ARTIFICIAL INTELLIGENCE MEMORY P2

Done by: José Manuel López-Serrano Tapia, Sergio Hernández Martín

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?

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

    iii. Implementation.

    iv. Other relevant information.

    b. **Efficiency of alpha-beta pruning.**

    i. Complete description of the evaluation protocol.

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

    iii. Computer independent measures of improvement.

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

    v. Other relevant information.

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

    a. **Review of previous work on Reversi strategies, including references in APA format.**

    b. **Description of the design process:**

    i. How was the design process planned and realized?

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

    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".

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

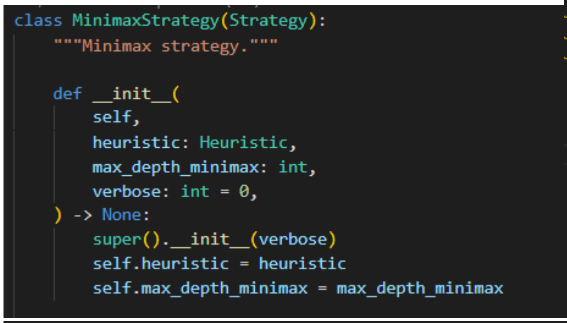
    d. **Other relevant information**

1. Minimax + alpha-beta pruning

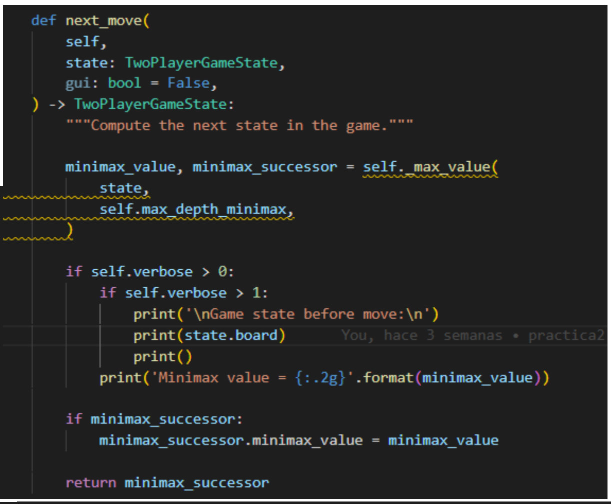
Code

The minimax algorithm simulates a two-person game, player max and player min, in which they are adversaries. Generating a game tree of a certain depth with every possible game configuration, the algorithm performs a depth-first search to extract the next optimal movement. The leaves are assigned a value (the higher it is the more desirable it is for player max) with either the utility function (if the game ends) or an evaluation function (depth limit is reached). Since player max and player min take turns playing, player max will choose the maximum value of its successors, while player min chooses the minimum value. From the leaves the algorithm works recursively until the final state with its value reaches the root.

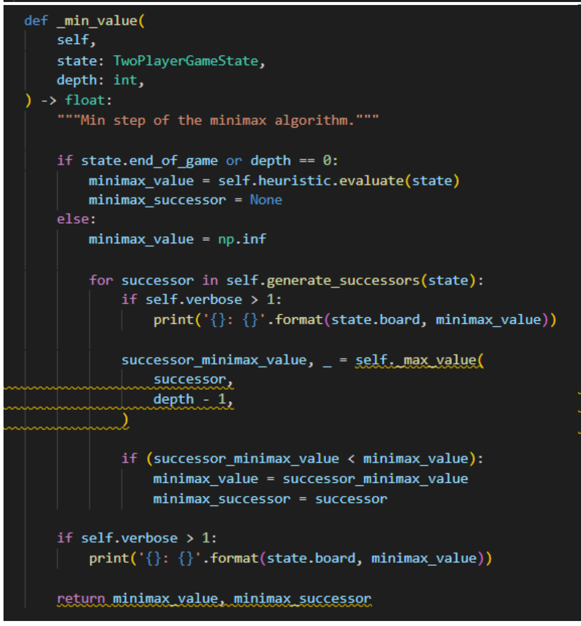
Although minimax without alpha-beta pruning was already implemented, we will add screenshots of the implementation since we based our own later code on it.



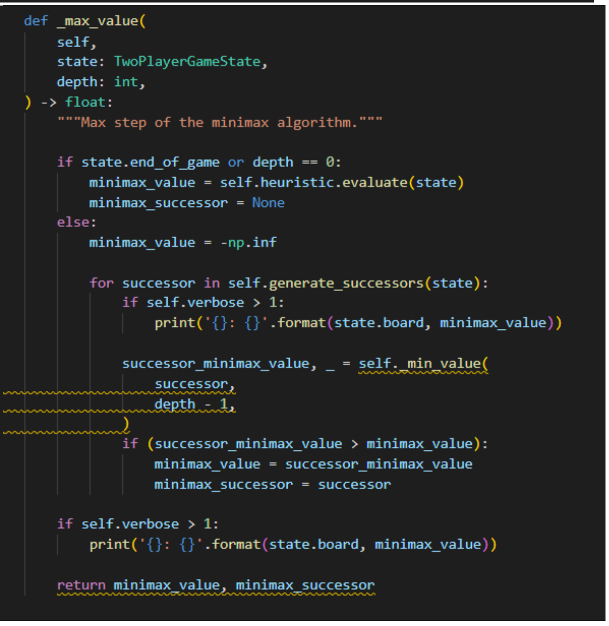
```python
class MinimaxStrategy(Strategy):
    """Minimax strategy."""

    def __init__(
        self,
        heuristic: Heuristic,
        max_depth_minimax: int,
        verbose: int = 0,
    ) -> None:
        super().__init__(verbose)
        self.heuristic = heuristic
        self.max_depth_minimax = max_depth_minimax
```



```python
def next_move(
    self,
    state: TwoPlayerGameState,
    gui: bool = False,
) -> TwoPlayerGameState:
    """Compute the next state in the game."""

    minimax_value, minimax_successor = self._max_value(
        state,
        self.max_depth_minimax,
    )

    if self.verbose > 0:
        if self.verbose > 1:
            print('\nGame state before move:\n')
            print(state.board)        You, hace 3 semanas • practica2
            print()
        print('Minimax value = {:.2g}'.format(minimax_value))

    if minimax_successor:
        minimax_successor.minimax_value = minimax_value

    return minimax_successor
```



```python
def _min_value(
    self,
    state: TwoPlayerGameState,
    depth: int,
) -> float:
    """Min step of the minimax algorithm."""

    if state.end_of_game or depth == 0:
        minimax_value = self.heuristic.evaluate(state)
        minimax_successor = None
    else:
        minimax_value = np.inf

        for successor in self.generate_successors(state):
            if self.verbose > 1:
                print('{}: {}'.format(state.board, minimax_value))

            successor_minimax_value, _ = self._max_value(
                successor,
                depth - 1,
            )

            if (successor_minimax_value < minimax_value):
                minimax_value = successor_minimax_value
                minimax_successor = successor

    if self.verbose > 1:
        print('{}: {}'.format(state.board, minimax_value))

    return minimax_value, minimax_successor
```

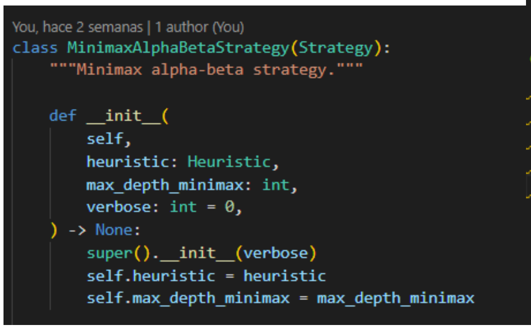


```python
def _max_value(
    self,
    state: TwoPlayerGameState,
    depth: int,
) -> float:
    """Max step of the minimax algorithm."""

    if state.end_of_game or depth == 0:
        minimax_value = self.heuristic.evaluate(state)
        minimax_successor = None
    else:
        minimax_value = -np.inf

        for successor in self.generate_successors(state):
            if self.verbose > 1:
                print('{}: {}'.format(state.board, minimax_value))

            successor_minimax_value, _ = self._min_value(
                successor,
                depth - 1,
            )
            if (successor_minimax_value > minimax_value):
                minimax_value = successor_minimax_value
                minimax_successor = successor

    if self.verbose > 1:
        print('{}: {}'.format(state.board, minimax_value))

    return minimax_value, minimax_successor
```

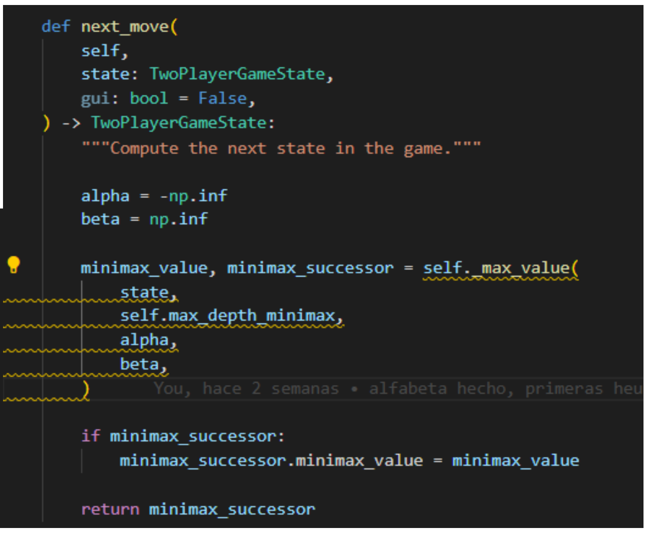
Note the prints in the min_value and max_value functions: for later tests in demo_simple_game_tree.py they will provide the information we need to test the algorithm.

Alpha-beta pruning is added to minimax to improve performance: both will get to the same value but alpha-beta pruning optimizes time. The general explanation is that alpha-beta pruning will not explore successors we already know will not get chosen by the minimax algorithm. This is possible by passing two values, alpha and beta, in each recursive call to min_value and max_value. If at any time alpha >= beta then all successors that are left to explored will be ignored (pruned).

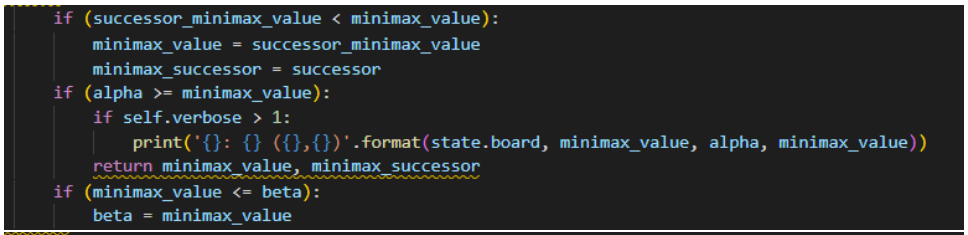
Here is our code for the alpha-beta implementation.



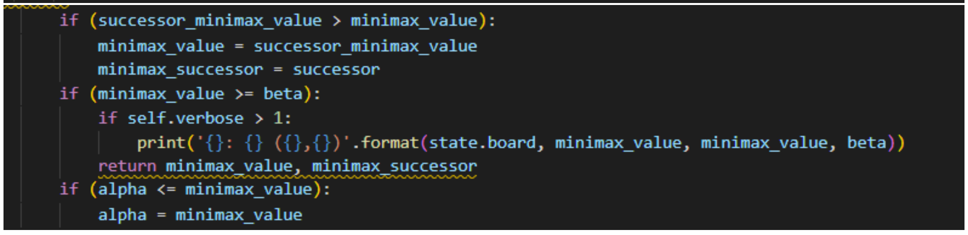
```
You, hace 2 semanas | 1 author (You)
class MinimaxAlphaBetaStrategy(Strategy):
    """Minimax alpha-beta strategy."""

    def __init__(
        self,
        heuristic: Heuristic,
        max_depth_minimax: int,
        verbose: int = 0,
    ) -> None:
        super().__init__(verbose)
        self.heuristic = heuristic
        self.max_depth_minimax = max_depth_minimax
```



```
def next_move(
    self,
    state: TwoPlayerGameState,
    gui: bool = False,
) -> TwoPlayerGameState:
    """Compute the next state in the game."""

    alpha = -np.inf
    beta = np.inf

    minimax_value, minimax_successor = self._max_value(
        state,
        self.max_depth_minimax,
        alpha,
        beta,
    )        You, hace 2 semanas • alfabeta hecho, primeras heu

    if minimax_successor:
        minimax_successor.minimax_value = minimax_value

    return minimax_successor
```



```
    if (successor_minimax_value < minimax_value):
        minimax_value = successor_minimax_value
        minimax_successor = successor
    if (alpha >= minimax_value):
        if self.verbose > 1:
            print('{}: {} ({},{})'.format(state.board, minimax_value, alpha, minimax_value))
        return minimax_value, minimax_successor
    if (minimax_value <= beta):
        beta = minimax_value
```

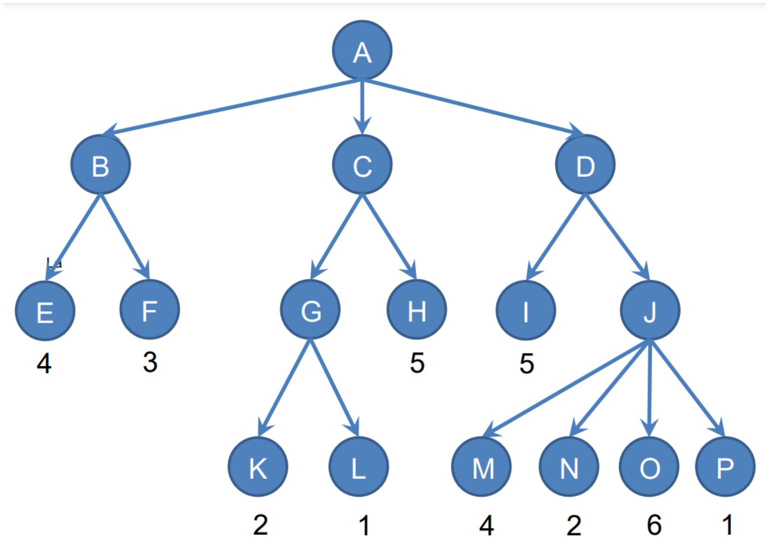


```
    if (successor_minimax_value > minimax_value):
        minimax_value = successor_minimax_value
        minimax_successor = successor
    if (minimax_value >= beta):
        if self.verbose > 1:
            print('{}: {} ({},{})'.format(state.board, minimax_value, minimax_value, beta))
        return minimax_value, minimax_successor
    if (alpha <= minimax_value):
        alpha = minimax_value
```

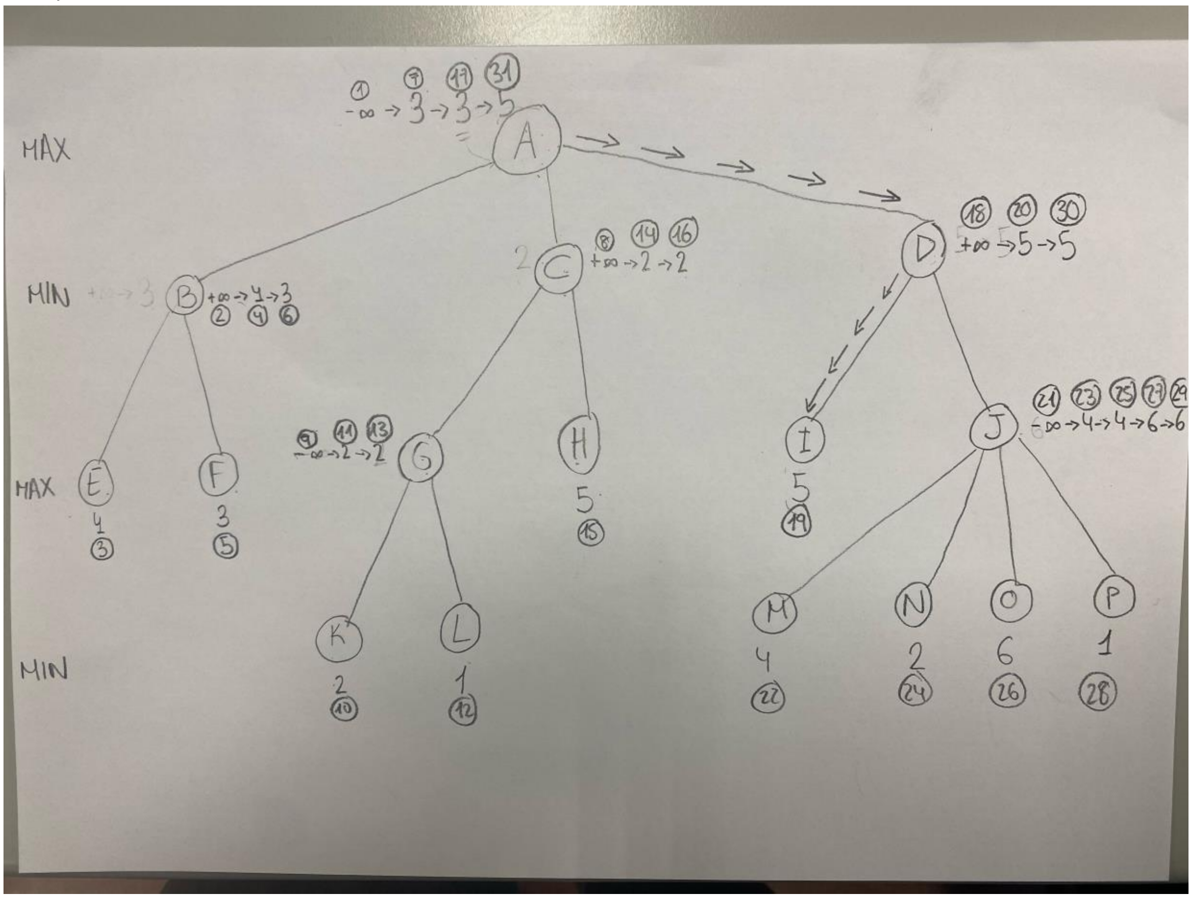
These two last screenshots show the extra code added to min_value and max_value respectively, the rest of the function stays the same. We can see min_value only modifies beta, while max_value only modifies alpha. In the second if we can see that if alpha >= beta (the minimax_value variable is beta in min_value, and alpha in max_value) we immediately return the current value, which means the rest of the successors will not be analysed. Also, note the change in the prints: the values previously shown are expanded to show (alpha, beta) as well, which will be useful in later tests.

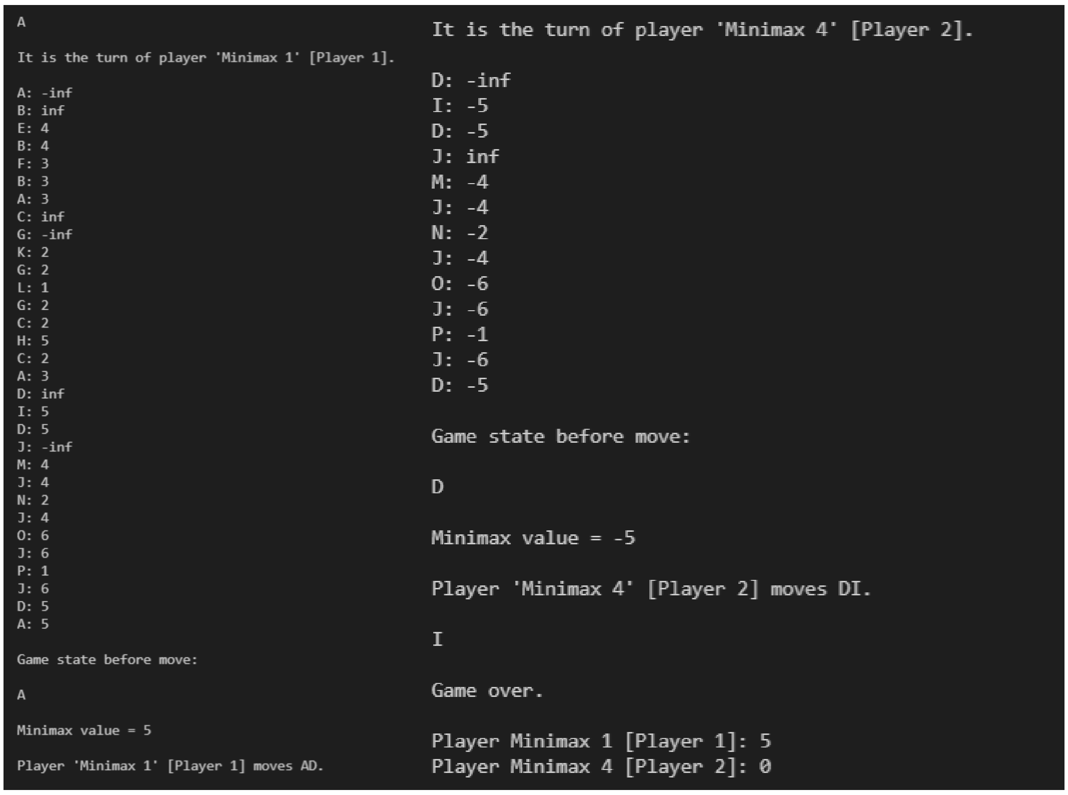
Test 1: demo_simple_game_tree.py

We will now test both algorithms with a simple game tree located in demo_simple_game_tree.py. The tree structure is as follows:



We will now show both our manual solving of this tree with the minimax algorithm, as well as the computer execution.

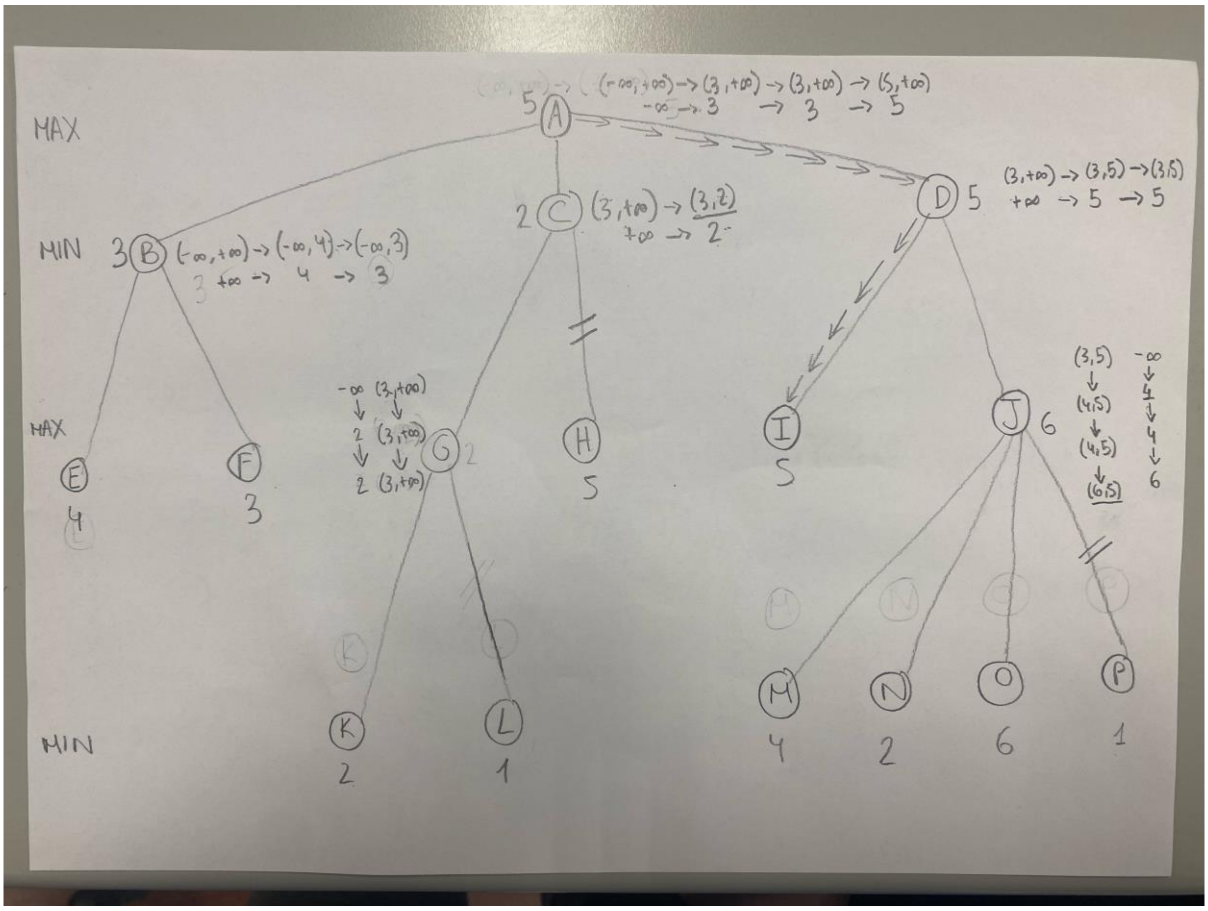




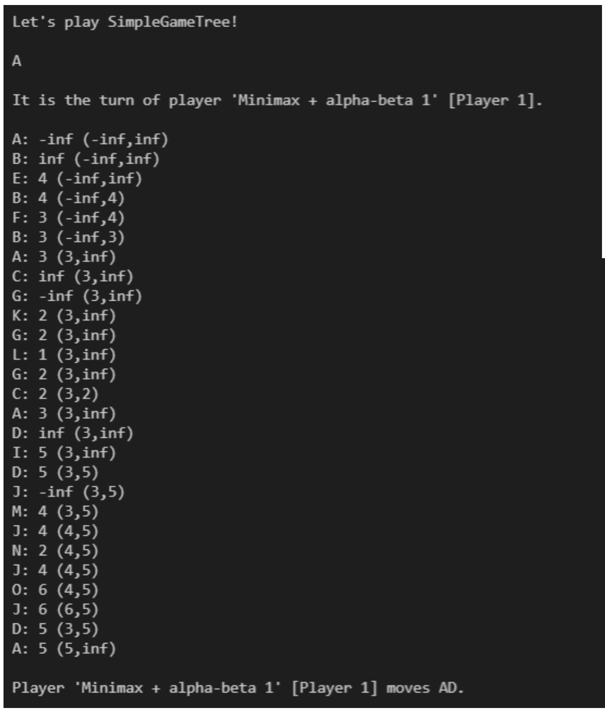
```
A

It is the turn of player 'Minimax 1' [Player 1].

A: -inf
B: inf
E: 4
B: 4
F: 3
B: 3
A: 3
C: inf
G: -inf
K: 2
G: 2
L: 1
G: 2
C: 2
H: 5
C: 2
A: 3
D: inf
I: 5
D: 5
J: -inf
M: 4
J: 4
N: 2
J: 4
O: 6
J: 6
P: 1
J: 6
D: 5
A: 5

Game state before move:

A

Minimax value = 5

Player 'Minimax 1' [Player 1] moves AD.
```

```
It is the turn of player 'Minimax 4' [Player 2].

D: -inf
I: -5
D: -5
J: inf
M: -4
J: -4
N: -2
J: -4
O: -6
J: -6
P: -1
J: -6
D: -5

Game state before move:

D

Minimax value = -5

Player 'Minimax 4' [Player 2] moves DI.

I

Game over.

Player Minimax 1 [Player 1]: 5
Player Minimax 4 [Player 2]: 0
```

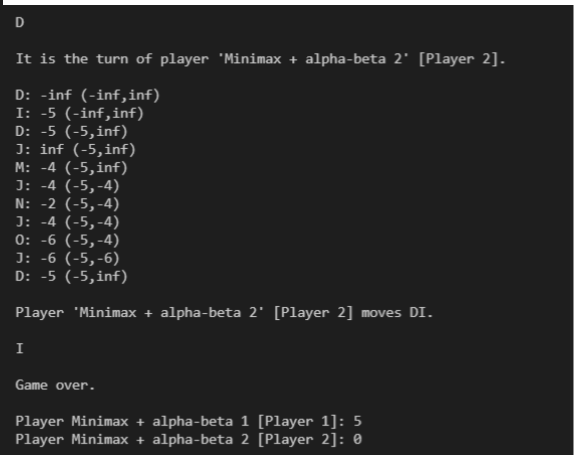
In our manual solving, we show how the minimax value changes through every node, with the circled numbers showing the order in which the algorithm works. It can be seen that it searches the tree depth-first. The path chosen is A>D>I. Next, we will show similar screenshots, only with the alpha-beta pruning.



Two lines drawn in an edge mean the node pruned the rest of successors. In each node that is not a leaf both the minimax value and the (alpha, beta) interval are updated as shown. For clarity purposes, the order of modifications was not shown, but if functions similarly to the one above, with depth-first search.



```
Let's play SimpleGameTree!

A

It is the turn of player 'Minimax + alpha-beta 1' [Player 1].

A: -inf (-inf,inf)
B: inf (-inf,inf)
E: 4 (-inf,inf)
B: 4 (-inf,4)
F: 3 (-inf,4)
B: 3 (-inf,3)
A: 3 (3,inf)
C: inf (3,inf)
G: -inf (3,inf)
K: 2 (3,inf)
G: 2 (3,inf)
L: 1 (3,inf)
G: 2 (3,inf)
C: 2 (3,2)
A: 3 (3,inf)
D: inf (3,inf)
I: 5 (3,inf)
D: 5 (3,5)
J: -inf (3,5)
M: 4 (3,5)
J: 4 (4,5)
N: 2 (4,5)
J: 4 (4,5)
O: 6 (4,5)
J: 6 (6,5)
D: 5 (3,5)
A: 5 (5,inf)

Player 'Minimax + alpha-beta 1' [Player 1] moves AD.
```



```
D

It is the turn of player 'Minimax + alpha-beta 2' [Player 2].

D: -inf (-inf,inf)
I: -5 (-inf,inf)
D: -5 (-5,inf)
J: inf (-5,inf)
M: -4 (-5,inf)
J: -4 (-5,-4)
N: -2 (-5,-4)
J: -4 (-5,-4)
O: -6 (-5,-4)
J: -6 (-5,-6)
D: -5 (-5,inf)

Player 'Minimax + alpha-beta 2' [Player 2] moves DI.

I

Game over.

Player Minimax + alpha-beta 1 [Player 1]: 5
Player Minimax + alpha-beta 2 [Player 2]: 0
```

For every step of the algorithm, we print the minimax value and the (alpha, beta) interval. There are two instances in which alpha>=beta, which coincide to the two times in the manual solving where we pruned the rest of successors. Node H and node P are never visited. The path calculated is A>D>I, as expected.

Though the algorithms' behaviour works as expected, the much higher efficiency of alpha-beta pruning is not noticeable since the game tree is too simple.

Test 2: demo_tournament.py

We recorded several runs of demo_tournament.py. In each tournament we ran all three strategies were equal so the results record a match in which the same strategy matched against itself. Since we ran the tournaments with n=1, we will have a total pool of 6 matches for each iteration.

The iterations in question consisted in: using depths 3 and 4, using minimax with and without alpha-beta pruning, and using the dummy and our own heuristic. The objective of gathering this data is to prove the sheer effectiveness of alpha-beta pruning, especially for trees with a very elevated number of leaves.

For each pool of n=6 matches we will show the average (in seconds)

Dummy heuristic	No pruning	Pruning	Speedup
Depth = 3	35.048 s	6.747 s	5.195
Depth = 4	216.049 s	22.399 s	9.645
Our heuristic	No pruning	Pruning	Speedup
Depth = 3	41.795 s	17.431 s	2.398
Depth = 4	>1200 s	147.584 s	>8.130

We can see all speedups are above 2, and the biggest one almost reaching 10, which proves the much higher efficiency of alpha-beta pruning. We can also notice a much higher speedup for depth 4, which can be explained by the fact that the more depth the game tree has the more nodes there are to prune.

2. Documentation of the design of the heuristic.

Guidelines

We never based our heuristics on specific step by step strategies, rather we always followed a general set of rules and guidelines that are very important in Reversi:

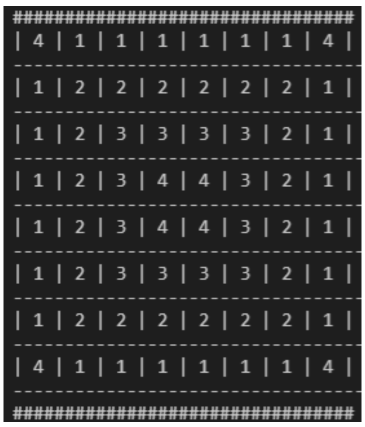
- Priority given to corners and edges: the only 4 coordinates in the board that cannot be invaded are the edges; once you successfully take a corner it will always be yours. Edges are not impossible to invade, but it is more difficult. This means that there is a much higher priority to having pieces in corners and edges as opposed to pieces in the centre of the board (as an example we will go over later, having one corner piece might be equally valuable as having eight non-corner pieces).
- Priority given to building clusters of pieces: while certain places in the board are more valuable than others, there is power in numbers in Reversi; having built a cluster of pieces makes it more difficult for the opponent to attack you. This general principle has also been taken into account while building our heuristics.
- Mobility: the last principle we took into account and by far the most important in terms of priority; mobility references the number of possible moves a player disposes of at a certain

state of the game. It is vital that a good strategy aims to reduce as much as possible the mobility of the opponent while improving your own. It may even be beneficial to sacrifice a corner in the short term to ensure mobility, or reduction of mobility of the opponent, in the long term

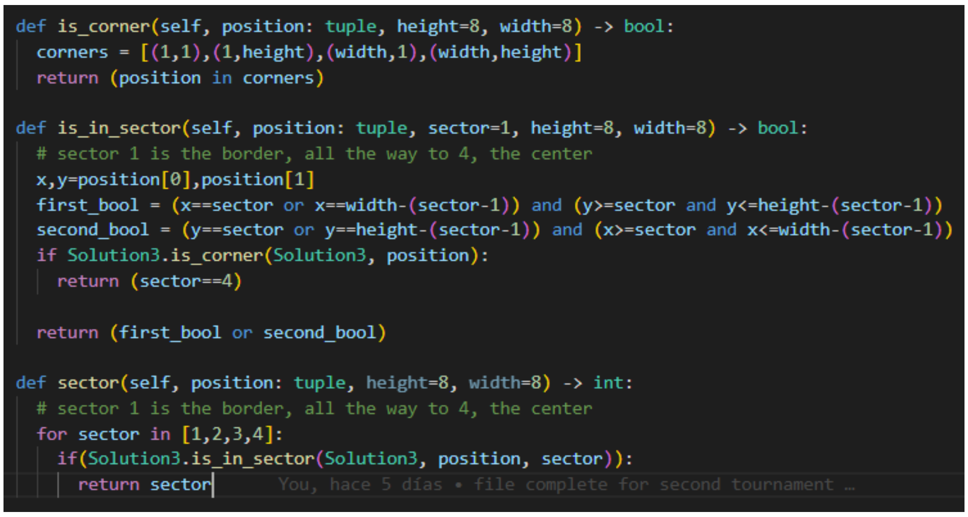
In the process of designing our heuristics, we start by including the first two guidelines. We later improve our own heuristic by introducing the third guideline.

First heuristic: four sectors

The first heuristic we devised takes into account the first two guidelines described above. Its premise is simple enough: divide the Reversi board into four sectors, to which it will be assigned a specific value depending on the spaces' priority. We went through several board configurations until we settled on this one:

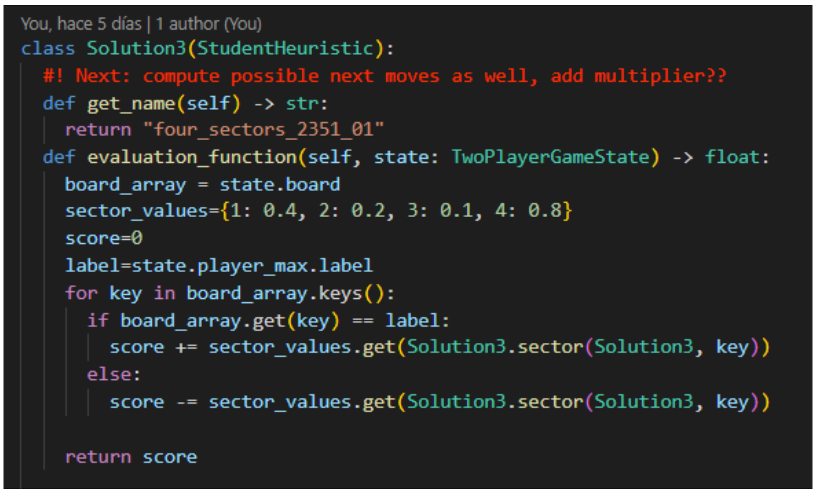

```
#################################
| 4 | 1 | 1 | 1 | 1 | 1 | 1 | 4 |
---------------------------------
| 1 | 2 | 2 | 2 | 2 | 2 | 2 | 1 |
---------------------------------
| 1 | 2 | 3 | 3 | 3 | 3 | 2 | 1 |
---------------------------------
| 1 | 2 | 3 | 4 | 4 | 3 | 2 | 1 |
---------------------------------
| 1 | 2 | 3 | 4 | 4 | 3 | 2 | 1 |
---------------------------------
| 1 | 2 | 3 | 3 | 3 | 3 | 2 | 1 |
---------------------------------
| 1 | 2 | 2 | 2 | 2 | 2 | 2 | 1 |
---------------------------------
| 4 | 1 | 1 | 1 | 1 | 1 | 1 | 4 |
---------------------------------
#################################
```

Sector 4 has the highest priority, followed by Sector 1, Sector 2, and finally Sector 3. Notice Sector 4 is composed of the corners and the 4 centre coordinates: we prioritise both the corners and the central board space, so as to build a cluster. The rest of sectors are higher priority the further they are from the centre, which also prioritises edges. It is evident this configuration conforms to the two first guidelines.


```python
def is_corner(self, position: tuple, height=8, width=8) -> bool:
  corners = [(1,1),(1,height),(width,1),(width,height)]
  return (position in corners)

def is_in_sector(self, position: tuple, sector=1, height=8, width=8) -> bool:
  # sector 1 is the border, all the way to 4, the center
  x,y=position[0],position[1]
  first_bool = (x==sector or x==width-(sector-1)) and (y>=sector and y<=height-(sector-1))
  second_bool = (y==sector or y==height-(sector-1)) and (x>=sector and x<=width-(sector-1))
  if Solution3.is_corner(Solution3, position):
    return (sector==4)

  return (first_bool or second_bool)

def sector(self, position: tuple, height=8, width=8) -> int:
  # sector 1 is the border, all the way to 4, the center
  for sector in [1,2,3,4]:
    if(Solution3.is_in_sector(Solution3, position, sector)):
      return sector        You, hace 5 días • file complete for second tournament …
```

This is the code that sorts a certain coordinate into one of the four sectors.

What does our heuristic actually compute? It concentrates on the board spaces that are taken by pieces in a certain state: if they belong to player max, they are added to the score, and subtracted if they belong to player min.

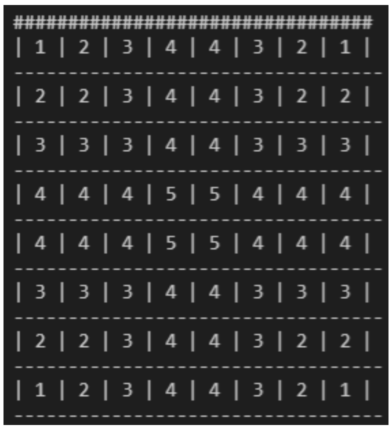


```
You, hace 5 días | 1 author (You)
class Solution3(StudentHeuristic):
    #! Next: compute possible next moves as well, add multiplier??
    def get_name(self) -> str:
        return "four_sectors_2351_01"
    def evaluation_function(self, state: TwoPlayerGameState) -> float:
        board_array = state.board
        sector_values={1: 0.4, 2: 0.2, 3: 0.1, 4: 0.8}
        score=0
        label=state.player_max.label
        for key in board_array.keys():
            if board_array.get(key) == label:
                score += sector_values.get(Solution3.sector(Solution3, key))
            else:
                score -= sector_values.get(Solution3.sector(Solution3, key))

        return score
```

We used a dictionary to store the value associated to each sector. For every piece, we would compute the score based on the sector that it was. The dictionary shown in the screenshot shows the values we have been working with on the tournaments.

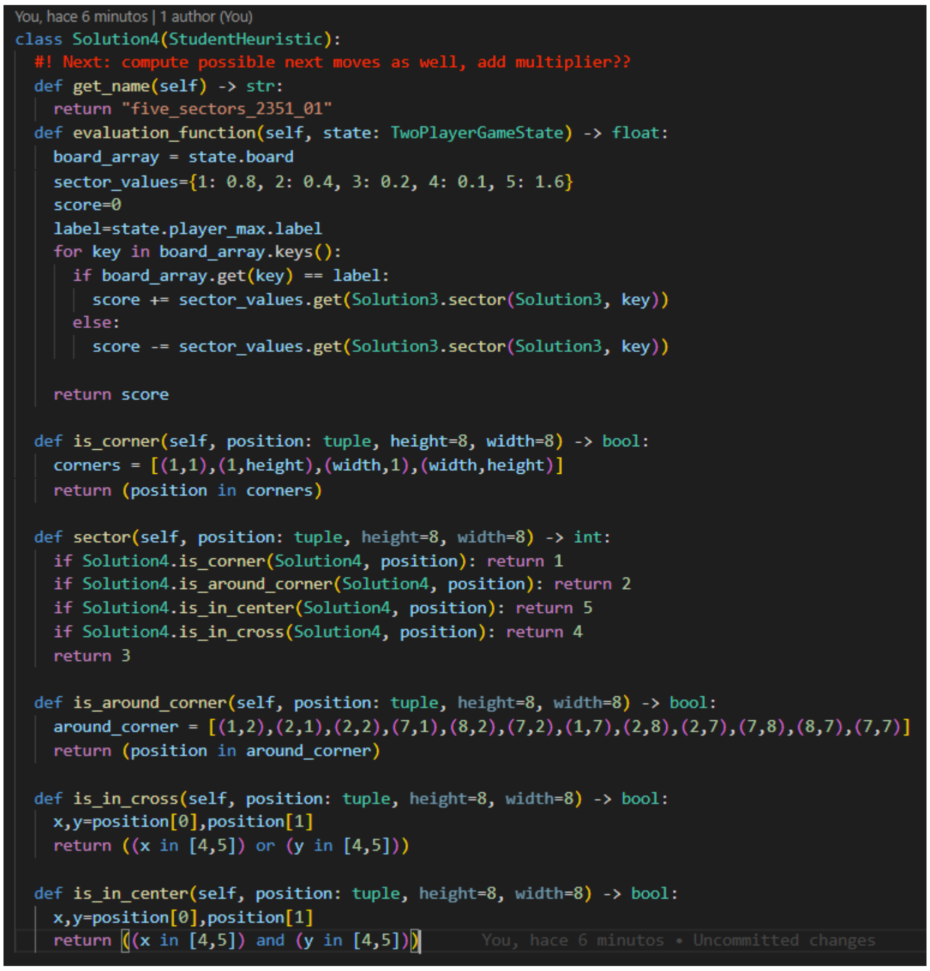
Second heuristic: five sectors

A different board configuration we came up with was as follows:



```
#################################
| 1 | 2 | 3 | 4 | 4 | 3 | 2 | 1 |
---------------------------------
| 2 | 2 | 3 | 4 | 4 | 3 | 2 | 2 |
---------------------------------
| 3 | 3 | 3 | 4 | 4 | 3 | 3 | 3 |
---------------------------------
| 4 | 4 | 4 | 5 | 5 | 4 | 4 | 4 |
---------------------------------
| 4 | 4 | 4 | 5 | 5 | 4 | 4 | 4 |
---------------------------------
| 3 | 3 | 3 | 4 | 4 | 3 | 3 | 3 |
---------------------------------
| 2 | 2 | 3 | 4 | 4 | 3 | 2 | 2 |
---------------------------------
| 1 | 2 | 3 | 4 | 4 | 3 | 2 | 1 |
---------------------------------
```

The class that generates it is at follows:


```
You, hace 6 minutos | 1 author (You)
class Solution4(StudentHeuristic):
  #! Next: compute possible next moves as well, add multiplier??
  def get_name(self) -> str:
    return "five_sectors_2351_01"
  def evaluation_function(self, state: TwoPlayerGameState) -> float:
    board_array = state.board
    sector_values={1: 0.8, 2: 0.4, 3: 0.2, 4: 0.1, 5: 1.6}
    score=0
    label=state.player_max.label
    for key in board_array.keys():
      if board_array.get(key) == label:
        score += sector_values.get(Solution3.sector(Solution3, key))
      else:
        score -= sector_values.get(Solution3.sector(Solution3, key))

    return score

  def is_corner(self, position: tuple, height=8, width=8) -> bool:
    corners = [(1,1),(1,height),(width,1),(width,height)]
    return (position in corners)

  def sector(self, position: tuple, height=8, width=8) -> int:
    if Solution4.is_corner(Solution4, position): return 1
    if Solution4.is_around_corner(Solution4, position): return 2
    if Solution4.is_in_center(Solution4, position): return 5
    if Solution4.is_in_cross(Solution4, position): return 4
    return 3

  def is_around_corner(self, position: tuple, height=8, width=8) -> bool:
    around_corner = [(1,2),(2,1),(2,2),(7,1),(8,2),(7,2),(1,7),(2,8),(2,7),(7,8),(8,7),(7,7)]
    return (position in around_corner)

  def is_in_cross(self, position: tuple, height=8, width=8) -> bool:
    x,y=position[0],position[1]
    return ((x in [4,5]) or (y in [4,5]))

  def is_in_center(self, position: tuple, height=8, width=8) -> bool:
    x,y=position[0],position[1]
    return ((x in [4,5]) and (y in [4,5]))        You, hace 6 minutos • Uncommitted changes
```

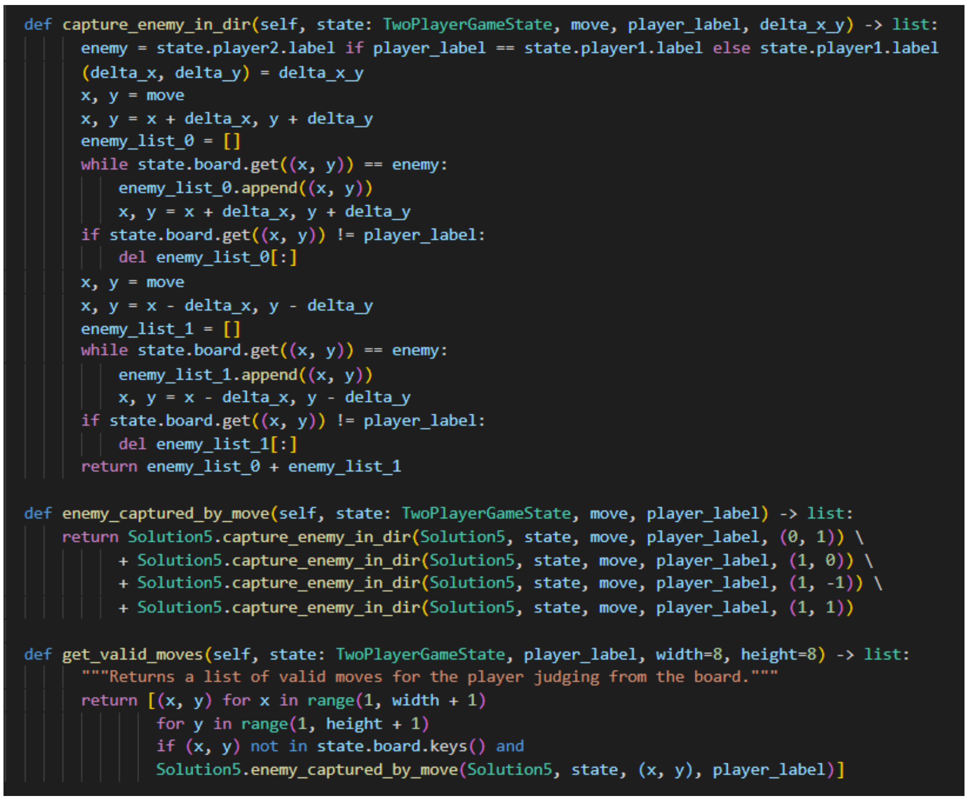
The five last functions generate the five sectors, and apart from that the code structure is identical to four_sectors: our new dictionary computes five sectors.

This heuristic was not as successful as the one above, this might be because the board distribution does not follow the prioritise corners and edges guideline (the edges are spread out in the different sectors). However, it will be part of our final submission.

Final heuristic: four sectors + successors

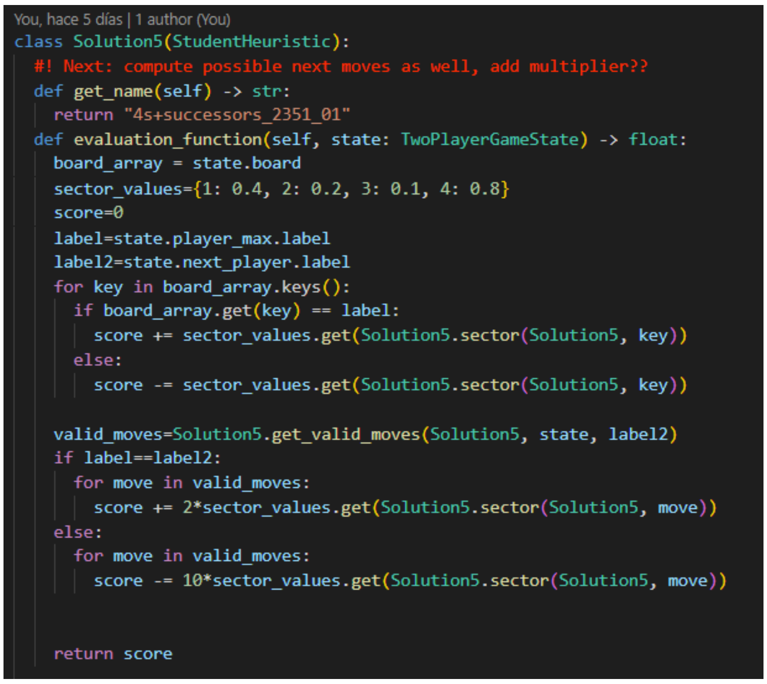
As the name implies, our definitive heuristic follows a similar distribution as the other two. In fact, the board distribution is identical to our four sectors heuristic (described above). This last one, though, introduces the vital third guideline of mobility. The premise is, again, simple enough: we calculate the list of next moves the next player has available, and we compute those spaces along with the pieces. We take into account mobility, the list of “potential spaces” to reach. This way, a state space with high mobility for player max, or low mobility for player min, is immediately more attractive to our algorithm.

We calculate the list of possible moves with the following functions:



```python
def capture_enemy_in_dir(self, state: TwoPlayerGameState, move, player_label, delta_x_y) -> list:
        enemy = state.player2.label if player_label == 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)) != player_label:
            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)) != player_label:
            del enemy_list_1[:]
        return enemy_list_0 + enemy_list_1

def enemy_captured_by_move(self, state: TwoPlayerGameState, move, player_label) -> list:
    return Solution5.capture_enemy_in_dir(Solution5, state, move, player_label, (0, 1)) \
            + Solution5.capture_enemy_in_dir(Solution5, state, move, player_label, (1, 0)) \
            + Solution5.capture_enemy_in_dir(Solution5, state, move, player_label, (1, -1)) \
            + Solution5.capture_enemy_in_dir(Solution5, state, move, player_label, (1, 1))

def get_valid_moves(self, state: TwoPlayerGameState, player_label, width=8, height=8) -> list:
        """Returns a list of valid moves for the player judging from the board."""
        return [(x, y) for x in range(1, width + 1)
                for y in range(1, height + 1)
                if (x, y) not in state.board.keys() and
                Solution5.enemy_captured_by_move(Solution5, state, (x, y), player_label)]
```

These are extracted from reversi.py, from the code we were given, and repurposed slightly so they fit into the class structure.



```python
You, hace 5 días | 1 author (You)
class Solution5(StudentHeuristic):
  #! Next: compute possible next moves as well, add multiplier??
  def get_name(self) -> str:
    return "4s+successors_2351_01"
  def evaluation_function(self, state: TwoPlayerGameState) -> float:
    board_array = state.board
    sector_values={1: 0.4, 2: 0.2, 3: 0.1, 4: 0.8}
    score=0
    label=state.player_max.label
    label2=state.next_player.label
    for key in board_array.keys():
      if board_array.get(key) == label:
        score += sector_values.get(Solution5.sector(Solution5, key))
      else:
        score -= sector_values.get(Solution5.sector(Solution5, key))

    valid_moves=Solution5.get_valid_moves(Solution5, state, label2)
    if label==label2:
      for move in valid_moves:
        score += 2*sector_values.get(Solution5.sector(Solution5, move))
    else:
      for move in valid_moves:
        score -= 10*sector_values.get(Solution5.sector(Solution5, move))


    return score
```

This is the heuristic, almost identical to four_sectors, except for the last section, in which we obtain the list of possible moves and compute them into the score in a similar way (only with added multipliers, so they weigh more in the final score).

With this final heuristic we consider we have successfully taken into account all three guidelines.

Testing and conclusions

Throughout the making of these heuristics, we religiously performed tournaments in demo_tournament.py, first against a dummy and a random heuristic, then eventually our heuristics against each other. It is in the testing where the advantage of our code really is noticeable: it is extremely malleable to change, and easy to do so, while also completely changing the performance of the heuristic. It is as easy as changing the scores in the dictionaries associated to each sector. Through trial and error, we ended up with these three heuristics.

This means that the advantage of our heuristics is not just the ideas and guidelines, but how malleable the code structure really is, so trying out many possible configurations in little time is easy.