the best results.

Our goal is to Implement a simple GUI, and the AI engine. The engine will include the Implementation of the Mini-max and the Alpha-beta algorithms along wwith the heuristic function. To find the right heuristic function, we need to do some experiments and testing.

2. The heuristic function - Steps towards solution

To teach sombody how to play a board game, first you must become a decent player yourself. It would also be a good idea to play against someone who is really good skills, so that way you can learn from your opponent and from your own mistakes.

Step A. Playing and learning

We started exploring the web for an on-line or free implementations of the Reversi game, that allow you to practice human vs human or human vs computer.

The first thing we noticed that there are some valuable positions:

1) Corners: If The player capture the corners, the opponent cannot take them back, and the corner open opportunities to capture cells on the main diagonal, and the side rows. This way we made a conclusion that the corners have a very significant role in the strategy of a good player.

2) The cells around the corners are dangerous to capture, unless the corners are already captured. The most dangerous one is the cell on the main diagonal, next to the corner. If the player takes this cell, there is a very big probability that the corner will be captured by his opponent. This way we made a conclusion that we have to give a different weight to some cells on the board, positive or negative.

3) The number of ocupied cells on the board. at the beginning we gave great significance to that factor, because that was defining the winner, but soon we discovered that even if you have great amount of occupied cells on the board, this status well might change closer to the end of the game.

Step B. Teaching our computer

We cooperated with other students in the course on the GUI Implementation in python3, using the “tkinter” library. So we had a modular system to work on. at-first, we implemented the Mini-max algorithm, and wrote a simple heuristic function, and a weight function.

Experiment 1: At first we tried simply to count the number of occupied cells on the board at different depths. This was no good, and The AI lost because it didn't consider the corners issue.

Experiment 2: We added the cells weight function to the picture. we gave +2 weight to the corners, -1.5 to the cell near the corner on the main diagonal and -0.5 to the other cells around the corners. then all the other cells got 0.

This made more effect, but still the AI lost quite every game.

Experiment 3: Balancing and a good combination. We combined the factors of the cells weight and the number of occupied cells. We made the factor of the cells weight more important at the beginning of the game, and the factor of the occupied cells less important.

Then, when the number of captured cells on the board reached a certain point (136 cells), so that meant that we are close to the end, we made the opposite. The number of occupied cells becomes the most important factor at the end of the game. To implement this we used a coefficient that changed during the game.

This made a great change in the AI strategy. but still something was missing.

Experiment 4: We modified the cells weight function. We gave positive weight of +0.5 to the cells on the 3rd level around the corner. For example, if the corner is (1,1) we are talking about cells (3,3), (3,2), (2,3), (3,1), (1,3). Those cells are important until the corner is captured. When those cells are captured by the player, we build a situation where the opponent has to step on the dangerous ones around the corners.

This also made a good effect on the strategy. but still, in some cases we didn't understand why the AI made certain move, even when it made deep calculations. Still we missed something.

Experiment 5: Cutting diagonals - after playing a little more, we discovered that Its good strategy to try and capture diagonals, and if the enemy has a full block of captured cells then its good to cut it with a diagonal of our color and avoid forming full blocks until the end of the game.

We found that this strategy can be easily translated to minimizing the number of possible moves of the opponent. At the end, we added this strategy to our engine and balance it depending on the current game state (beginning or end). The exact values fo the coefficients fill be shown in the Implementation part.

After adding this strategy, we achieved the desired effect from the AI.

3. The heuristic function - Implementation

Here we present our final Implementation of the main heuristic function in Python. The following dictionary defines the constants we used for calculating the coefficient that will define the final value of the board, depending where are we in the game (GAME_BEGIN, GAME_LATE, GAME_END)

Here we present the final numbers that we used for best performance so far. No doubt, there is room for more experiments and tests with different numbers.

AI CONSTANTS DICTIONARY CODE (from ReversiAI.py)

```
AI_Constants = {
      'PIECES_TO_FULL_SEARCH': 130,
      'FULL_SEARCH_DEPTH': 8,
      'MAX_SEARCH_TREE_DEPTH': 3,
      'INFINITY': 10000,

      #The Heuristic function
      'GAME_BEGIN_PIECES_FACTOR': 0.2,
      'GAME_BEGIN_POSSIBLE_MOVES_FACTOR': 0.15,
      'GAME_BEGIN_WEIGHTS_FACTOR': 0.65,
      'GAME_LATE_PIECES': 138,
      'GAME_LATE_PIECES_FACTOR': 0.8,
      'GAME_LATE_POSSIBLE_MOVES_FACTOR': 0.1,
      'GAME_LATE_WEIGHTS_FACTOR': 0.1,
      'GAME_END_PIECES': 141,
      'GAME_END_PIECES_FACTOR': 1,
      'GAME_END_POSSIBLE_MOVES_FACTOR': 0,
      'GAME_END_WEIGHTS_FACTOR': 0
}
```

The following code is the entire implementation of the heuristic function h(). The function is used when we reach the bottom (`MAX_SEARCH_TREE_DEPTH or FULL_SEARCH_DEPTH`) of the Mini-max or Alpha-beta recursive search tree. Every significant part of the presented code is commented.

THE HEURISTIC FUNCTION CODE (from ReversiAI.py)

```
### The heuristic function h()
# This function is used to evaluate the board in it's current position
# @param board - The Liteboard on witch we make the evaluation
# @param playerAColor - string "black" or "white"
# @return - The value of the board in it's current position
#
# The Idea
# We use 3 parameters:
# 1) The number of ocupied cells of each player
# 2) The number of possible moves of each player
# 3) The weights of the cells on the board (explained in getCellWeight() in
#    ReversiBoard.py)
#
# Each of thease parameters has it's factor on the evaluation of the board at
# the current position. As we get closer to the end of the game, the number of
# cells becomes more important.
# In the begining and middle of the game, the main factor is the weight of
```

```
# the cells. The logic behind this, is that we should build a strong basis
# by using valueble cells. More about the weights of the cells is explained
# in ReversiBoard.py on the getCellWeight() function.
#
# If playerColor, doesn't have any ocupied cells at the current position, the
# function will return -AI_Constants['INFINITY'] (it means that he lost), on
# the other hand, if his opponent has no occupied cells, the function will
# return +AI_Constants['INFINITY'] (it means he won)
#
def h(board, playerAColor):

      #Counting black and white pieces on the liteBoard (auxiliary board)
      aPiecesCount = 0
      bPiecesCount = 0
      pieces = board.pieceCount()
      boardData = board.getBoardData()
      if playerAColor == "white":
            aPiecesCount = pieces[0]
            bPiecesCount = pieces[1]
      else:
            aPiecesCount = pieces[1]
            bPiecesCount = pieces[0]

      weights = board.calculateWeights(playerAColor)
      aWeights = weights[0]
      bWeights = weights[1]

      playerBColor = board.flip(playerAColor)

      #Counting the possible moves of each player
      aPossibleMoves = board.getPossMoves(playerAColor, playerBColor)
      if aPossibleMoves == "No moves":
            aPossibleMoves = 0
      else:
            aPossibleMoves = len(aPossibleMoves)

      bPossibleMoves = board.getPossMoves(playerBColor, playerAColor)
      if bPossibleMoves == "No moves":
            bPossibleMoves = 0
      else:
            bPossibleMoves = len(bPossibleMoves)

      # If tho opponent captured all our pieces, we lose. and the oposite.
      if aPiecesCount == 0:
            return -AI_Constants['INFINITY']
      elif bPiecesCount == 0:
            return AI_Constants['INFINITY']

      if aPiecesCount + bPiecesCount >= AI_Constants['GAME_LATE_PIECES']:

            piecesCoeff = AI_Constants['GAME_LATE_PIECES_FACTOR']
            posMovesCoeff = AI_Constants['GAME_LATE_POSSIBLE_MOVES_FACTOR']
            weightsCoeff = AI_Constants['GAME_LATE_WEIGHTS_FACTOR']

      elif aPiecesCount + bPiecesCount >= AI_Constants['GAME_END_PIECES']:

            piecesCoeff = AI_Constants['GAME_END_PIECES_FACTOR']
```

```
            posMovesCoeff = AI_Constants['GAME_END_POSSIBLE_MOVES_FACTOR']
            weightsCoeff = AI_Constants['GAME_END_WEIGHTS_FACTOR']

      else:

            piecesCoeff = AI_Constants['GAME_BEGIN_PIECES_FACTOR']
            posMovesCoeff = AI_Constants['GAME_BEGIN_POSSIBLE_MOVES_FACTOR']
            weightsCoeff = AI_Constants['GAME_BEGIN_WEIGHTS_FACTOR']

      value = piecesCoeff * (aPiecesCount - bPiecesCount) + weightsCoeff *
(aWeights - bWeights) + posMovesCoeff * (aPossibleMoves - bPossibleMoves)
      return value
```

The function that calculate the weights of the cells:

CELL WEIGHTS CALCULATION (from ReversiBoard.py)

```
# function - used by the heuristic function h() in reversi AI.py
# to determine the weight of a specific cell on the board
# @param cell : a list of two items (x,y)
# @param leftUpperCorner : *
# @param rightUpperCorner : *
# @param leftBottomCorner : *
# @param rightBottomCorner : *
#      * four boolean params that tell if the corners are captured
# @returnes : The weight of the given cell on the board
#
# The Idea:
# According to the strategy of the Reversi game, there are a better and worse
# positions that the player could be in.
# 1) Taking the Corners: this can be an advantage, because the opponent
#    can't take them back. each corner gives us opportunities to capture
#    cells on the main diagonal and the side row and column.
#    The corners will have value +2
# 2) The "cells near the corners" are dangerous, in case the corners
#      are not captured yet.
#    If we take for example the corner (1,1), we can are talking about
#    the cells : (1,2),(2,1),(2,2)
#    The near the corner on the  main diagonal will have a value of -3,
#    it is the most dangerous. (2,2) in our example.
# 3) Other cells near the corner will have value -1.5.
#    in our example (1,2), (2,1)
# 4) The cells in the "third level to the corner" are better then other cells
#    (in our example its: [1,3],[2,3],[3,3],[3,2],[3,1]) when we take them,
#    at the end we leave the oponnent no choise but to step on the dangerous
#    ones (the cells near the corner)
#    So we gice them a value of +0.5
#
def getCellWeight(cell, leftUpperCorner, rightUpperCorner, leftBottomCorner,
rightBottomCorner):
      CellWeights = {
            'CORNER': 2,
            'NEAR_CORNER_DIAGONAL': -3,
            'NEAR_CORNER_OTHER': -1.5,
            'THIRD_LEVEL_NEAR_CORNER': 0.5,
```

```
        'OTHER_CELLS': 0
    }
    # Checking corners: value +2
    if cell[0] in [1, 12] and cell[1] in [1, 12]:
        return CellWeights['CORNER']

    # Checking the "cells near the corners".
    # The cells on the main diagonal :
    # value NEAR_CORNER_DIAGONAL, unless the corner is captured
    elif cell[0] == 2 and cell[1] == 2:
        if leftUpperCorner == True:
            return CellWeights['OTHER_CELLS']
        else:
            return CellWeights['NEAR_CORNER_DIAGONAL']
    elif cell[0] == 11 and cell[1] == 11:
        if rightBottomCorner == True:
            return CellWeights['OTHER_CELLS']
        else:
            return CellWeights['NEAR_CORNER_DIAGONAL']
    elif cell[0] == 2 and cell[1] == 11:
        if rightUpperCorner == True:
            return CellWeights['OTHER_CELLS']
        else:
            return CellWeights['NEAR_CORNER_DIAGONAL']
    elif cell[0] == 11 and cell[1] == 2:
        if leftBottomCorner == True:
            return CellWeights['OTHER_CELLS']
        else:
            return CellWeights['NEAR_CORNER_DIAGONAL']

    # Other cells near the corners:
    # value NEAR_CORNER_OTHER, unless the corner is captured
    elif cell[0] == 1 and cell[1] == 2:
        if leftUpperCorner == True:
            return CellWeights['OTHER_CELLS']
        else:
            return CellWeights['NEAR_CORNER_OTHER']
    elif cell[0] == 1 and cell[1] == 11:
        if rightUpperCorner == True:
            return CellWeights['OTHER_CELLS']
        else:
            return CellWeights['NEAR_CORNER_OTHER']
    elif cell[0] == 2 and cell[1] == 1:
        if leftUpperCorner == True:
            return CellWeights['OTHER_CELLS']
        else:
            return CellWeights['NEAR_CORNER_OTHER']
    elif cell[0] == 2 and cell[1] == 12:
        if rightUpperCorner == True:
            return CellWeights['OTHER_CELLS']
        else:
            return CellWeights['NEAR_CORNER_OTHER']
    elif cell[0] == 11 and cell[1] == 1:
        if leftBottomCorner == True:
            return CellWeights['OTHER_CELLS']
        else:
            return CellWeights['NEAR_CORNER_OTHER']
```

```
    elif cell[0] == 11 and cell[1] == 12:
        if rightBottomCorner == True:
            return CellWeights['OTHER_CELLS']
        else:
            return CellWeights['NEAR_CORNER_OTHER']
    elif cell[0] == 12 and cell[1] == 2:
        if leftBottomCorner == True:
            return CellWeights['OTHER_CELLS']
        else:
            return CellWeights['NEAR_CORNER_OTHER']
    elif cell[0] == 12 and cell[1] == 11:
        if rightBottomCorner == True:
            return CellWeights['OTHER_CELLS']
        else:
            return CellWeights['NEAR_CORNER_OTHER']

    # cells in the "third level near the corner"
    # value THIRD_LEVEL_NEAR_CORNER, unless the corner is captured
    elif cell[0] == 3:
        if cell[1] in [1, 2, 3] and leftUpperCorner == False:
            return CellWeights['THIRD_LEVEL_NEAR_CORNER']
        elif cell[1] in [10, 11, 12] and rightUpperCorner == False:
            return CellWeights['THIRD_LEVEL_NEAR_CORNER']
        else:
            return CellWeights['OTHER_CELLS']
    elif cell[0] == 10:
        if cell[1] in [1, 2, 3] and leftBottomCorner == False:
            return CellWeights['THIRD_LEVEL_NEAR_CORNER']
        elif cell[1] in [10, 11, 12] and rightBottomCorner == False:
            return CellWeights['THIRD_LEVEL_NEAR_CORNER']
        else:
            return CellWeights['OTHER_CELLS']
    elif cell[1] == 3:
        if cell[0] in [1, 2] and leftUpperCorner == False:
            return CellWeights['THIRD_LEVEL_NEAR_CORNER']
        elif cell[0] in [11, 12] and leftBottomCorner == False:
            return CellWeights['THIRD_LEVEL_NEAR_CORNER']
        else:
            return CellWeights['OTHER_CELLS']
    elif cell[1] == 10:
        if cell[0] in [1, 2] and rightUpperCorner == False:
            return CellWeights['THIRD_LEVEL_NEAR_CORNER']
        elif cell[0] in [11, 12] and rightBottomCorner == False:
            return CellWeights['THIRD_LEVEL_NEAR_CORNER']
        else:
            return CellWeights['OTHER_CELLS']

    # All other cells on the board have the same weight.
    return CellWeights['OTHER_CELLS']
```

4. Mini-Max algorithm - Implementation

MINI-MAX IMPLEMENTATION CODE (from ReversiAI.py)

```
### Minimax Implementation:
#
# @param playerAColor - string "black" or "white"
# @param playerBColor - string "black" or "white"
# @param board - an object of class Board for the board representation
# @return - move, value: The move object is the cordinates of the next cell
# we should take (x,y) and the calculated value for this move
def getBestMinimaxMove(playerAColor, playerBColor, board, coefficients,
currentDepth, maxDepth):

      # determine if we can do "Full search"
      piecesNumber = board.piecesCount
      if piecesNumber >= AI_Constants['PIECES_TO_FULL_SEARCH'] and
AI_Constants['PIECES_TO_FULL_SEARCH'] > maxDepth:
            maxDepth = AI_Constants['FULL_SEARCH_DEPTH']


      # Translate the guiBoard to Liteboard
      liteBoard = LiteBoard(board,playerAColor,playerBColor)
      # Set Indicator that tells if the corners are captured or not
      liteBoard.setCorners()
      possibleMoves = liteBoard.getPossMoves(playerAColor, playerBColor)
      if possibleMoves == "No moves":
            return None, None

      # The Maximizer turn
      if currentDepth % 2 == 1:
            bestMoveResult = -AI_Constants['INFINITY']
            # loop - for each of the possible moves:
            for move in possibleMoves:
                  # apply the move on liteBoard2 (auxiliary board)
                  liteBoard2 = LiteBoard(board,playerAColor,playerBColor,
liteBoard)
                  liteBoard2.applyMove(move, playerAColor)

                  # If we reach the maximum search tree depth,
                  # evaluate the board using the heuristic function h()
                  if currentDepth == maxDepth:
                        moveResult = h(liteBoard2, playerAColor, coefficients)

                  # else - get down in the search tree (recursive call)
                  else:
                        move2, moveResult = getBestMinimaxMove(playerBColor,
playerAColor, liteBoard2, coefficients, currentDepth+1, maxDepth)
                        # If we don't have any possible moves in the further
search
                        # we want to evaluete the board in the current leveland
then
                        # compare it to the best option we already have
                        if move2 == None:
                              moveResult = h(liteBoard2, playerAColor,
coefficients)
                  if moveResult > bestMoveResult:
```

```
                        bestMoveResult = moveResult
                        bestMove = move

                  # Adding randomness
                  if moveResult == bestMoveResult:
                        randomNumber = randint(0,1)
                        if randomNumber == 1:
                              bestMoveResult = moveResult
                              bestMove = move

      # The Minimizer turn
      else:
            bestMoveResult = AI_Constants['INFINITY']
            # loop - for each of the possible moves:
            for move in possibleMoves:
                  # apply the move on liteBoard2 (auxiliary board)
                  liteBoard2 = LiteBoard(board,playerAColor,playerBColor,
liteBoard)
                  liteBoard2.applyMove(move, playerAColor)

                  # If we reach the maximum search tree depth,
                  # evaluate the board using the heuristic function h()
                  if currentDepth == maxDepth:
                        moveResult = h(liteBoard2, playerBColor, coefficients)

                  # else - get down in the search tree (recursive call)
                  else:
                        move2, moveResult = getBestMinimaxMove(playerBColor,
playerAColor, liteBoard2, coefficients, currentDepth+1, maxDepth)
                        # If we don't have any possible moves in the further
search
                        # we want to evaluete the board in the current level and
then
                        # compare it to the best option we already have
                        if move2 == None:
                              moveResult = h(liteBoard2, playerBColor,
coefficients)
                  if moveResult < bestMoveResult:
                        bestMoveResult = moveResult
                        bestMove = move

                  # Adding randomness
                  if moveResult == bestMoveResult:
                        randomNumber = randint(0,1)
                        if randomNumber == 1:
                              bestMoveResult = moveResult
                              bestMove = move

      return bestMove, bestMoveResult

### End of Minimax Implementation
```

5. Alpha-beta pruning algorithm - Implementation

ALPHA-BETA IMPLEMENTATION CODE (from ReversiAI.py)

```
### Alphabeta Implementation:
#
# @param playerAColor - string "black" or "white"
# @param playerBColor - string "black" or "white"
# @param board - an object of class Board for the board representation
# @param depth - defines in witch level of the search tree we are now
# @param alpha - best already explored option, along the path to the root
# for the Maximizer
# @param beta - best already explored option, along the path to the root
# for the Minimizer
# @return - move, value: The move object is the cordinates of the next cell
# we should take (x,y) and the calculated value for this move
def getBestAlphaBetaMove(playerAColor, playerBColor, board, coefficients,
currentDepth, maxDepth, alpha, beta):

      # determine if we can do "Full search"
      piecesNumber = board.piecesCount
      if piecesNumber >= AI_Constants['PIECES_TO_FULL_SEARCH']:
            maxDepth = AI_Constants['FULL_SEARCH_DEPTH']


      # Translate the guiBoard to Liteboard
      liteBoard = LiteBoard(board,playerAColor,playerBColor)
      # Set Indicator that tells if the corners are captured or not
      liteBoard.setCorners()
      possibleMoves = liteBoard.getPossMoves(playerAColor, playerBColor)
      if possibleMoves == "No moves":
            return None, None

      # The MAximizer turn
      if currentDepth % 2 == 1:

            bestMoveResult = alpha
            bestMove = None

            # loop - for each of the possible moves:
            for move in possibleMoves:
                  # apply the move on liteBoard2 (auxiliary board)
                  liteBoard2 = LiteBoard(board,playerAColor,playerBColor,
liteBoard)
                  liteBoard2.applyMove(move, playerAColor)

                  # If we reach the maximum search tree depth,
                  # evaluate the board using the heuristic function h()
                  if currentDepth == maxDepth:
                        moveResult = h(liteBoard2, playerAColor, coefficients)

                  # else - get down in the search tree (recursive call)
                  else:
                        move2, moveResult = getBestAlphaBetaMove(playerBColor,
playerAColor, liteBoard2, coefficients, currentDepth+1, maxDepth, alpha,
```

```
bestMoveResult)
                        # If we don't have any possible moves in the further
search
                        # we want to evaluete the board in the current level and
then
                        # compare it to the best option we already have
                        if move2 == None:
                              moveResult = h(liteBoard2, playerAColor,
coefficients)

                  # AlphaBeta pruning : cut off the branch if the current
                  # result greater then beta
                  if moveResult >= beta:
                        return move, moveResult

                  if moveResult > bestMoveResult:
                        bestMoveResult = moveResult
                        bestMove = move

                  # Adding randomness
                  if moveResult == bestMoveResult:
                        randomNumber = randint(0,1)
                        if randomNumber == 1:
                              bestMoveResult = moveResult
                              bestMove = move

      # The Minimizer turn
      else:
            bestMoveResult = beta
            bestMove = None
            # loop - for each of the possible moves:
            for move in possibleMoves:
                  # apply the move on liteBoard2 (auxiliary board)
                  liteBoard2 = LiteBoard(board,playerAColor,playerBColor,
liteBoard)
                  liteBoard2.applyMove(move, playerAColor)

                  # If we reach the maximum search tree depth,
                  # evaluate the board using the heuristic function h()
                  if currentDepth == maxDepth:
                        moveResult = h(liteBoard2, playerAColor, coefficients)

                  # else - get down in the search tree (recursive call)
                  else:
                        move2, moveResult = getBestAlphaBetaMove(playerBColor,
playerAColor, liteBoard2, coefficients, currentDepth+1, maxDepth, bestMoveResult,
beta)
                        # If we don't have any possible moves in the further
search
                        # we want to evaluete the board in the current level and
then
                        # compare it to the best option we already have
                        if move2 == None:
                              moveResult = h(liteBoard2, playerBColor,
coefficients)

                  # AlphaBeta pruning : cut off the branch if the current
```

```
                  # result greater then beta
                  if moveResult <= alpha:
                        return move, moveResult

                  if moveResult < bestMoveResult:
                        bestMoveResult = moveResult
                        bestMove = move

                  # Adding randomness
                  if moveResult == bestMoveResult:
                        randomNumber = randint(0,1)
                        if randomNumber == 1:
                              bestMoveResult = moveResult
                              bestMove = move

      return bestMove, bestMoveResult
### End of Alphabeta Implementation
```

6. Discussion and conclusion

As we seen there are a lot of different options for the heuristic function. each option gives us the impression that the computer plays better or worse. There are many experiments we can do to try and achieve the best performance. The experiments include not only changing the heuristics but also playing with the depth of the search tree.

Our Implementation prooves the fact that the choice of the searching algorithm (Mini-max or Alpha-beta) plays an important role on the preformance of our program. We had mesured time on a position in the middle of the game, with givven depth 5 and aproximately 10-12 possible moves for the next turn, with both Mini-max and Alpha-beta. We noticed that It took for the Mini-max algorithm aproximately 20 minutes, when the next turn calculated with Alpha-beta in aproximately 5 minutes.

Also The data structures used for the project implementation are important for the program working faster. This priject was not exactly focusing on fast performance, but when we are talking about search through thousends of different board combinations, the working speed is sensable. To find the perfect heuristic function, we should make a lot of tests, and if each turn takes the computer around 5 minutes to perform, the research would take a lot of time.

7. Technical overview

We implemented our project using python3. We took the GUI source code, main part of witch was already written by other students, and modified it to serve our needs. The GUI was written using the Tkinter library.

The main part of our work focused on ReversiBoard.py witch is the Liteboard implementation (The auxiliary board that we used for the search and evaluation), and ReversiAI.py. witch includes the heuristic function, the Mini-max and the Alpha-beta algorithms.

We used Git for version control. The repository is available on Github. We made public access to the repository as soon as we finished writing the code and the documentation. All stages of development could be inspected in the commit history.

8. Installation and using guide

The windows executable ReversiMain.exe, along with all necessary files to run the program, are submitted on usb flash drive, zipped into reversi.zip. It doesn't require an installation to the windows regestry.

Beside the zip file, you will find the /source folder with the source code and the docs.

Usage:
0) unzip reversi.zip to any desired place (e.g C:\tmp\reversi)
1)Run the ReversiMain.exe
2) Choose one of 3 desired modes to play the game:
    2.1 Human vs AI
    2.2 Human vs AI
    2.3 AI vs AI
4) For each of the desired options, you will be guided to make a choice of the searching algorithm (Mini-max or Alpha-beta), the search tree depth, and the parameters for the Heuristic function you desire to experiment with.

The Default Heuristic function is h(), as documented above.