**Breakthrough Game**

The purpose of this report is to describe the development of a breakthrough game. It is played on a square board with eight-by-eight cells, and each player has sixteen pieces. Our objective was to assist the agent in making decisions during the game by developing offensive and defensive strategies. The game was developed using the Python programming language and the Pygame GUI library.

First let’s discuss the given game rules. Initially, the 8x8 board is set up like a chessboard, with 16 workers for each player. It is a two-player game, where the players alternate turns and can only move one piece (of their own) at a time. The worker can only move one square forward or diagonally forward in each turn. Also, workers of the opponent can be captured if they are placed diagonally in front of them. It is not possible to capture an opponent worker if it is directly in front of the player. The game ends when one worker reaches the opponents team's home base which is the last row or when all opponents workers have been captured.

**Algorithm description**

First, we will initialize the board with 16 pieces for each player using a two-dimensional array "startingBoardRepresentation" and we have initialized a variable "status" to track the state of the game. Using this "status" variable we need to continue on looping over the game until one player wins. Until the status indicates no one is won or there is a draw the steps discussed in the below paragraph will be continuously performed.

In this loop, the current state of the game board is first determined by referring to "startingBoardRepresentation". Then, different heuristic strategies are applied to suggest moves for two players: offensive heuristic 1 for player 1, defensive heuristic 1 for player 2, followed by offensive heuristic 2 for player 2, and defensive heuristic 2 for player 1. Each heuristic provides a strategy or set of guidelines to help decide the best move, with offensive heuristics focusing on aggressive play and defensive heuristics on protecting one's position. After these suggestions are generated, the agent will select the preferred move from the options provided and the chosen move is executed, altering the state of the board to reflect the new position of the game pieces. Like this the game is played by and to give alternate chances to each player we are using the "alterturn" function.

**Game Implementation**

To implement this game we had used python, as discussed earlier we had written the logic for the algorithm using python and to display the output we had used the Pygame library. The algorithm will suggest best move for each player and state of the board will be changed according to that the alterturn function will give alternate turns to each user. After making changes to GUI, we need to refresh the window to reflect the changes. Next, we will discuss the heuristic functions used.

The agents in game uses:

**Minmax:** In AI, there are two agents: Maximiser and Minimizer. The two agents play the game differently, with one trying to get the most score or benefit possible, and the other trying to get the least score or benefit possible.

**Alpha-beta pruning:** Alpha-beta pruning is a search algorithm designed to reduce the number of nodes that are evaluated by the minimax algorithm.

**Psuedo Code**

def offensiveHeuristic1(self, turn):

return 2\*(30-self.opponentScore(turn))+random.random()/10

def defensiveHeuristic1(self, turn):

return 2\*self.myScore(turn)+random.random()/10

def offensiveHeuristic2(self, turn):

return 1 \* self.myScore(turn) - 2 \* self.opponentScore(turn)

def defensiveHeuristic2(self, turn):

return 2 \* self.myScore(turn) - 2 \* self.opponentScore(turn)

The above pseudo code shows the implementation of heuristic functions. We had implemented the heuristics as per the requirements, the offensive heuristics focus on capturing the pawns whereas defensive heuristic focus on defensive approach. Next, we will see the implementation of Alpha beta and Minimax move algorithms, these algorithms use the implementation of MINIMAX and Alpha Beta algorithms. The ai\_move\_minimax function Executes a move using the MINIMAX algorithm and ai\_move\_alphabeta function Executes a move using the Alpha-Beta Pruning algorithm. A key difference between the two functions is what algorithm is used (Minimax vs Alpha-Beta Pruning), as well as the depth of search. In general, Alpha-Beta Pruning is more efficient than Minimax because it prunes unnecessary branches in the search tree, allowing for deeper searches in the same amount of time. These functions are capable of making strategic decisions based on the current state of the game.

def ai\_move\_minimax(self, function\_type):

board, blocks, piece = MinimaxAgent(self.BoardRepresentation, self.turn, 3, function\_type).minimax\_decision()

self.BoardRepresentation = board.getMatrix()

if self.turn == 1:

self.total\_blocks\_1 += blocks

self.turn = 2

elif self.turn == 2:

self.total\_blocks\_2 += blocks

self.turn = 1

self.eat\_piece = 16 - piece

if self.isgoalstate():

self.gameStatus = 3

def ai\_move\_alphabeta(self, function\_type):

board, blocks, piece = AlphaBetaAgent(self.BoardRepresentation, self.turn, 4, function\_type).alpha\_beta\_decision()

self.BoardRepresentation = board.getMatrix()

if self.turn == 1:

self.total\_blocks\_1 += blocks

self.turn = 2

elif self.turn == 2:

self.total\_blocks\_2 += blocks

self.turn = 1

self.eat\_piece = 16 - piece

if self.isgoalstate():

self.gameStatus = 3

**Game Play**

Playing game is pretty straight forward, if we click the purple button the pawns will reset. For playing each strategy we need to click each button, they are labelled as Offensive Heuristic 1 vs Offensive Heuristic 1. Here is the game play.

Minimax (Offensive Heuristic 1) vs Alpha-beta (Offensive Heuristic 1)

A screenshot of a computer game

Description automatically generated

Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 1)

A screenshot of a computer game

Description automatically generated

Alpha-beta (Defensive Heuristic 2) vs Alpha-beta (Offensive Heuristic 1)

A computer screen shot of a game

Description automatically generated

Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Offensive Heuristic 1)

A screenshot of a computer game

Description automatically generated

Alpha-beta (Defensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 1)

A screenshot of a computer game

Description automatically generated

Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 2)

A screenshot of a computer game

Description automatically generated

We had attached the output in a game\_output.txt file attached in the zip file submitted.