# 

# **Report Assignment 2**

Artificial Intelligence



Daniel Varela Sánchez

Guillermo Martín-Coello Juárez

\*\* All work done for “Part I” of the assignment is considered in 1.b.iii. Computer independent measures of improvement.

**1. Documentation of minimax + alpha-beta pruning**

**A. IMPLEMENTATION DETAILS**

**i. Which tests have been designed and applied to determine whether the implementation is correct?**

To test the implementation of minimax algorithm with alpha-beta pruning was correct we made three different independent tests:

First, we timed the absolute time of the used function and compared it with the minimax algorithm without alpha-beta pruning.

The second test consists of manually running the minimax algorithm with alpha-beta pruning on a simple tree and comparing the results with the implemented algorithm.

Finally, we compared the number of iterations made by the minimax algorithm with alpha-beta pruning and the minimax algorithm without alpha-beta pruning.

**ii. Design: Data structures selected, functional decomposition, etc.**

A class MinimaxAlphaBetaStrategy was created. For the implementation of a minimax algorithm with alpha-beta pruning. This class contains the heuristic, the max depth, and the verbose. Methods of this class are next move, min value and max value. These methods use the current state, current depth and the a (alpha) and b (beta) values to compute the minimax algorithm with alpha-beta pruning.

|  |
| --- |

**iii. Implementation.**

We followed the same implementation as in the MinMax class by filling out the next state, the max value and the min value methods only in this case we had to adapt the code to make it work with pruning which would mean to add alpha and beta values and checking for their correction.

**iv. Other relevant information.**

A class MinimaxAlphaBetaStrategy was created with inheritance of the strategy class. This is necessary for the program to use the algorithm.

**B. EFFICIENCY OF ALPHA-BETA PRUNING.**

**i. Complete description of the evaluation protocol.**

To evaluate the implementation of the minimax algorithm with alpha-beta pruning, as previously mentioned, we used three different methods, consisting of absolute times and computer independent measurements of improvement.

**ii. Tables in which times with and without pruning are reported.**

The implementation of the minimax algorithm with alpha-beta pruning was executed 10 times measuring the time and compared with the execution of the implementation of the minimax algorithm without alpha-beta pruning, also executed 10 times. A constant heuristic was used to provide the best comparison possible. As it is shown, the absolute time without alpha-beta pruning is higher than the time with alpha-beta pruning. This is not a perfect measure and can vary depending on the hardware and other factors, so other independent measurements must be done.

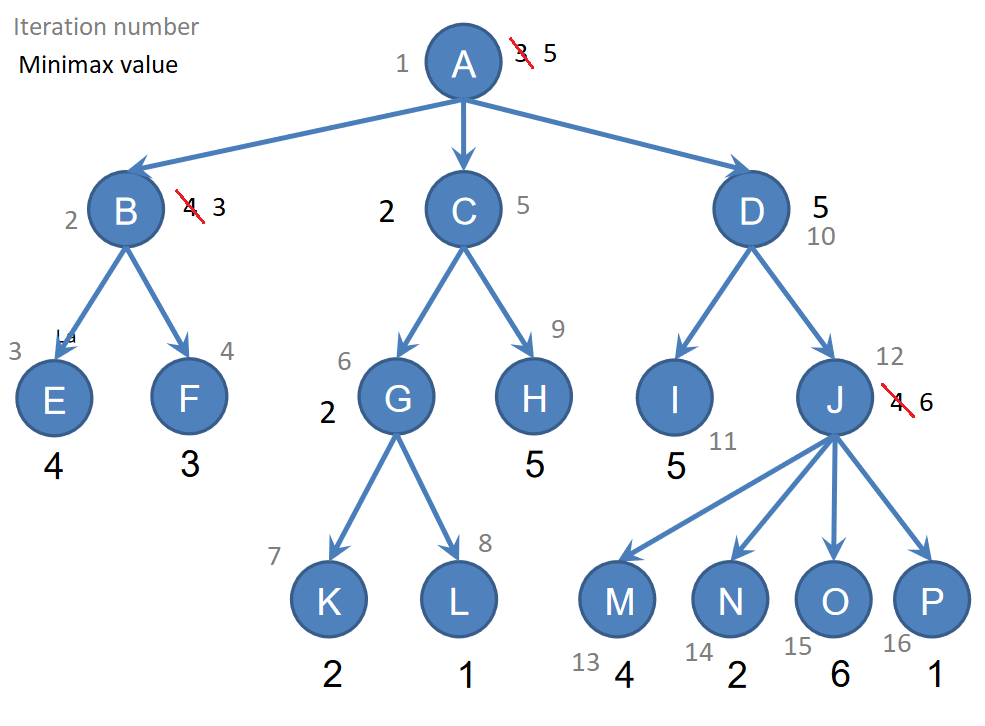
|  |  |
| --- | --- |
| Image I: Time of executing 10 times demo\_reversi with the minimax algorithm without alpha-beta pruning (in both players) | Image II: Time of 10 demo\_reversi executions with minimax algorithm with alpha-beta pruning (in both players) |

**iii. Computer independent measures of improvement.**

Computer independent measures of improvement were done to assure the implementation of the minimax algorithm with alpha-beta pruning was correct.

(i) Apply manually the minimax algorithm. On a copy of the tree, identify the iteration number, the node that is being visited, and the estimation of the minimax value for that node in each step of the algorithm.

The minimax algorithm travels through each node of the tree finding the biggest possible son node value in case of MAX and the smallest possible son node value in case of MIN. This value is taken by the parent node.



(ii) Check that the manual execution agrees with the one performed by the minimax player in demo\_simple\_game\_tree.py that starts the match. For this, include in the report the corresponding information to the first move of the game with verbose=3.

The minimax execution performed by the minimax player in demo\_simple\_game\_tree looks just like the manual one. As it is shown, the first node visited (after the root) is node B, where first the algorithm explores E and picks 4, and then it explores F and changes the 4 to the value 3 (MIN is playing). The new value goes to B and then to A and it continues exploring.

|  |
| --- |

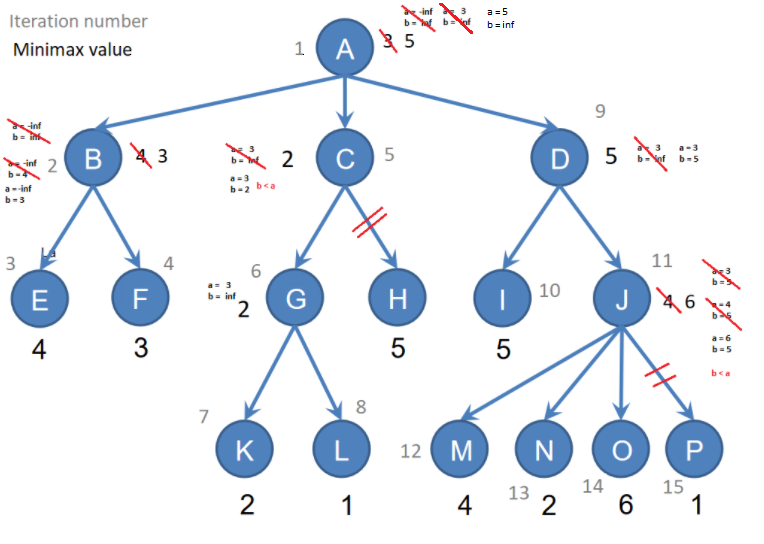
(iii) Apply manually the minimax algorithm with alpha-beta pruning. On a copy of the tree, identify

the iteration number, the node that is being visited, and the estimation of the interval [, ] for

that node in each step of the algorithm. Finally, show the minimax value in the root node and the

sequence of moves until the end of the game.

The minimax algorithm is manually applied again, this time with alpha beta pruning. This method cuts trivial nodes to provide a faster result.



(iv) Check that the manual execution agrees with the one performed by the minimax player that uses alpha-beta pruning in demo\_simple\_game\_tree.py when starting the match. To this end, include in the report the sequence of values of the [, ] interval associated to the nodes visited at each iteration of the algorithm.

Finally, the minimax player execution that uses alpha-beta pruning shows the expected results, just as shown in the previous image. This execution doesn’t travel through all nodes, because some of them with the knowledge of their brother nodes value.

|  |
| --- |

**iv. Correct, clear, and complete analysis of the results.**

The time comparison resulted in a faster result for the minimax algorithm with alpha-beta pruning, as expected. This can be explained because the pruning done reduces the number of nodes traveled.

The algorithm comparison with a manual interpretation resulted in the same path taken by both implementations, but with some advantage in process execution by the implementation of a minimax algorithm with alpha-beta pruning, which doesn’t visit trivial nodes.

Finally we compared the number of iterations of each algorithm to find the difference between them.

As a result, we can see the implementation of a minimax algorithm with alpha-beta pruning improves performance and speed.

**v. Other relevant information.**

It is important to know that the improvements obtained by the implementation of minimax algorithm with alpha-beta pruning may vary depending on the circumstances and order the nodes are in. This could end up pushing the limits of the extreme that the implementation of minimax algorithm with alpha-beta pruning could end up being slower than the implementation of minimax algorithm without alpha-beta pruning (As the latter has less code and logic behind).

**2. Documentation of the design of the heuristic.**

**A. REVIEW OF PREVIOUS WORK ON REVERSI STRATEGIES, INCLUDING REFERENCES IN APA FORMAT.**

Buro, M. (1995, March). Logistello: A strong learning othello program. In *19th Annual Conference Gesellschaft für Klassifikation eV* (Vol. 2).

Thomas Wolf. (2000, August 2). The anatomy of a game program. Inside Reversi/Othello. Retrieved March 24, 2022, from http://home.datacomm.ch/t\_wolf/tw/misc/reversi/html/index.html

William A. Greene. (n.d.). Machine learning of Othello heuristics Retrieved March 24, 2022, from https://compsci.sites.tjhsst.edu/ai/othello\_heuristics.pdf

**B. DESCRIPTION OF THE DESIGN PROCESS:**

**i. How was the design process planned and realized?**

We started by reading the PDF and getting to know how the game worked. To do so, we played a few games of Othello online between us. Once we knew how the game worked we started by brainstorming ideas on what were the important factors in the game that could give our heuristics a win. After all this process we looked at the code and filtered those that were feasible and when we had those, we started writing code to implement them starting with the basic ones and ending with more complex ones.

**ii. Did you have a systematic procedure to evaluate the heuristics designed?**

Yes, when it came to testing our heuristics, we added them to the demo\_tournament file in the heuristic classes that were given to us. We changed the board size to an 8x8 to make it exactly like the game and all we had to do was execute the file. At the start we competed with the heuristics that were given to us (which were basic and just to try), but once we had more heuristics, we started testing between the ones previously created. That made the differences between the effectiveness of them become less and less so we had to start incrementing the amount of games played to have more precise measurements of the heuristics.

**iii. Did you take advantage of strategies developed by others? If these are publicly available, provide references in APA format; otherwise, include the name of the person who provided the information and give proper credit of the contribution as “private communication”.**

Yes, we investigated previous work done on the othello heuristics as previously said on part 2.a

**c. Description of the final heuristic submitted.**

Our heuristic was based on four different pillars:

Stability

Mobility

Corners

Coin parity

First let’s start with the stability, in this part we measure how stable is that position meaning how hard it is for the opponent to steal it, the more stable the position, the higher the punctuation. To do this we analyze the board and check for the weak and strong spots (for example the corners are very strong positions and the ones next to it are the weakest because you allow the opponent to take the corners).

When it comes to the mobility, we measure the proportion of moves it gives to our player and the ones taken from the enemy, rewarding this way those moves that give us more moves while taking options to the enemy.

As said in the stability, corners play a big part in the strategy of this game since they are positions that once conquered, the enemy can not take away, this is why there is an extra field of the heuristic that makes it keen to take the corner at all costs,

In the end the most important part, the coin parity, is the amount of coin difference, and it rewards the moves that give more coins to our player and take more coins to the enemy since this is the final goal of the game (to have more coins than the enemy at the end).

Nevertheless these are all parts of the heuristic and the best way to make them work is by combining them, to do so, we arrived at the conclusion that the best way was to instead of putting everything together at once, to prioritize some strategies depending on the state of the game (initial, medium or advanced state). To do so, we get the current move of the game and since there are as many moves as cells on the board, we are able to know the state of the game and make some adjustments to the overall strategy.

| class Heuristic(StudentHeuristic):  def get\_name(self) -> str:  return "h3"    def evaluation\_function(self, state: TwoPlayerGameState) -> float:  t =len(state.board)  #print("mob = "+ str(self.mobility(state))+"\ncor = "+str(self.corners(state))+"\ncoins = "+str(self.coin\_parity(state))+"\n")  if t <= 20:  return self.mobility(state) + self.corners(state) + 0.6 \*self.stability(state)  elif t <= 40:  return self.mobility(state) + self.corners(state) + self.stability(state) + .6 \* self.coin\_parity(state)  else:  return 0.5 \* self.mobility(state) + self.corners(state) + 0.6 \*self.stability(state) + self.coin\_parity(state)  # COIN DIFFERENCE:  ##########################################################  def coin\_parity(self, state: TwoPlayerGameState) -> float:  """Difference in the number of coins."""  scores = state.scores  assert isinstance(scores, (Sequence, np.ndarray))  if state.is\_player\_max(state.player1):  return 100\*( (state.scores[0] - state.scores[1])/(state.scores[0] + state.scores[1]))  elif state.is\_player\_max(state.player2):  return 100\*( (state.scores[1] - state.scores[0])/(state.scores[0] + state.scores[1]))  else:  raise ValueError('Player MAX not defined')  # CHOICE DIFFERENCE:  ##########################################################  def mobility(self, state: TwoPlayerGameState) -> float:  """Difference in the number of choices available."""  black\_moves\_num = len(self.availableMoves(state, state.player1.label))  white\_moves\_num = len(self.availableMoves(state, state.player2.label))  if (black\_moves\_num + white\_moves\_num) != 0:  if state.is\_player\_max(state.player1):  #print("w: "+str(white\_moves\_num)+" -- b: " + str(black\_moves\_num))  return 100 \* (black\_moves\_num - white\_moves\_num) / (black\_moves\_num + white\_moves\_num)  if state.is\_player\_max(state.player2):  #print("w: "+str(white\_moves\_num)+" -- b: " + str(black\_moves\_num))  return 100 \* (white\_moves\_num - black\_moves\_num) / (black\_moves\_num + white\_moves\_num)  else:  return 0  return 0  def enemiesInMov(self, state: TwoPlayerGameState, move, p: Any, delta\_x\_y) -> list:  enemy = state.player2.label if p==state.player1.label else state.player1.label  (delta\_x, delta\_y) = delta\_x\_y  x, y = move  x, y = x + delta\_x, y + delta\_y  enemy\_list\_0 = []  while state.board.get((x, y)) == enemy:  enemy\_list\_0.append((x, y))  x, y = x + delta\_x, y + delta\_y  if state.board.get((x, y)) != p:  del enemy\_list\_0[:]  x, y = move  x, y = x - delta\_x, y - delta\_y  enemy\_list\_1 = []  while state.board.get((x, y)) == enemy:  enemy\_list\_1.append((x, y))  x, y = x - delta\_x, y - delta\_y  if state.board.get((x, y)) != p:  del enemy\_list\_1[:]  return enemy\_list\_0 + enemy\_list\_1  def enemiesCaptured(self, state: TwoPlayerGameState, move, p: Any) -> list:  return self.enemiesInMov(state, move, p, (0, 1)) \  + self.enemiesInMov(state, move, p, (1, 0)) \  + self.enemiesInMov(state, move, p, (1, -1)) \  + self.enemiesInMov(state, move, p, (1, 1))  def availableMoves(self, state: TwoPlayerGameState, p: Any) -> list:  """Returns a list of valid moves for the player judging from the board."""  return [(x, y) for x in range(1, 8 + 1)  for y in range(1, 8 + 1)  if (x, y) not in state.board.keys() and  self.enemiesCaptured(state, (x, y), p)]  # CORNER DIFFERENCE:  ##########################################################  def corners(self, state: TwoPlayerGameState) -> float:  """  Difference in the number of corners captured.  """  corner = [state.board.get((1, 1)), state.board.get((1, 8)), state.board.get((8, 1)),  state.board.get((8, 8))]    black\_corner = corner.count(state.player1.label)  white\_corner = corner.count(state.player2.label)  if (black\_corner + white\_corner) != 0:  if state.is\_player\_max(state.player1):  return 100 \* (black\_corner - white\_corner) / (black\_corner + white\_corner)  if state.is\_player\_max(state.player2):  return 100 \* (white\_corner - black\_corner) / (black\_corner + white\_corner)    return 0  def stability(self, state: TwoPlayerGameState) -> float:  if not state.move\_code:  return 0  new\_piece\_position = self.letter\_to\_number(state.move\_code[4]) + (int(state.move\_code[1])-1)\*8    # Check if the new piece is in the corner  if new\_piece\_position in [0, 7, 56, 63]:  return 200  # Check if the new piece is close to corner  elif new\_piece\_position in [1, 6, 8, 15, 48, 55, 57, 62]:  return -10  # Check if the new piece is diagonal to corner  elif new\_piece\_position in [9, 14, 49, 54]:  return -20  # Check if the new piece is in the wall  elif new\_piece\_position in [2, 3, 4, 5, 16, 23, 24, 31, 32, 39, 40, 47, 58, 59, 60, 61]:  return 10  # Check if the new piece is in the second circle from exterior  elif new\_piece\_position in [10, 11, 12, 13, 17, 22, 25, 30, 33, 38, 41, 46, 50, 51, 52, 53]:  return 1  # Check if the new piece is in the middle  elif new\_piece\_position in [18, 19, 20, 21, 26, 29, 34, 37, 42, 43, 44, 45]:  return 5  # CHeck if the new piece is in the center  elif new\_piece\_position in [27, 28, 35, 36]:  return 2    else:  return 0  def letter\_to\_number(self, letter: str) -> int:  """Converts a letter to a number.  Args:  letter (str): The letter to convert.  Returns:  int: The number corresponding to the letter.  """  return ord(letter) - ord("a")  def create\_match(player1: Player, player2: Player) -> TwoPlayerMatch:  initial\_board = None#np.zeros((dim\_board, dim\_board))  initial\_player = player1  """game = TicTacToe(  player1=player1,  player2=player2,  dim\_board=dim\_board,  )"""  initial\_board = (  ['........',  '........',  '........',  '...BW...',  '...WB...',  '........',  '........',  '........']  )  if initial\_board is None:  height, width = 8, 8  else:  height = len(initial\_board)  width = len(initial\_board[0])  try:  initial\_board = from\_array\_to\_dictionary\_board(initial\_board)  except ValueError:  raise ValueError('Wrong configuration of the board')  else:  print("Successfully initialised board from array")  game = Reversi(  player1=player1,  player2=player2,  height=8,  width=8  )  game\_state = TwoPlayerGameState(  game=game,  board=initial\_board,  initial\_player=initial\_player,  )  return TwoPlayerMatch(game\_state, max\_seconds\_per\_move=1000, gui=False)  tour = Tournament(max\_depth=3, init\_match=create\_match)  strats = {'opt1': [Heuristic1], 'opt2': [Heuristic2], 'opt3': [Heuristic3]}  n = 2  scores, totals, names = tour.run(  student\_strategies=strats,  increasing\_depth=False,  n\_pairs=n,  allow\_selfmatch=False,  )  print(  'Results for tournament where each game is repeated '  + '%d=%dx2 times, alternating colors for each player' % (2 \* n, n),  )  # print(totals)  # print(scores)  print('\ttotal:', end='')  for name1 in names:  print('\t%s' % (name1), end='')  print()  for name1 in names:  print('%s\t%d:' % (name1, totals[name1]), end='')  for name2 in names:  if name1 == name2:  print('\t---', end='')  else:  print('\t%d' % (scores[name1][name2]), end='')  print() | |
| --- | --- |