Alex Shung
UIN: 653384880
UNET: ashung2
Credits: 3

Assignment 2: Flow-free + Breakthrough

Part 1: Free-flow

Implementation

The game was implemented by reading the input and converting it into a 2d array. The variables were written as their own object, as was the State which represented the assignment. Variables will be discussed in more details further on. A state held all the assigned variables and unassigned variables in a priority queue, as well as a variable map which allowed quick access to variables.

A depth first search was used in both approaches. The method began by checking if the state is completed, which was defined as the number of unassigned variables being 0, and that every position in the state was assigned. This is effectively the goal state of the assignment / state. The next variable in the unassigned priority queue for the state was popped, then for every color in the domain, which will be discussed more in depth later, the variable was assigned to that color if it did not violate any constraints, which will be discussed later. Then the function was recursively called until either it completed the state, and the returned state was the completed state, or it was not possible and nothing was returned. If nothing was returned, then un-assign that variable and try a different color.

Variables

Each position in the grid represented a variable. The variable had a position, the values that it could have, the picked value if it was assigned, and a heuristic that was used for the smart and smarter methods to figure out where to place it in the priority queue of unassigned variables. The heuristic was set to the number of possible values if it was the smart / smarter method. Otherwise it was set to a constant for the dumb method.

Domains

The domain was a set of all colors that appeared in the original board. It is figured out during the initial board state creation, where every time it finds a character that isn't '_', it adds it to the domain. These domains represent the possible assignment colors of each grid. Each variable also keeps track of its own domain, which is the list of possible values it can take.

Constraints

I used strict constraints. For a given variable assignment, I considered all the surrounding points. For each of those points, I identified how many assigned neighbors it had. An assigned neighbor was a point that existed in the state.assignedVariables, in other words it had a picked value / was already assigned OR it was outside of the board. If the number of neighbors that had the same color as this one exceeded 2, it was not a valid assignment as we

invalidated snaking. If the number of assigned neighbors was 3, it was a valid assignment if the number of same colored neighbors was greater than 0 or if it was an end. If the number of assigned neighbors was 4, then it was a valid assignment if it there were 2 of the same color if it wasn't an end, otherwise the number of same color neighbors must have been 1. If the number of assigned neighbors did not match any of the cases above, it was a valid assignment. For a more detailed implementation, please refer to canAssign and isConsistent methods in Part1.py. Some pseudocode is provided below:

```
isConsistent(state, point, color):
        neighbors = [up, down, left, right]
        numSame = 0
        numAssigned = 0

        for n in neighbors:
                if is off grid:
                        numAssigned += 1
                else if n is assigned:
                        numAssigned += 1
                        if n.color is the same or n.possibleValues has color and length of
possibleValues = 0:
                                numSame += 1

        if numSame > 2:
                return False

        # Early termination case
        elif numAssigned == 3:
                return numSame > 0 if point not in ends else True
        elif numAssigned == 4:
                return numSame == 2 if point not in ends else numSame == 1
        else:
                return True

def canAssign(state, var, color, ends, puzzleLength, puzzleWidth, domain):
        (x,y) = var.point
        points = [var.point] + makeNextPoints(x,y)
        for point in points:
                spaceColor = state.assignedVars[point].pickedValue if point in state.assignedVars
else color
                if point in state.assignedVars or point == var.point:
                        pointConsistent = isConsistent(state, point, spaceColor, ends,
puzzleLength, puzzleWidth, domain)
                        if not pointConsistent:
                                # print(f"{point} was determiend to be inconsistent")
                                return False
```

```
return True
```

Dumb Method

This method does the implementation as described above. There is no ordering of which variable is selected as the heuristic used for each variable is the same, so the priority queue randomly selects one based on how it happened to heapify exactly similar objects. Thus the selection of the piece is random. Similarly, the color domain is held as a set, so which one is chosen at any point is up to how the hashing function worked. If it cannot find a valid assignment it will return none. Backtracking was implemented by recursively calling the csp method as described in the pseudocode below. Effectively it attempted to make an assignment, and if that assignment ended up failing, trying a different assignment. I believe this is valid as the spec only states that it should have "random variable and value ordering / no forward checking", which seems to imply that backtracking should be fine.

For more information please refer to cspRandom in Part1.py. Some pseudocode is provided below:

```
cspRandom(state, domain):
    if state is goal state:
        return state

    var = state.unassignedVars.pop
    for color in domain:
        assign var to color in state
        if can assign var to color based on above constraints:
            result = cspRandom(state, domain)
            if there is a result:
                return result
        unassign color from var
    return None
```

Smart Method

Smart method differs from dumb method in that it updates all the unassigned variables after each assignment. It updates the heuristic, which is the number of variables possible for that variable. For example, if the variable has color options {B, R, G}, then its heuristic would be 3 and thus represents the constraint on that variable. The unassigned variables are given possible values based on whether or not assigning the color would cause be inconsistent with the immediate neighbors, as described in the constraints portion. The most constrained variable is selected first since it has the fewest available options in its domain. By updating after every assignment, it also allows for forward checking as if any variable appears to have 0 legal assignments, the smart method knows to short circuit and return to either where a choice was made or to break entirely. This is, in essence, inference as it allows for earlier termination of bad states. Also, instead of traversing all the possible colors in the domain, it would only

traverse the colors present in its own domain. Backtracking was implemented by recursively calling the csp method as described in the pseudocode below. Effectively it attempted to make an assignment, and if that assignment ended up failing, trying a different assignment.

For more information please refer to cspSmart in Part1.py. Some pseudocode is provided below:

```
cspSmart(state, domain):
    if state is complete:
        return state

    # This updates all the variables with the available values
    # Return true if all unassigned variables have at least one availible colors
    # Updates by trying to assign as described above
    if updateAllValues():
        var = state.unassignedVars.pop

        for color in var.domain:
            assign var to color in state
            result = cspSmart(state, domain)
            if there is a result:
                return result
            un-assign var from color in state
    return None
```

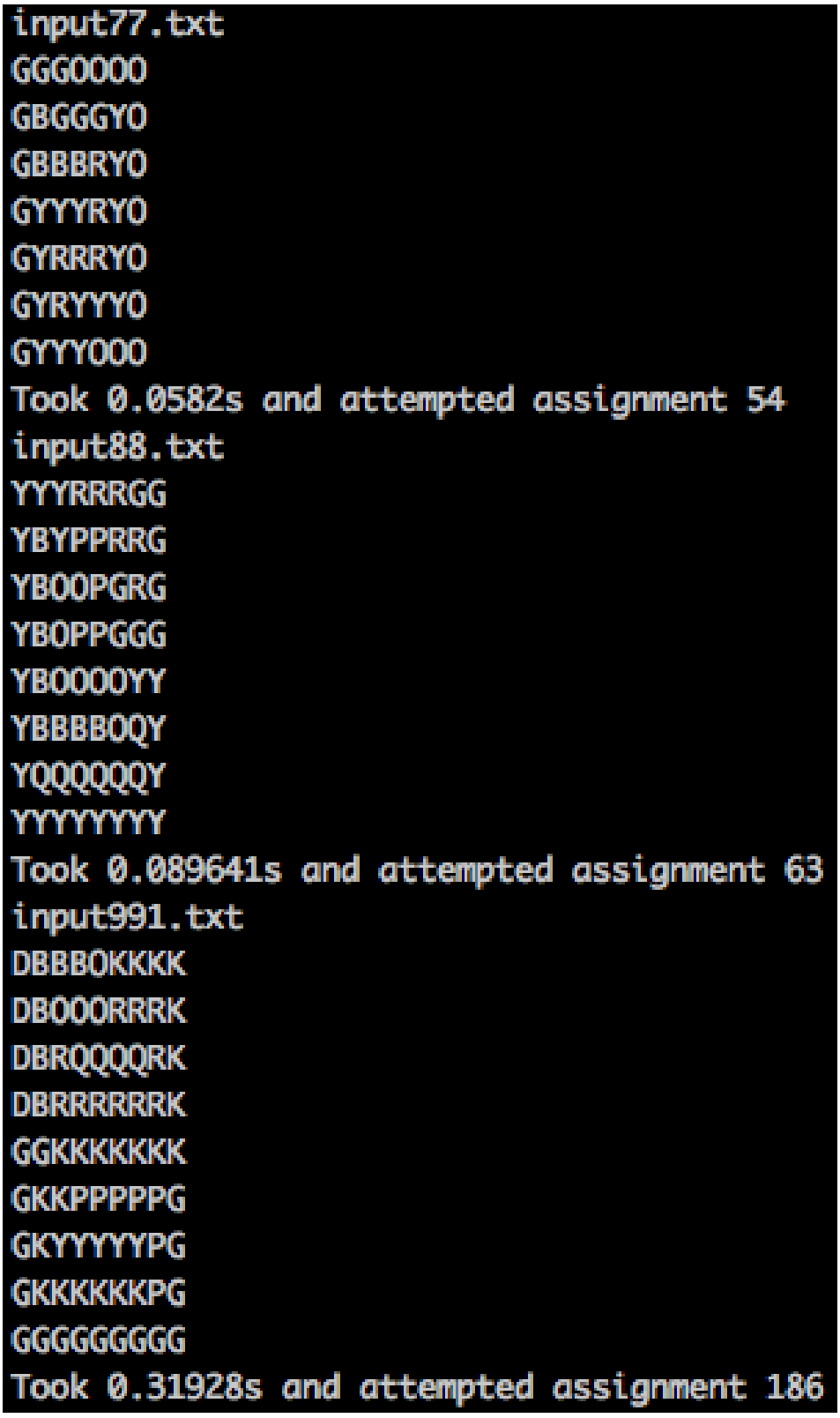
Suggested Smarter Method (Extra Credit)

An even smarter method would be instead of updating the entire board, mimic arc consistency by updating only the neighbors, and then continuing to update only the neighbors which have had their number of available colors change. This would be arc consistency as each piece depends on its neighbors for their available colors. Therefore, the variable's domain would only change when the adjacent variable changes. Thus, doing the above would introduce arc consistency, something similar to what is implemented in updateSingleValue. Update single value currently has bugs, however it was along the right idea of making a recursive update until nothing has been changed.

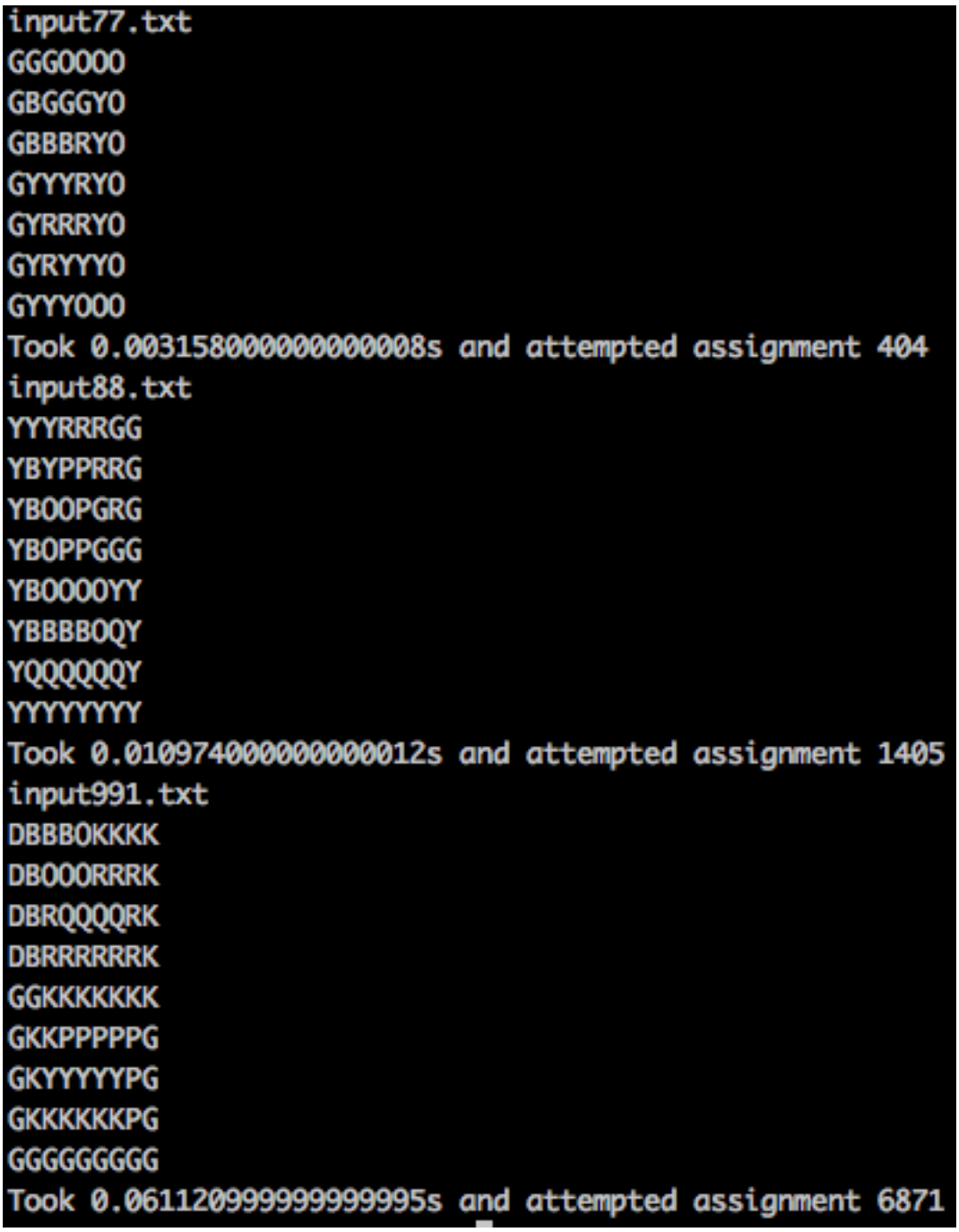
Results

Smaller

Smart



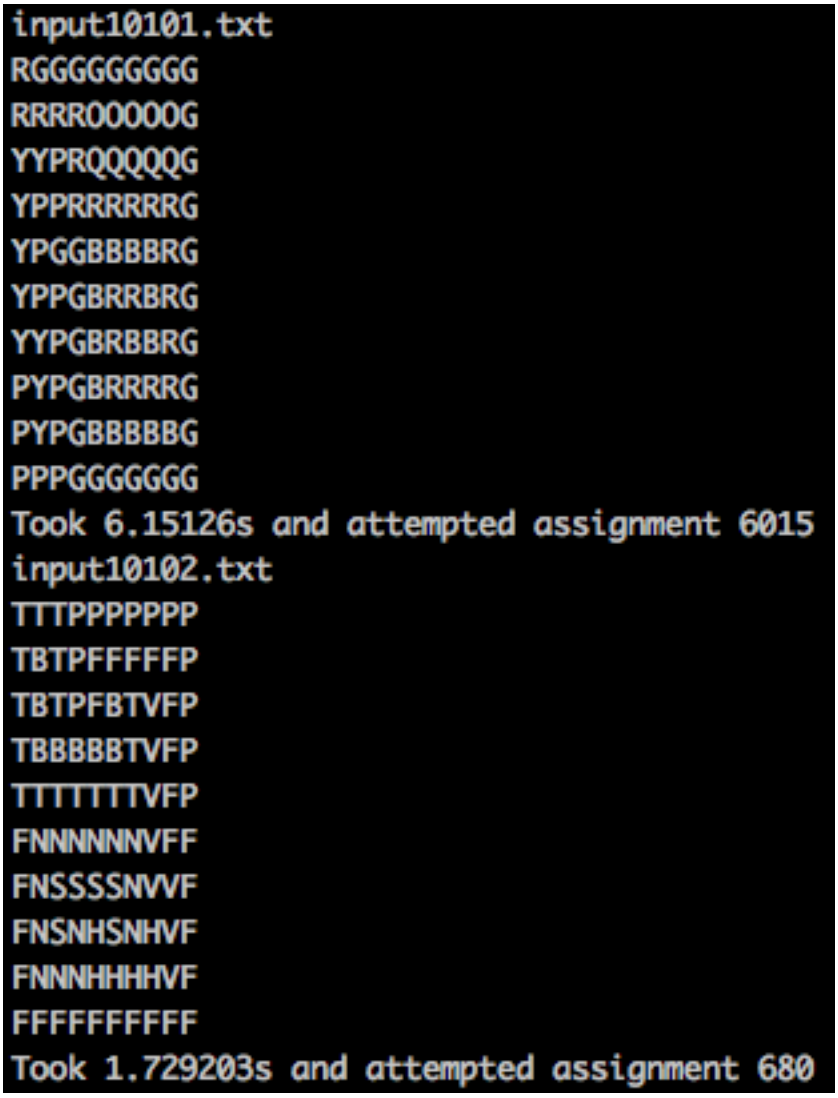
Dumb



The attempted assignment was an order of magnitude lower for the smart assignment. The overall time for the dumb assigner was actually faster though. This gap most likely changes as the size of the board changes, however there is the possibility that it might not, given how the update all values function ends up working. Also since backtracking was implemented using dfs as the base algorithm (dumb), it most likely performed fairly decently.

Bigger (Extra Credit)

Smart



Part 2: Breakthrough

Implementation

Game

The game was implemented as a Player (Player 1) vs another Player (Player 2) on a Board. The Player had the depth limit, the type (Minimax vs AlphaBeta), and the heuristic function that it would run. The Player's name was also defined as the type + the heuristic name. The Board was two lists and a dictionary. The two lists represented the black and white pieces on the board, which in turn represented Player 2 and Player 1, respectively. The dictionary was a map from the position to the piece that currently occupied it. A Piece was defined as the color and coordinate that it occupied.

The game was played until the board was in a winning state. The board was checked after each move to see if that move caused a winning state. During each player's turn, they would evaluate the board based on their heuristic. The player would evaluate each move for every piece that they had, and a depth first search was used to search to the depth. At the depth, or if it was both alphabeta and the board was won (as an early termination step), the board would be evaluated as defined by the heuristic. Heuristics Offensive 1 and Defensive 1 were implemented based as described, so 2 * numPieces + r.random() and 2 * (30 - numPiecesLeft) + r.random() respectively. This heuristic was then returned, and evaluated if it was "better" than the current best move for that piece. The better evaluation was done with the logic of if there is no bestPieceScore defined, this piece is true, otherwise this score is better if it's greater than the bestPieceScore if this is a max level, as defined by if the color of the person who's move was being evaluated is the same as the player who is thinking. Otherwise the board score is better if the score is less than the current bestPiece score. The pieces were then moved back to their original positions, and any taken pieces were placed back on the board. The bestPieceScore is then compared to the current bestScore, and follows similar logic to above to determine if that bestPieceScore was better. If it was, the bestScore is then set to that bestPieceScore. For Alphabeta, I also passed in the bestPieceScore as I continued down the depth first search. I decided if I could short circuit, or basically prune the rest away, by False if the parentScore was undefined (effectively the first piece being evaluated) otherwise if the bestPieceScore was > parentScore if it was a max, otherwise bestPieceScore < parentScore. The best move associated with the best score and the best score were then returned.

For visual representation, a method to print the Board object was created, which created an 8x8 array of '_'. Then it looked at all the pieces in the Board, and changed the '_' to the color of the piece, either "B" or "W". This is what is represented in the results.

Offensive Heuristic 2

Offensive heuristic 1 did not consider position, depth (IE how many moves it took to reach that score), or if there was a winning move at all. My Offensive 2 focused on dealing with these issues. The position element was improved upon by adding the farthest distance any piece had moved multiplied by 2, and added the piece's current x position. The addition of the x-position allowed for a more natural progression towards the right side of the board, and since the goal was to defeat defensive 1, stacking and pushing along one side proved to be a stronger strategy. The baseline heuristic was also used, however multiplied by 3 instead of 2 as the weight of taking a piece was prioritized over moving forward, and the adjusted weight accounts for that. Next the depth was also added as a heuristic, since if you can achieve the same heuristic score board in less moves, the smaller the number of moves is the better choice. There were cases where the heuristic would consider not taking / making a winning move. This heuristic then compensates for that by looking for winning board states, and assigning a max score with a depth adjustment (so that if it can win in fewer moves it will do so). If the board is in a losing state, it will also assign a maximum negative score with a depth adjustment, so it will try it's hardest to prevent a victory state.

Finally, it also considered the opponent's score. It evaluated the board using offensiveHeuristic1, but for the other player. This number was subtracted from the score, which

would allow this player to consider what the opponent was doing and actively encourage them to make a worse move.

Defensive Heuristic 2

Defensive heuristic 1 suffered from the same problems as offensive 1. To compensate, the same idea of looking for a winning / losing board was implemented with similar max / min scores. The depth adjustments were also made to encourage more optimal play. The position was implemented slightly differently, as instead of picking the farthest piece it would pick the closest piece. This caused the defensive pieces to move more like a wall, having minimum space between each piece. The base heuristic was also the same as the defensive 1, and was added to the score.

Finally, the opponent's moves were also analyzed using defensive 1, and the consequent score subtracted from the score.

General Trends

Defensive builds tended to only win if they were pressured into winning, or a random move allowed them to win. Often when given the chance to take a piece, the defensive heuristic will depend on randomness to decide whether or not to take it since the heaviest weighted factor is whether or not it will get eaten. This can also lead to weird move decisions, where since even if it ate the next piece it would get eaten regardless, it won't eat that piece and instead choose a random piece to move. Because of the randomness of the selection, the defensive builds ended up more likely moving all pieces closer to equally, until one was threatened and could be saved.

Offensive builds tended to set themselves up in poor positions. Since they only consider the chance for taking another piece, they will sometimes put themselves into positions where they will be taken since they can't find a position where they can manage to take a piece. From the offensive perspective, moving forward and letting yourself be taken vs. moving a different random piece are equivalent.

Forks were also an interesting event. They seemed to be caused when taking a piece did not change the heuristic value, for example because they would get taken back instantly as defensive, so either way the piece is lost. Thus, they might move a different piece forward into an attacking position, and deciding which piece to take in that case depended on whether they would be taken in turn. As we've divided the heuristics into defensive and offensive, often it would not matter that the piece was taken in return as the heuristic wouldn't care about one of the takes.

A general trend I noticed was that both of my heuristics tended to build towers or diagonals. The moves were forcing them to move in such a way that they would line up along a single y axis sometimes, or move to defend a piece diagonally. It also seemed like for the position based component, a piece would seemingly slip forward until a random move caused an opposing piece to move out of the way, thus allowing that piece to slip past and win. This seemed to happen more when heuristics which had positional values played against heuristics without positional values, and makes sense because the piece that is moving out of the way isn't considering the value of the opponent being able to move forward.

Lastly, as mentioned above, the baseline heuristics don't consider moves that might instantly win the game. A prime example is if a piece is at a single move away from the other end of the board, it will only move forward randomly if there are no other pieces to either defend (defensive) or attack (offensive). This is because their heuristic function doesn't look for a winning state, nor does it consider position at all. Thus, the game is effectively being played to eliminate (offensive) or outlast (defensive) your opponent, and often just coincidentally wins the game.

Results

Minimax has depth of 3, Alphabeta has depth of 5

```
Offensive2 vs Defensive1
Final Board:
__W_B___
B__B_BBB
________
B__W__W_
B_W___W_
W_W___W_
W__W__W_
W____W_W

winning player: AlphaBeta - offensiveHeuristic2
W = AlphaBeta - offensiveHeuristic2: # of game tree nodes -> 17124, avg nodes per move -> 544, avg amount of time -> 0.288, # of pieces
 captured -> 8
B = AlphaBeta - defensiveHeuristic1: # of game tree nodes -> 16845, avg nodes per move -> 552, avg amount of time -> 0.23, # of pieces
captured -> 2
total moves: 61
```

```
AlphaBeta vs Minimax
Final Board:
________
________
____W___
________
__W____W
W___W___
________
WW_W_WW_

winning player: AlphaBeta - offensiveHeuristic1
W = AlphaBeta - offensiveHeuristic1: # of game tree nodes -> 4462976, avg nodes per move -> 119013, avg amount of time -> 46.954, # of
pieces captured -> 16
B = Minimax - offensiveHeuristic1: # of game tree nodes -> 15153, avg nodes per move -> 415, avg amount of time -> 0.091, # of pieces c
aptured -> 6
total moves: 73
```

```
Offensive2 vs Offensive1
Final Board:
_WBBBBBB
_B_B___B
_____B__
________
____W__W
________
W_W_W__W
WWW_WWWW

winning player: AlphaBeta - offensiveHeuristic2
W = AlphaBeta - offensiveHeuristic2: # of game tree nodes -> 9839, avg nodes per move -> 562, avg amount of time -> 0.319, # of pieces
captured -> 6
B = AlphaBeta - offensiveHeuristic1: # of game tree nodes -> 8707, avg nodes per move -> 528, avg amount of time -> 0.212, # of pieces
captured -> 2
total moves: 33
```

```
Defensive2 vs Offensive1
Final Board:
___B___W
________
BB_B_B__
________
________
WW_WWW_W
WWWWW_WW
________

winning player: AlphaBeta - defensiveHeuristic2
W = AlphaBeta - defensiveHeuristic2: # of game tree nodes -> 15786, avg nodes per move -> 518, avg amount of time -> 0.273, # of pieces
 captured -> 11
B = AlphaBeta - offensiveHeuristic1: # of game tree nodes -> 15003, avg nodes per move -> 509, avg amount of time -> 0.177, # of pieces
 captured -> 2
total moves: 59
```

```
Offensive2 vs Defensive2
Final Board:
__W_____
________
_B_____B
_____B__
_BBB___B
________
__W___W_
_______W

winning player: AlphaBeta - offensiveHeuristic2
W = AlphaBeta - offensiveHeuristic2: # of game tree nodes -> 20072, avg nodes per move -> 432, avg amount of time -> 0.188, # of pieces
 captured -> 9
B = AlphaBeta - defensiveHeuristic2: # of game tree nodes -> 20086, avg nodes per move -> 441, avg amount of time -> 0.197, # of pieces
 captured -> 12
total moves: 91
```

```
Defensive2 vs Defensive1
Final Board:
___W____
______BB
________
_W_BWBW_
WB_W_WBW
W__W__WB
_______W
________

winning player: AlphaBeta - defensiveHeuristic2
W = AlphaBeta - defensiveHeuristic2: # of game tree nodes -> 22910, avg nodes per move -> 515, avg amount of time -> 0.179, # of pieces
 captured -> 9
B = AlphaBeta - defensiveHeuristic1: # of game tree nodes -> 22183, avg nodes per move -> 510, avg amount of time -> 0.132, # of pieces
 captured -> 4
total moves: 87
```

Pseudocode

```
Player 1 = W
Player 2 = B

prevPlayer = Player 2
board = initialBoard
while isNotBoardWon(board):
        curPlayer = swap(prevPlayer)
        nextMove = evalBoard(board, curPlayer, curPlayer.color)
        board.makeMove(nextMove)


evalBoard(board, curPlayer, color):
        isMax True if curPlayer == color

        if depthLimitReached or isBothAlphaAndBoardWon:
                return (curPlayer.heur(board), None)

        bestDepthScore = None
        bestMove = None
        shortCircuit = False

        for every piece in board.getMyPieces(color):
                if isAlpha and shortCircuit == true:
                        break

                allMoves = [up, right, left]
                bestPieceScore = None
                bestPieceMove = None

                for move in allMoves:
                # This is defined: if up, can't have a piece. If diag, either can't have a piece or the
piece is the opposite color
                        if isValid(move, piece, board):
                                (takenPiece, isNextBoardWin) = board.makeMove(move, piece)

                                (curScore, nothing) = evalBoard(board, curPlayer, oppositeColor)

                                isBetter = True if not bestPieceScore else boardScore >
bestPieceScore if isMax else boardScore < bestPieceScore

                                if isBetter:
                                        bestPieceScore = boardScore
                                        bestPieceMove = (move, piece)
```

```
                                board.undoMove(move, piece)

                                isShortCircuit = False if not parentScore else bestPieceScore >
parentScore if isMax else bestPieceScore < parentScore
                                if isAlphaBeta:
                                        if (isNextBoardWon and isMax) or isShortCircuit:
                                                shortCircuit = True
                                                break
                        isBetterPiece = True if not bestScore else False if not bestPieceScore else
bestPieceScore > bestScore if isMax else bestPieceScore < bestScore
                        if isBetterPiece:
                                bestScore = bestPieceScore
                                bestMove = bestPieceMove
        return (bestScore, bestMove)

defensiveHeuristic2(board, color, depth, isBoardWon, isMax):
        if isBoardWon:
                return maxScore - depth if max, else maxScore * -1 + depth
        else:
                numPieces = len(board.black) if color == bp else len(board.white)
                basicHeur = 2 * numPieces
                positionScore = getFarthestPiece(board, color)
                oppColor = wp if color == bp else bp
                opponentScore = defensiveHeuristic1(board, oppColor)
                score = basicHeur + r.random() - depth + positionScore - opponentScore

def offensiveHeuristic2(board, color, depth, isBoardWon, isMax):
        score = None
        if isBoardWon:
                score = maxScore - depth
                if isMax:
                        score = score * -1
                score = -1 * maxScore + depth if isMax else maxScore - depth
        else:
                numPiecesLeft = len(board.black) if color == wp else len(board.white)
                basicHeur = 3 * (30 - numPiecesLeft)
                positionScore = getFarthestPiece(board, color, True)

                oppColor = wp if color == bp else bp
                opponentScore = offensiveHeuristic1(board, oppColor)
                score = basicHeur + r.random() - depth + positionScore * 2 - opponentScore
        return score
```

```
# This returns the farthest distance a piece has traveled + how far to the right it is OR the shortest distance a piece has traveled
def getFarthestPiece(board, color, isOffensive = False):
	isBlack = True if color == bp else False
	myPieces = board.black if color == bp else board.white
	farthest = None
	for piece in myPieces:
		curPos = piece.y if not isBlack else 8 - piece.y #* 2 + piece.x
		if isOffensive:
			curPos = curPos * 2 + piece.x
		isBetter = True if not farthest else curPos > farthest if isOffensive else curPos < farthest
		if isBetter:
			farthest = curPos
	return farthest
```