**SSN COLLEGE OF ENGINEERING (Autonomous)**

**Affiliated to Anna University**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**Team Project**

***"Strategic Gameplay in Chain Reaction: An Adversarial Search Approach"***

Subject : **Foundations of Artificial Intelligence**

Class  : V Sem - III year

Sub. Code : **UCS2504**                                           Acad. Year:  2023-2024 (Odd)

Batch : 2021-2025

Staff Incharge : **Dr. Saritha M** Date: 27-11-2023

**Team members:**

1. **Sham S – 3122 21 5001 099**
2. **Sudharsan K - 3122 21 5001 111**
3. **Vaishal Raj G - 3122 21 5001 119**
4. **Kevin John Daniel - 3122 21 5001 701**

**Problem Statement:**

To develop an artificial intelligence system for the game 'Chain Reaction' to autonomously make strategic moves. The challenge is to implement adversarial search techniques, specifically Minimax and Alpha-Beta Pruning, to enhance the decision-making capabilities of the AI.

**Objectives:**

1. Implement Minimax algorithm for strategic decision-making.
2. Incorporate Alpha-Beta Pruning to optimize search efficiency.
3. Develop an intelligent AI agent for autonomous gameplay.
4. Evaluate and refine the AI system for optimal performance.

**Solution:**

The solution involves implementing a Minimax-based artificial intelligence system with Alpha-Beta Pruning for efficient decision-making in the Chain Reaction game. This approach aims to create an autonomous AI agent capable of strategic gameplay, enhancing the overall user experience. Thorough testing and refinement ensure optimal performance and showcase the effectiveness of adversarial search techniques.

**Software Requirements Specification:**

**Functional Requirements:**

1. AI Decision-Making:

- The system must implement the Minimax algorithm to enable the AI to make strategic decisions during the Chain Reaction game.

2. Alpha-Beta Pruning Integration:

- The system must incorporate Alpha-Beta Pruning to optimize the search process, reducing computational complexity and improving efficiency.

3. Game State Representation:

- The system must accurately represent the current state of the Chain Reaction game board, including the positions and values of orbs.

**Non-functional Requirements:**

Performance:

* Ensure the game responds promptly to user clicks.
* Limit the AI decision-making time to provide a smooth user experience.

User Experience:

* Design an intuitive and visually appealing interface.
* Craft clear and informative game messages for better user understanding.

**ALGORITHMS, DATASETS, AND TEST CASES:**

**Algorithms:**

**Minimax Algorithm:**

* Utilized for AI decision-making to ensure optimal moves.

**Datasets**:

* Game Board State:

Represented by a m x n matrix to track the positions of player 1 and 2.

**Test Cases:**

1. Verify the correct initialization of the game.
2. Confirm that user moves update the board and scores correctly.
3. Test the AI's decision-making with different initial board states.
4. Check the game-over messages for different outcomes.
5. Ensure the play-again button resets the game as expected.

**DESIGN AND DEVELOPMENT:**

**Algorithm Design:**

**Minimax Algorithm**:

Developed functions for maximizing and minimizing players to determine the best move.

**Methodology:**

Object-oriented Approach:

Structured the code using a class-based approach for better organization and readability.

**Implementation:**

Game Logic:

Coded the game logic using Python and Tkinter.

Testing:

Conducted tests for various scenarios to ensure the functionality and reliability of the code.

TECHNOLOGICAL IMPROVEMENT AND LEARNING OUTCOME:

**Technological Improvement:**

***Minimax Algorithm Implementation:***

Enhanced the game by incorporating the minimax algorithm for AI decision-making, providing a challenging opponent.

Reasoning:

***Optimal Decision-Making:***

The minimax algorithm ensures that the AI player makes optimal decisions, creating a more engaging and competitive gaming experience.

**SOURCE CODE:**

***Structure.py:***

import copy

import time

sgn = lambda n: 0 if n == 0 else n/abs(n)

class Board():

    def \_\_init\_\_(self, n=6, m=9, new\_move=1):

        self.m = m

        self.n = n

        self.board = [[0 for i in range(self.n)] for i in range(self.m)]

        self.new\_move = new\_move

    def \_\_getitem\_\_(self, pos):

        return self.board[pos[0]][pos[1]]

    def \_\_setitem\_\_(self, pos, value):

        self.board[pos[0]][pos[1]]=value

    def \_\_str\_\_(self):

        s = ""

        for i in range(self.m):

            for j in range(self.n):

                s += str(self[(i,j)])

                s += " "

            s += "\n"

        return s

    def hash(self):

        return str(self.board)+str(self.new\_move)

    def critical\_mass(self,pos):

        if pos == (0,0) or pos == (self.m - 1, self.n - 1) or pos == (self.m - 1, 0) or pos == (0, self.n - 1):

            return 2

        elif pos[0] == 0 or pos[0] == self.m-1 or pos[1] == 0 or pos[1] == self.n-1:

            return 3

        else:

            return 4

    def neighbors(self,pos):

        n = []

        for i in [(pos[0],pos[1]+1), (pos[0],pos[1]-1), (pos[0]+1,pos[1]), (pos[0]-1,pos[1])]:

            if 0 <= i[0] < self.m and 0 <= i[1] < self.n:

                n.append(i)

        return n

def move(board, pos):

    board = copy.deepcopy(board)

    assert board.new\_move == sgn(board[pos]) or 0 == sgn(board[pos])

    board[pos] = int(board[pos] + board.new\_move)

    t = time.time()

    while True:

        unstable = []

        for pos in [(x,y) for x in range(board.m) for y in range(board.n)]:

            if abs(board[pos]) >= board.critical\_mass(pos):

                unstable.append(pos)

        if time.time() - t >= 3:

            #Can't afford to spend more time, strange loop here!

            #print board, pos

            #raw\_input()

            break

        #print board

        #raw\_input()

        if not unstable:

            break

        for pos in unstable:

            board[pos] -= (board.new\_move\*board.critical\_mass(pos))

            for i in board.neighbors(pos):

                board[i] = int(sgn(board.new\_move)\*(abs(board[i])+1))

    board.new\_move \*= -1

    return board

def chains(board,player):

    board = copy.deepcopy(board)

    lengths = []

    for pos in [(x,y) for x in range(board.m) for y in range(board.n)]:

        if abs(board[pos]) == (board.critical\_mass(pos) - 1) and sgn(board[pos]) == player:

            l = 0

            visiting\_stack = []

            visiting\_stack.append(pos)

            while visiting\_stack:

                pos = visiting\_stack.pop()

                board[pos] = 0

                l += 1

                for i in board.neighbors(pos):

                    if abs(board[i]) == (board.critical\_mass(i) - 1) and sgn(board[i]) == player:

                        visiting\_stack.append(i)

            lengths.append(l)

    return lengths

def score(board, player):

    sc = 0

    my\_orbs, enemy\_orbs = 0, 0

    for pos in [(x,y) for x in range(board.m) for y in range(board.n)]:

        if sgn(board[pos]) == player:

            my\_orbs += abs(board[pos])

            flag\_not\_vulnerable = True

            for i in board.neighbors(pos):

                if sgn(board[i]) == -player and (abs(board[i]) == board.critical\_mass(i) - 1):

                    sc -= 5-board.critical\_mass(pos)

                    flag\_not\_vulnerable = False

            if flag\_not\_vulnerable:

                #The edge Heuristic

                if board.critical\_mass(pos) == 3:

                    sc += 2

                #The corner Heuristic

                elif board.critical\_mass(pos) == 2:

                    sc += 3

                #The unstability Heuristic

                if abs(board[pos]) == board.critical\_mass(pos) - 1:

                    sc += 2

                #The vulnerablity Heuristic

        else:

            enemy\_orbs += abs(board[pos])

    #The number of Orbs Heuristic

    sc += my\_orbs

    #You win when the enemy has no orbs

    if enemy\_orbs == 0 and my\_orbs > 1:

        return 10000

    #You loose when you have no orbs

    elif my\_orbs == 0 and enemy\_orbs > 1:

        return -10000

    #The chain Heuristic

    sc += sum([2\*i for i in chains(board,player) if i > 1])

    return sc

**algo.py:**

from structure import \*

def bestn(board,n=10):

    conf = {}

    for pos in [(x,y) for x in range(board.m) for y in range(board.n)]:

        if board.new\_move == sgn(board[pos]) or 0 == sgn(board[pos]):

            conf[pos] = score(move(board,pos),board.new\_move)

            #Return just the winning position in case you find one

            if conf[pos]==10000:

                return [pos]

    return sorted(conf, key=conf.get, reverse=True)[:n]

def minimax(board,depth=3,breadth=5):

    best\_moves = bestn(board,n=breadth)

    best\_pos, best\_val = (best\_moves[0], score(move(board,best\_moves[0]),board.new\_move))

    if depth == 1:

        return best\_pos, best\_val

    for b\_new\_pos in bestn(board):

        b\_new = move(board,b\_new\_pos)

        val = minimax(b\_new, depth=depth-1)[1]

        if val > best\_val:

            best\_val = val

            best\_pos = b\_new\_pos

    return best\_pos, best\_val

def alpha\_beta(board, depth=3, alpha=float('-inf'), beta=float('inf')):

    def max\_value(board, depth, alpha, beta):

        if depth == 0:

            return score(board, board.new\_move)

        value = float('-inf')

        for move in valid\_moves(board):

            new\_board = move(board, move)

            value = max(value, min\_value(new\_board, depth-1, alpha, beta))

            alpha = max(alpha, value)

            if alpha >= beta:

                break  # Beta cutoff

        return value

    def min\_value(board, depth, alpha, beta):

        if depth == 0:

            return score(board, board.new\_move)

        value = float('inf')

        for move in valid\_moves(board):

            new\_board = move(board, move)

            value = min(value, max\_value(new\_board, depth-1, alpha, beta))

            beta = min(beta, value)

            if alpha >= beta:

                break  # Alpha cutoff

        return value

    best\_move = None

    best\_val = float('-inf')

    for move in valid\_moves(board):

        new\_board = move(board, move)

        value = min\_value(new\_board, depth-1, alpha, beta)

        if value > best\_val:

            best\_val = value

            best\_move = move

        alpha = max(alpha, value)

        if alpha >= beta:

            break  # Beta cutoff

    return best\_move, best\_val

def valid\_moves(board):

    return [(x, y) for x in range(board.m) for y in range(board.n)

            if board.new\_move == sgn(board[(x, y)]) or 0 == sgn(board[(x, y)])]

**game.py:**

import pygame

import minimax, structure

import \_thread as thread

m, n = 9, 6

pygame.init()

surface = pygame.display.set\_mode((50\*n, 50\*m))

pygame.display.set\_caption('Chain Reaction')

lock = thread.allocate\_lock()

FUNCS = {

    'minimax':minimax.minimax,

    'alphabeta':minimax.alpha\_beta

}

def drawBoard(board=structure.Board()):

    surface.fill((0,0,0))

    font = pygame.font.Font('Font.ttf', 48)

    for pos in [(x,y) for x in range(board.m) for y in range(board.n)]:

        if abs(board[pos]) >= board.critical\_mass(pos):

            color = (255,255,0)

        elif structure.sgn(board[pos]) == 0:

            color = (90,90,90)

        elif structure.sgn(board[pos]) == 1:

            color = (255,0,0)

        else:

            color = (0,255,0)

        text = font.render(str(board[pos])[-1], 1, color)

        textpos = text.get\_rect(centerx = pos[1]\*50 + 25, centery = pos[0]\*50 + 25)

        surface.blit(text, textpos)

    pygame.display.update()

def slowMove(board, pos):

    board = structure.copy.deepcopy(board)

    assert board.new\_move == structure.sgn(board[pos]) or 0 == structure.sgn(board[pos])

    board[pos] = board[pos] + board.new\_move

    while True:

        drawBoard(board)

        pygame.time.wait(250)

        unstable = []

        for pos in [(x,y) for x in range(board.m) for y in range(board.n)]:

            if abs(board[pos]) >= board.critical\_mass(pos):

                unstable.append(pos)

        #raw\_input()

        if not unstable:

            break

        for pos in unstable:

            board[pos] -= board.new\_move\*board.critical\_mass(pos)

            for i in board.neighbors(pos):

                board[i] = structure.sgn(board.new\_move)\*(abs(board[i])+1)

    drawBoard(board)

    lock.release()

def show\_move(pos):

    rect = pygame.Rect(pos[1]\*50,pos[0]\*50,50,50)

    pygame.draw.rect(surface,(255,255,0),rect,0)

    pygame.display.update()

    pygame.time.wait(250)

def main():

    global m,n, surface

    FUNC = FUNCS['minimax']

    #start screen

    font = pygame.font.Font('Font.ttf', 72)

    text = font.render("Red", 1, (255,0,0))

    textpos = text.get\_rect(centerx = 25\*n, centery = 12\*m)

    surface.blit(text, textpos)

    text = font.render("Green", 1, (0,255,0))

    textpos = text.get\_rect(centerx = 25\*n, centery = 36\*m)

    surface.blit(text, textpos)

    font = pygame.font.Font('Font.ttf', 12)

    text = font.render("Choose a Color", 1, (100,100,100))

    textpos = text.get\_rect(centerx = 25\*n, centery = 25\*m)

    surface.blit(text, textpos)

    pygame.display.update()

    this\_loop = True

    while this\_loop:

        for event in pygame.event.get():

            if event.type == pygame.MOUSEBUTTONDOWN:

                y = pygame.mouse.get\_pos()[1]

                if y < 25\*m:

                    player\_first = True

                else:

                    player\_first = False

                this\_loop = False

    #depth screen

    surface.fill((0,0,0))

    font = pygame.font.Font('Font.ttf', 12)

    text = font.render("How deep should I look?", 1, (100,100,100))

    textpos = text.get\_rect(centerx = 25\*n, centery = 12\*m)

    surface.blit(text, textpos)

    font = pygame.font.Font('Font.ttf', 48)

    depth = 3

    text = font.render(str(depth),1,(255,255,0))

    textpos = text.get\_rect(centerx = 25\*n, centery = 25\*m)

    surface.blit(text, textpos)

    pygame.display.update()

    this\_loop = True

    while this\_loop:

        for event in pygame.event.get():

            if (event.type == pygame.KEYDOWN and event.key == pygame.K\_RETURN) or event.type == pygame.MOUSEBUTTONDOWN:

                this\_loop = False

            elif event.type == pygame.KEYDOWN:

                if event.key < 256 and chr(event.key) in '1234567890':

                    rect = pygame.Rect(12\*m,25\*n,100,100)

                    pygame.draw.rect(surface,(0,0,0),rect,0)

                    pygame.display.update()

                    depth = int(chr(event.key))

                    text = font.render(str(depth),1,(255,255,0))

                    textpos = text.get\_rect(centerx = 25\*n, centery = 25\*m)

                    surface.blit(text, textpos)

                    pygame.display.update()

    #rows screen

    surface.fill((0,0,0))

    font = pygame.font.Font('Font.ttf', 12)

    text = font.render("How many rows?", 1, (100,100,100))

    textpos = text.get\_rect(centerx = 25\*n, centery = 12\*m)

    surface.blit(text, textpos)

    font = pygame.font.Font('Font.ttf', 48)

    rows = 9

    text = font.render(str(rows),1,(255,255,0))

    textpos = text.get\_rect(centerx = 25\*n, centery = 25\*m)

    surface.blit(text, textpos)

    pygame.display.update()

    this\_loop = True

    while this\_loop:

        for event in pygame.event.get():

            if (event.type == pygame.KEYDOWN and event.key == pygame.K\_RETURN) or event.type == pygame.MOUSEBUTTONDOWN:

                this\_loop = False

            elif event.type == pygame.KEYDOWN:

                if event.key < 256 and chr(event.key) in '1234567890':

                    rect = pygame.Rect(12\*m,25\*n,100,100)

                    pygame.draw.rect(surface,(0,0,0),rect,0)

                    pygame.display.update()

                    rows = int(chr(event.key))

                    text = font.render(str(rows),1,(255,255,0))

                    textpos = text.get\_rect(centerx = 25\*n, centery = 25\*m)

                    surface.blit(text, textpos)

                    pygame.display.update()

    #columns screen

    surface.fill((0,0,0))

    font = pygame.font.Font('Font.ttf', 12)

    text = font.render("How many columns?", 1, (100,100,100))

    textpos = text.get\_rect(centerx = 25\*n, centery = 12\*m)

    surface.blit(text, textpos)

    font = pygame.font.Font('Font.ttf', 48)

    columns = 6

    text = font.render(str(columns),1,(255,255,0))

    textpos = text.get\_rect(centerx = 25\*n, centery = 25\*m)

    surface.blit(text, textpos)

    pygame.display.update()

    this\_loop = True

    while this\_loop:

        for event in pygame.event.get():

            if (event.type == pygame.KEYDOWN and event.key == pygame.K\_RETURN) or event.type == pygame.MOUSEBUTTONDOWN:

                this\_loop = False

            elif event.type == pygame.KEYDOWN:

                if event.key < 256 and chr(event.key) in '1234567890':

                    rect = pygame.Rect(12\*m,25\*n,100,100)

                    pygame.draw.rect(surface,(0,0,0),rect,0)

                    pygame.display.update()

                    columns = int(chr(event.key))

                    text = font.render(str(columns),1,(255,255,0))

                    textpos = text.get\_rect(centerx = 25\*n, centery = 25\*m)

                    surface.blit(text, textpos)

                    pygame.display.update()

    #some initialization code

    m, n = rows, columns

    surface = pygame.display.set\_mode((50\*n, 50\*m))

    pygame.display.set\_caption('Chain Reaction')

    board = structure.Board(m=m,n=n)

    total\_moves = 0

    #game screen

    drawBoard(board)

    if not player\_first:

        new\_move = FUNC(board)[0]

        lock.acquire()

        thread.start\_new\_thread(slowMove, (board, new\_move))

        board = structure.move(board, new\_move)

        total\_moves += 1

    this\_loop = True

    while this\_loop:

        for event in pygame.event.get():

            if event.type == pygame.MOUSEBUTTONDOWN:

                x,y = pygame.mouse.get\_pos()

                x,y = x//50,y//50

                if not (board.new\_move == structure.sgn(board[(y,x)]) or 0 == structure.sgn(board[(y,x)])):

                    print("Illegal Move!")

                    continue

                show\_move((y,x))

                lock.acquire()

                thread.start\_new\_thread(slowMove, (board,(y,x)))

                board = structure.move(board,(y,x))

                total\_moves += 1

                if total\_moves >= 2:

                    if structure.score(board,board.new\_move\*(-1)) == 10000:

                        winner = board.new\_move\*(-1)

                        this\_loop = False

                        break

                new\_move = FUNC(board,depth)[0]

                show\_move(new\_move)

                lock.acquire()

                thread.start\_new\_thread(slowMove, (board, new\_move))

                board = structure.move(board, new\_move)

                total\_moves += 1

                if total\_moves >= 2:

                    if structure.score(board,board.new\_move\*(-1)) == 10000:

                        winner = board.new\_move\*(-1)

                        this\_loop = False

                        break

    #winning screen

    while lock.locked():

        continue

    m, n = 9, 6

    surface = pygame.display.set\_mode((50\*n, 50\*m))

    font = pygame.font.Font('Font.ttf', 72)

    pygame.display.set\_caption('Chain Reaction')

    if winner == 1:

        text = font.render("Red", 1, (255,0,0))

    else:

        text = font.render("Green", 1, (0,255,0))

    textpos = text.get\_rect(centerx = 25\*n, centery = 12\*m)

    surface.blit(text, textpos)

    font = pygame.font.Font('Font.ttf', 48)

    text = font.render("Wins!", 1, (100,100,100))

    textpos = text.get\_rect(centerx = 25\*n, centery = 25\*m)

    surface.blit(text, textpos)

    pygame.display.update()

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**METHODS DEFINITION:**

**1. move(board, pos)**

- Purpose: Represents a move in the Chain Reaction game.

- Implementation: `board[pos] = int(board[pos] + board.new\_move)`

**2. chains(board, player)**

- Purpose: Identifies chain lengths for a given player on the game board.

- Implementation: Returns a list of chain lengths.

**3. score(board, player)**

- Purpose: Calculates the score for a player based on various heuristics.

- Implementation: Computes and returns the overall score for the player.

**4. alpha\_beta(board, depth=3, alpha=float('-inf'), beta=float('inf'))**

- Purpose: Implements the Alpha-Beta Pruning algorithm for decision optimization.

- Implementation: The alpha-beta pruning algorithm for selecting the best move.

5. bestn(board, n=10)

- Purpose: Finds the best `n` moves for the current player based on scores.

- Implementation: Returns a list of the top `n` moves sorted by their scores.

**6. minimax(board, depth=3, breadth=5)**

- Purpose: Implements the Minimax algorithm for strategic decision-making.

- Implementation: Recursively explores the game tree to find the best move.

**7. valid\_moves(board)**

- Purpose: Identifies all valid moves for the current player on the game board.

- Implementation: Returns a list of valid moves.

**8. slowMove(board, pos)**

- Purpose: Simulates a slow motion move for visualization purposes.

- Implementation: Simulates the move on the board with a delay for visualization.

**9. drawBoard(board=structure.Board())**

- Purpose: Draws the current state of the game board.

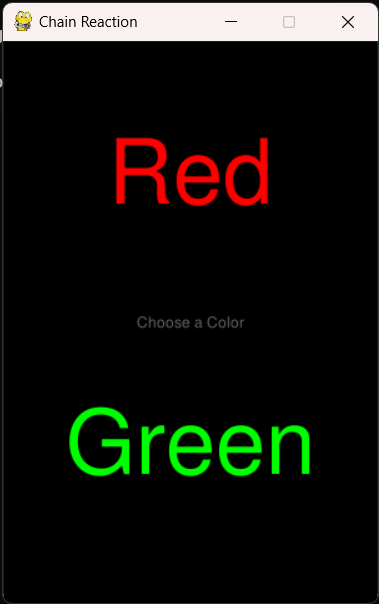
- Implementation: Uses Pygame to draw the current state of the Chain Reaction game board.

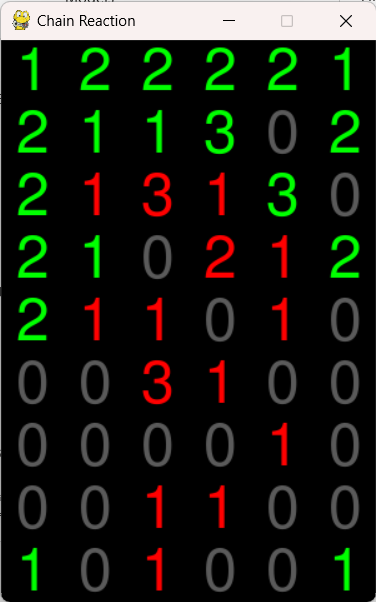
**10. main()**

- Purpose: Main function to initialize and run the Chain Reaction game.

- Implementation: Contains the main logic for user interaction, AI moves, and game progression.

**OUTPUT SCREENSHOTS**

**** ****

**LEARNING OUTCOMES**

GUI Development:

Gained proficiency in developing graphical user interfaces using Tkinter.

Algorithm Implementation:

Acquired knowledge and practical experience in implementing the minimax algorithm for decision-making in games.

User Interaction:

Learned how to handle user input, game logic, and interface design in Python.