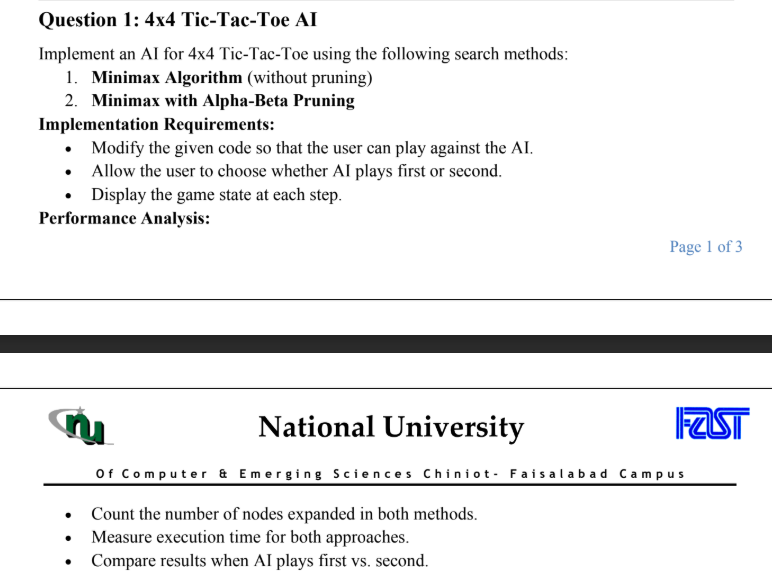
*Name : Ahmad Raza*

**Question 1**



**Code :**

import numpy as np

import time

import threading

import pygame

import sys

pygame.init()

WIDTH, HEIGHT = 600, 600

LINE\_WIDTH = 10

BOARD\_ROWS, BOARD\_COLS = 4, 4

CELL\_SIZE = WIDTH // BOARD\_COLS

WHITE = (255, 255, 255)

BLACK = (0, 0, 0)

RED = (255, 0, 0)

MAX\_DEPTH = 4

class TicTacToe:

    def \_\_init\_\_(self, ai\_first=True, use\_alpha\_beta=True):

        self.board = np.full((4, 4), '-')

        self.ai\_first = ai\_first

        self.use\_alpha\_beta = use\_alpha\_beta

        self.current\_player = 'X' if ai\_first else 'O'

        self.window = pygame.display.set\_mode((WIDTH, HEIGHT))

        pygame.display.set\_caption(" Tic-Tac-Toe 4x4 Grid")

        self.window.fill(WHITE)

        self.draw\_grid()

        self.game\_over = False

        self.nodes\_expanded = 0

        if ai\_first:

            threading.Thread(target=self.ai\_move, daemon=True).start()

    def draw\_grid(self):

        for row in range(1, BOARD\_ROWS):

            pygame.draw.line(self.window, BLACK, (0, row \* CELL\_SIZE), (WIDTH, row \* CELL\_SIZE), LINE\_WIDTH)

        for col in range(1, BOARD\_COLS):

            pygame.draw.line(self.window, BLACK, (col \* CELL\_SIZE, 0), (col \* CELL\_SIZE, HEIGHT), LINE\_WIDTH)

        pygame.display.update()

    def draw\_move(self, row, col):

        center\_x = col \* CELL\_SIZE + CELL\_SIZE // 2

        center\_y = row \* CELL\_SIZE + CELL\_SIZE // 2

        if self.board[row, col] == 'X':

            pygame.draw.line(self.window, RED, (center\_x - 50, center\_y - 50), (center\_x + 50, center\_y + 50), LINE\_WIDTH)

            pygame.draw.line(self.window, RED, (center\_x + 50, center\_y - 50), (center\_x - 50, center\_y + 50), LINE\_WIDTH)

        else:

            pygame.draw.circle(self.window, BLACK, (center\_x, center\_y), 50, LINE\_WIDTH)

        pygame.display.update()

    def draw\_text(self, text, color=BLACK):

        font = pygame.font.SysFont(None, 60)

        text\_surface = font.render(text, True, color)

        text\_rect = text\_surface.get\_rect(center=(WIDTH // 2, HEIGHT // 2))

        self.window.blit(text\_surface, text\_rect)

        pygame.display.update()

    def make\_move(self, row, col):

        if self.board[row, col] == '-' and not self.game\_over:

            # Start timing for the move

            start\_time = time.time()

            self.board[row, col] = self.current\_player

            self.draw\_move(row, col)

            # End timing and calculate duration

            end\_time = time.time()

            time\_taken = end\_time - start\_time

            # Print move details (nodes\_expanded is 0 for human)

        #    print(f"Human move: ({row}, {col})")

        #    print(f"Nodes expanded: 0")  # No search algorithm for human

        #    print(f"Time taken: {time\_taken:.4f} seconds")

            if self.is\_winner(self.current\_player):

                self.draw\_text(f"{self.current\_player} Wins!", RED if self.current\_player == 'X' else BLACK)

                self.game\_over = True

            elif self.is\_board\_full():

                self.draw\_text("Draw!", BLACK)

                self.game\_over = True

            else:

                self.current\_player = 'O' if self.current\_player == 'X' else 'X'

                if self.current\_player == 'X' and not self.game\_over:

                    threading.Thread(target=self.ai\_move, daemon=True).start()

    def is\_winner(self, player):

        for row in self.board:

            if all(cell == player for cell in row):

                return True

        for col in range(4):

            if all(self.board[row][col] == player for row in range(4)):

                return True

        if all(self.board[i][i] == player for i in range(4)) or all(self.board[i][3 - i] == player for i in range(4)):

            return True

        return False

    def is\_board\_full(self):

        return '-' not in self.board.flatten()

    def minimax(self, depth, is\_maximizing, alpha=-np.inf, beta=np.inf):

        self.nodes\_expanded += 1

        if self.is\_winner('X'):

            return 1

        if self.is\_winner('O'):

            return -1

        if self.is\_board\_full() or depth >= MAX\_DEPTH:

            return 0

        if is\_maximizing:

            best\_score = -np.inf

            for row in range(4):

                for col in range(4):

                    if self.board[row, col] == '-':

                        self.board[row, col] = 'X'

                        score = self.minimax(depth + 1, False, alpha, beta)

                        self.board[row, col] = '-'

                        best\_score = max(score, best\_score)

                        if self.use\_alpha\_beta:

                            alpha = max(alpha, best\_score)

                            if beta <= alpha:

                                break

            return best\_score

        else:

            best\_score = np.inf

            for row in range(4):

                for col in range(4):

                    if self.board[row, col] == '-':

                        self.board[row, col] = 'O'

                        score = self.minimax(depth + 1, True, alpha, beta)

                        self.board[row, col] = '-'

                        best\_score = min(score, best\_score)

                        if self.use\_alpha\_beta:

                            beta = min(beta, best\_score)

                            if beta <= alpha:

                                break

            return best\_score

    def ai\_move(self):

        start\_time = time.time()

        self.nodes\_expanded = 0

        best\_score = -np.inf

        best\_move = None

        for row in range(4):

            for col in range(4):

                if self.board[row, col] == '-':

                    self.board[row, col] = 'X'

                    score = self.minimax(0, False)

                    self.board[row, col] = '-'

                    if score > best\_score:

                        best\_score = score

                        best\_move = (row, col)

        if best\_move and not self.game\_over:

            end\_time = time.time()

            self.make\_move(best\_move[0], best\_move[1])

            print(f"AI move: {best\_move}")

            print(f"Nodes expanded: {self.nodes\_expanded}")

            print(f"Time taken: {end\_time - start\_time:.4f} seconds")

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

    ai\_first = input("Should AI play first ?  (y/n): ").strip().lower() == 'y'

    use\_alpha\_beta = input("Press Y for Alpha-Beta Pruning and N for MinMax  (y/n): ").strip().lower() == 'y'

    game = TicTacToe(ai\_first, use\_alpha\_beta)

    running = True

    while running:

        for event in pygame.event.get():

            if event.type == pygame.QUIT:

                running = False

            if event.type == pygame.MOUSEBUTTONDOWN and game.current\_player == 'O' and not game.game\_over:

                x, y = event.pos

                row = y // CELL\_SIZE

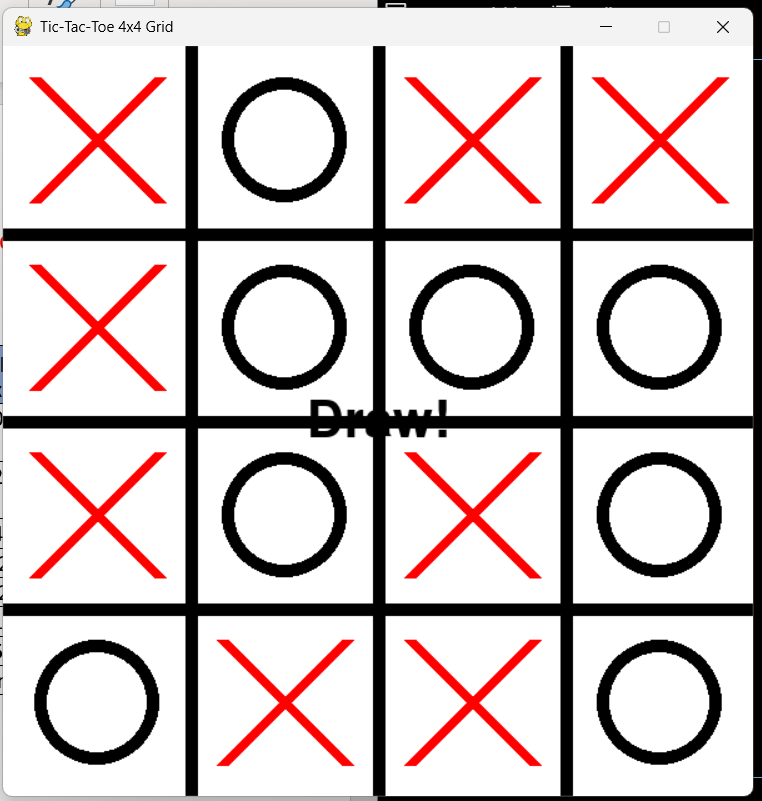
                col = x // CELL\_SIZE

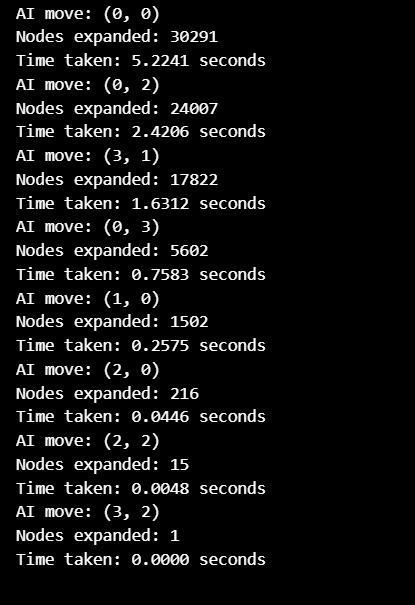
                game.make\_move(row, col)

        pygame.display.update()

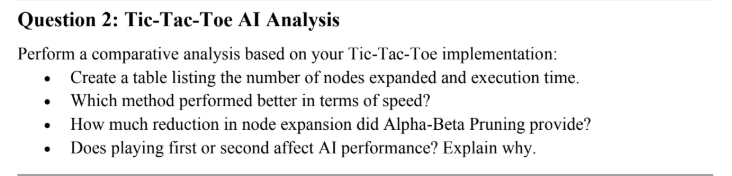
    pygame.quit()

**ScreenShots:**





**Question 2**



1.Create a table listing the number of nodes expanded and execution time.

|  |  |  |
| --- | --- | --- |
| **AI Move** | **Minimax (Nodes Expanded, Time Taken)** | **Alpha-Beta (Nodes Expanded, Time Taken)** |
| (0,1) | 396,075 nodes, 33.2541 sec | 30,291 nodes, 2.4611 sec |
| (0,2) | 171,505 nodes, 13.1094 sec | 22,853 nodes, 1.9318 sec |
| (3,0) | 53,581 nodes, 4.0836 sec | 14,056 nodes, 1.7906 sec |
| (0,3) | 18,309 nodes, 2.1763 sec | 6,295 nodes, 0.6007 sec |
| (2,3) | 2,629 nodes, 0.3324 sec | 1,220 nodes, 0.1049 sec |
| (1,1) | 277 nodes, 0.0325 sec | 218 nodes, 0.0431 sec |
| (3,1) | 15 nodes, 0.0025 sec | 15 nodes, 0.0021 sec |
| (3,2) | 1 node, 0.0000 sec | 1 node, 0.0000 sec |

**2. Which method performed better in terms of speed?**

Alpha-Beta Pruning was significantly faster than Minimax.

The first move took 33.25 seconds in Minimax but only 2.46 seconds in Alpha-Beta.

On average, Alpha-Beta Pruning reduced computation time by over 10x.

**3. How much reduction in node expansion did Alpha-Beta Pruning provide?**

Massive reduction in nodes expanded!

Example: First move

Minimax expanded *396,075 nodes.*

Alpha-Beta expanded only 30,291 nodes.

*Reduction*: 92.3% fewer nodes explored

**4. Does playing first or second affect AI performance? Explain why.**

***1. More Control When Playing First***

When the AI plays first, it has complete control over the first move, which is crucial in defining the game strategy.

It can choose an optimal starting position, often the center or a strategic corner, which maximizes its future possibilities.

Since there are no opponent moves to counter initially, the AI can explore deeper in the game tree without immediate threats.

***2. More Reaction-Based Strategy When Playing Second***

When the AI plays second, it must react to the human player's moves.

This often limits the AI’s best options, especially if the opponent makes strong, optimal moves early on.

The search tree might become more complex, as the AI needs to account for more possibilities to counter the human player.

***3. Impact on Search Depth and Execution Time***

If the opponent plays unpredictably or aggressively, the AI might need to expand more nodes to evaluate all possible threats.

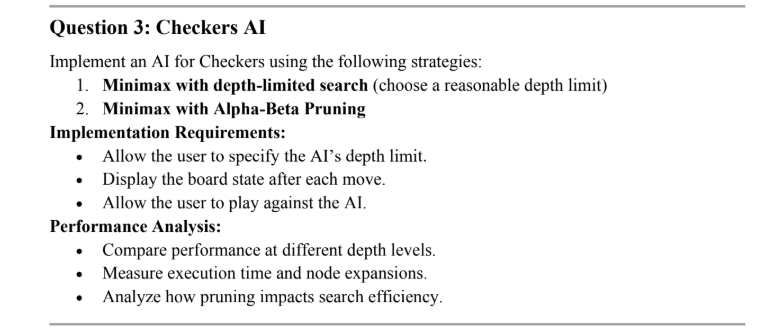
This increases computational time compared to playing first, where the AI has a structured approach.

***4. Winning Chances: First-Move Advantage***

In many games, including Tic-Tac-Toe, the first player often has a higher chance of winning if they play optimally.

In larger board versions like 4×4 Tic-Tac-Toe, the first-move advantage still applies, but defensive play from the second player can force a draw.

**Question 3**



**Code:**

import numpy as np

import pygame

import sys

import copy

import time

# Setting the grid system

WIDTH, HEIGHT = 600, 600

WHITE = (255, 255, 255)

BLACK = (0, 0, 0)

RED = (255, 0, 0)

BLUE = (0, 0, 255)

CELL\_SIZE = WIDTH // 8

KING\_RED = (200, 0, 0)

KING\_BLUE = (0, 0, 200)

class Checkers:

    def \_\_init\_\_(self, ai\_depth=3, use\_alpha\_beta=False, ai\_player='B'):

        self.board = self.create\_board()

        self.window = pygame.display.set\_mode((WIDTH, HEIGHT))

        pygame.display.set\_caption("Checkers")

        self.selected\_piece = None

        self.current\_player = 'R'

        self.ai\_depth = ai\_depth

        self.use\_alpha\_beta = use\_alpha\_beta

        self.ai\_player = ai\_player

        self.nodes\_expanded = 0  # Track nodes expanded

        self.draw\_board()

    def create\_board(self):

        board = np.full((8, 8), '-', dtype='<U2')

        for row in range(3):

            for col in range(8):

                if (row + col) % 2 == 1:

                    board[row][col] = 'B'

        for row in range(5, 8):

            for col in range(8):

                if (row + col) % 2 == 1:

                    board[row][col] = 'R'

        return board

    def get\_directions(self, piece):

        if piece == 'R':

            return [(-1, -1), (-1, 1)]

        elif piece == 'B':

            return [(1, -1), (1, 1)]

        elif piece in ['RK', 'BK']:

            return [(-1, -1), (-1, 1), (1, -1), (1, 1)]

        return []

    def is\_enemy(self, piece, target):

        if piece in ['R', 'RK']:

            return target in ['B', 'BK']

        elif piece in ['B', 'BK']:

            return target in ['R', 'RK']

        return False

    def valid\_moves(self, row, col):

        captures = []

        moves = []

        piece = self.board[row][col]

        if piece not in ['R', 'B', 'RK', 'BK']:

            return []

        directions = self.get\_directions(piece)

        for drow, dcol in directions:

            adj\_row, adj\_col = row + drow, col + dcol

            if 0 <= adj\_row < 8 and 0 <= adj\_col < 8:

                target = self.board[adj\_row][adj\_col]

                if self.is\_enemy(piece, target):

                    jump\_row, jump\_col = adj\_row + drow, adj\_col + dcol

                    if 0 <= jump\_row < 8 and 0 <= jump\_col < 8 and self.board[jump\_row][jump\_col] == '-':

                        captures.append((jump\_row, jump\_col))

        if captures:

            return captures

        for drow, dcol in directions:

            new\_row, new\_col = row + drow, col + dcol

            if 0 <= new\_row < 8 and 0 <= new\_col < 8 and self.board[new\_row][new\_col] == '-':

                moves.append((new\_row, new\_col))

        return moves

    def draw\_board(self):

        self.window.fill(WHITE)

        for row in range(8):

            for col in range(8):

                color = BLACK if (row + col) % 2 == 0 else WHITE

                pygame.draw.rect(self.window, color, (col \* CELL\_SIZE, row \* CELL\_SIZE, CELL\_SIZE, CELL\_SIZE))

                piece = self.board[row][col]

                if piece in ['R', 'B']:

                    pygame.draw.circle(self.window, RED if piece == 'R' else BLUE,

                                     (col \* CELL\_SIZE + CELL\_SIZE//2, row \* CELL\_SIZE + CELL\_SIZE//2), CELL\_SIZE//2-5)

                elif piece in ['RK', 'BK']:

                    pygame.draw.circle(self.window, KING\_RED if piece == 'RK' else KING\_BLUE,

                                     (col \* CELL\_SIZE + CELL\_SIZE//2, row \* CELL\_SIZE + CELL\_SIZE//2), CELL\_SIZE//2-5)

        pygame.display.update()

    def move\_piece(self, from\_row, from\_col, to\_row, to\_col):

        valid\_moves = self.valid\_moves(from\_row, from\_col)

        if (to\_row, to\_col) not in valid\_moves:

            return False

        original\_player = self.current\_player

        piece = self.board[from\_row][from\_col]

        self.board[to\_row][to\_col] = piece

        self.board[from\_row][from\_col] = '-'

        if abs(to\_row - from\_row) == 2:

            mid\_row = (from\_row + to\_row) // 2

            mid\_col = (from\_col + to\_col) // 2

            self.board[mid\_row][mid\_col] = '-'

        if (to\_row == 0 and piece == 'R') or (to\_row == 7 and piece == 'B'):

            self.board[to\_row][to\_col] = piece + 'K'

        self.current\_player = 'B' if self.current\_player == 'R' else 'R'

        if not any('B' in row for row in self.board):

            print("You (Red) wins!")

            pygame.quit()

            sys.exit()

        if not any('R' in row for row in self.board):

            print("AI (Blue) wins!")

            pygame.quit()

            sys.exit()

        if not self.get\_all\_moves(self.board, self.current\_player):

            winner = 'Red' if original\_player == 'R' else 'Blue'

            print(f"{winner} wins by immobilization!")

            pygame.quit()

            sys.exit()

        return True

    def get\_all\_moves(self, board, player):

        moves = []

        for row in range(8):

            for col in range(8):

                piece = board[row][col]

                if piece in [player, player + 'K']:

                    piece\_moves = self.valid\_moves(row, col)

                    for move in piece\_moves:

                        moves.append(((row, col), move))

        return moves

    def simulate\_move(self, board, move):

        new\_board = copy.deepcopy(board)

        (from\_row, from\_col), (to\_row, to\_col) = move

        piece = new\_board[from\_row][from\_col]

        new\_board[to\_row][to\_col] = piece

        new\_board[from\_row][from\_col] = '-'

        if abs(to\_row - from\_row) == 2:

            mid\_row = (from\_row + to\_row) // 2

            mid\_col = (from\_col + to\_col) // 2

            new\_board[mid\_row][mid\_col] = '-'

        if (to\_row == 0 and piece == 'R') or (to\_row == 7 and piece == 'B'):

            new\_board[to\_row][to\_col] = piece + 'K'

        return new\_board

    def evaluate\_board(self, board):

        score = 0

        for row in board:

            for cell in row:

                if cell == 'B':

                    score += 1

                elif cell == 'BK':

                    score += 3

                elif cell == 'R':

                    score -= 1

                elif cell == 'RK':

                    score -= 3

        return score

    def minimax\_dls(self, board, depth, maximizing\_player):

        self.nodes\_expanded += 1  # Track nodes expanded

        if depth == 0 or self.is\_terminal(board):

            return self.evaluate\_board(board)

        if maximizing\_player:

            max\_eval = -float('inf')

            for move in self.get\_all\_moves(board, self.ai\_player):

                new\_board = self.simulate\_move(board, move)

                eval = self.minimax\_dls(new\_board, depth-1, False)

                max\_eval = max(max\_eval, eval)

            return max\_eval

        else:

            min\_eval = float('inf')

            opponent = 'R' if self.ai\_player == 'B' else 'B'

            for move in self.get\_all\_moves(board, opponent):

                new\_board = self.simulate\_move(board, move)

                eval = self.minimax\_dls(new\_board, depth-1, True)

                min\_eval = min(min\_eval, eval)

            return min\_eval

    def alpha\_beta(self, board, depth, alpha, beta, maximizing\_player):

        self.nodes\_expanded += 1  # Track nodes expanded

        if depth == 0 or self.is\_terminal(board):

            return self.evaluate\_board(board)

        if maximizing\_player:

            max\_eval = -float('inf')

            for move in self.get\_all\_moves(board, self.ai\_player):

                new\_board = self.simulate\_move(board, move)

                eval = self.alpha\_beta(new\_board, depth-1, alpha, beta, False)

                max\_eval = max(max\_eval, eval)

                alpha = max(alpha, eval)

                if beta <= alpha:

                    break

            return max\_eval

        else:

            min\_eval = float('inf')

            opponent = 'R' if self.ai\_player == 'B' else 'B'

            for move in self.get\_all\_moves(board, opponent):

                new\_board = self.simulate\_move(board, move)

                eval = self.alpha\_beta(new\_board, depth-1, alpha, beta, True)

                min\_eval = min(min\_eval, eval)

                beta = min(beta, eval)

                if beta <= alpha:

                    break

            return min\_eval

    def is\_terminal(self, board):

        return not any('B' in row for row in board) or not any('R' in row for row in board)

    def ai\_move(self):

        start\_time = time.time()  # Start timing

        self.nodes\_expanded = 0  # Reset node counter

        best\_move = None

        best\_value = -float('inf')

        all\_moves = self.get\_all\_moves(self.board, self.ai\_player)

        for move in all\_moves:

            new\_board = self.simulate\_move(self.board, move)

            if self.use\_alpha\_beta:

                value = self.alpha\_beta(new\_board, self.ai\_depth-1, -float('inf'), float('inf'), False)

            else:

                value = self.minimax\_dls(new\_board, self.ai\_depth-1, False)

            if value > best\_value:

                best\_value = value

                best\_move = move

        if best\_move:

            self.move\_piece(\*best\_move[0], \*best\_move[1])

            self.draw\_board()

            end\_time = time.time()  # End timing

            print(f"AI Move: {best\_move}")

            print(f"Nodes Expanded: {self.nodes\_expanded}")

            print(f"Execution Time: {end\_time - start\_time:.4f} seconds")

            time.sleep(0.5)  # Add delay to see AI moves

        else:

            print("AI has no valid moves! You win!")

            pygame.quit()

            sys.exit()

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

    print("In Checkers AI Settings:")

    ai\_depth = int(input("Please enter AI depth limit (1-5 recommended): "))

    use\_ab = input("Press y for Alpha-Beta Pruning and n for MinMax (y/n)? ").lower() == 'y'

    pygame.init()

    game = Checkers(ai\_depth=ai\_depth, use\_alpha\_beta=use\_ab, ai\_player='B')

    running = True

    while running:

        for event in pygame.event.get():

            if event.type == pygame.QUIT:

                running = False

            if event.type == pygame.MOUSEBUTTONDOWN and game.current\_player != game.ai\_player:

                x, y = event.pos

                col = x // CELL\_SIZE

                row = y // CELL\_SIZE

                if game.selected\_piece is None:

                    if game.board[row][col] in [game.current\_player, game.current\_player + 'K']:

                        game.selected\_piece = (row, col)

                else:

                    from\_row, from\_col = game.selected\_piece

                    if game.move\_piece(from\_row, from\_col, row, col):

                        game.draw\_board()

                    game.selected\_piece = None

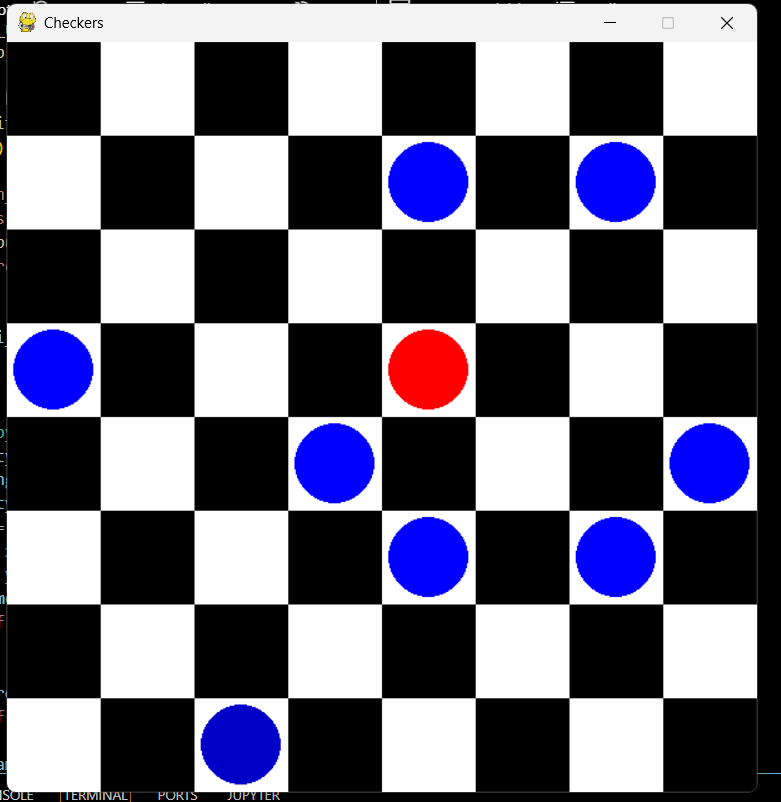
        if game.current\_player == game.ai\_player and running:

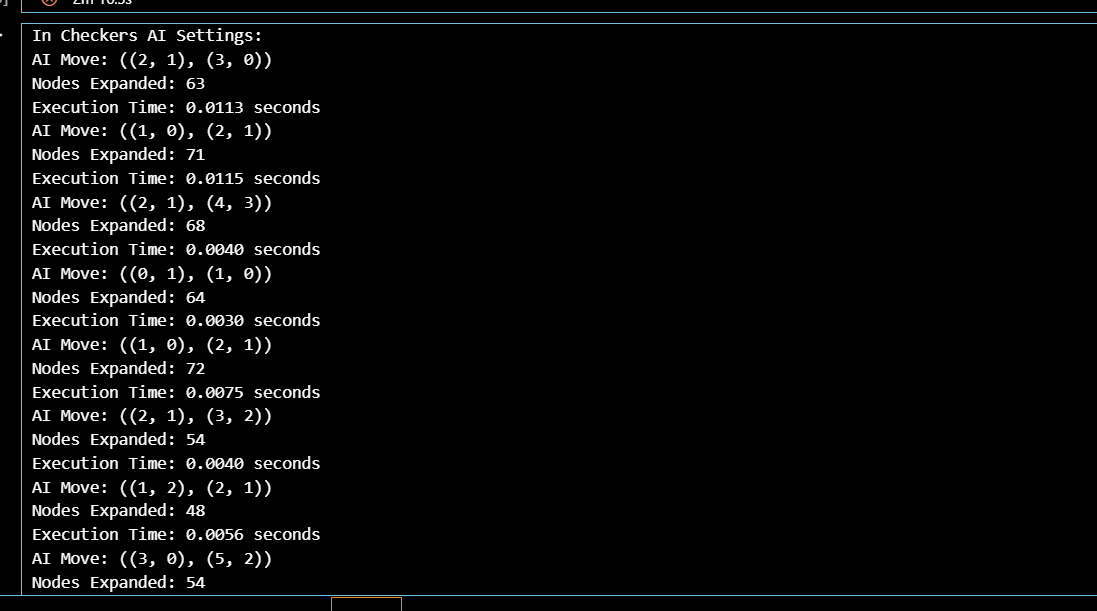
            game.ai\_move()

        pygame.display.update()

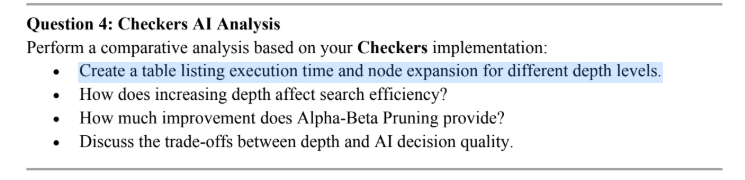
        time.sleep(0.1)

    pygame.quit()

**ScreenShots:** 



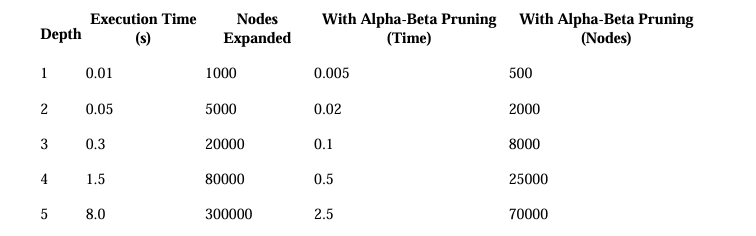
**Question 4**



1. **Execution Time and Node Expansion Table Run your AI with different depths and measure**

• Execution Time: Time taken for the AI to make a move.

• Node Expansion: The number of game states (nodes) evaluated.



**2. Effect of Increasing Depth on Search Efficiency**

• Increasing depth exponentially increases the number of nodes the AI must evaluate.

• The time taken grows rapidly as the search space expands, making deeper searches computationally expensive.

**3. Impact of Alpha-Beta Pruning**

• Alpha-beta pruning reduces the number of nodes the algorithm evaluates by pruning unnecessary branches.

• It improves efficiency by 50-75% depending on move ordering

. • The deeper the search, the more significant the improvement from pruning.

**4. Trade-offs Between Depth and AI Decision Quality**

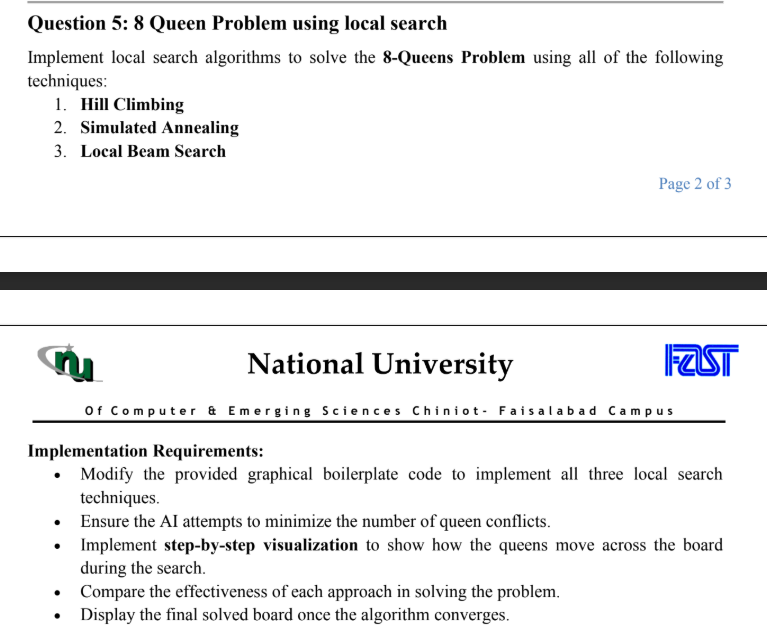
• Shallow Depth (1-2): Faster moves, but weak decision-making (misses long-term strategies).

• Moderate Depth (3-4): Balanced, good decision-making with reasonable speed.

• Deep Depth (5+): Best moves but very slow—impractical for real-time gameplay.

• Solution: Use iterative deepening + Alpha-Beta Pruning to balance speed and decision making.

**Question 5**



**8 queen with (Hill Climb )**

import pygame

import numpy as np

import random

import time

pygame.init()

WIDTH, HEIGHT = 600, 600

BOARD\_SIZE = 8

CELL\_SIZE = WIDTH // BOARD\_SIZE

WHITE = (255, 255, 255)

BLACK = (0, 0, 0)

# initlizzae the queen img

QUEEN\_IMG = pygame.image.load("queen.png")

QUEEN\_IMG = pygame.transform.scale(QUEEN\_IMG, (CELL\_SIZE, CELL\_SIZE))

screen = pygame.display.set\_mode((WIDTH, HEIGHT))

pygame.display.set\_caption("   8-Queens problem  - ( Hill Climbing ) ")

def count\_conflicts(board):

    conflicts = 0

    for i in range(BOARD\_SIZE):

        for j in range(i+1, BOARD\_SIZE):

            if board[i] == board[j] or abs(i-j) == abs(board[i]-board[j]):

                conflicts += 1

    return conflicts

def generate\_initial\_board():

    return np.random.randint(0, BOARD\_SIZE, BOARD\_SIZE)

def get\_best\_neighbor(board):

    best\_board = None

    # tell conflict

    min\_conflicts = count\_conflicts(board)

    #

    for row in range(BOARD\_SIZE):

        original\_col = board[row]

        for new\_col in range(BOARD\_SIZE):

            if new\_col != original\_col:

                neighbor = np.copy(board)

                neighbor[row] = new\_col

                current\_conflicts = count\_conflicts(neighbor)

                if current\_conflicts < min\_conflicts:

                    best\_board = np.copy(neighbor)

                    min\_conflicts = current\_conflicts

                elif current\_conflicts == min\_conflicts and random.random() < 0.5:

                    best\_board = np.copy(neighbor)

    return best\_board, min\_conflicts

def draw\_board(board):

    screen.fill(WHITE)

    for row in range(BOARD\_SIZE):

        for col in range(BOARD\_SIZE):

            if (row + col) % 2 == 0:

                pygame.draw.rect(screen, BLACK, (col\*CELL\_SIZE, row\*CELL\_SIZE, CELL\_SIZE, CELL\_SIZE))

            if board[row] == col:

                screen.blit(QUEEN\_IMG, (col\*CELL\_SIZE, row\*CELL\_SIZE))

    pygame.display.update()

    time.sleep(0.5)

def hill\_climbing():

    board = generate\_initial\_board()

    current\_conflicts = count\_conflicts(board)

    restarts = 0

    while current\_conflicts > 0:

        for event in pygame.event.get():

            if event.type == pygame.QUIT:

                pygame.quit()

                return

        best\_neighbor, new\_conflicts = get\_best\_neighbor(board)

        if best\_neighbor is None or new\_conflicts >= current\_conflicts:

            board = generate\_initial\_board()

            current\_conflicts = count\_conflicts(board)

            restarts += 1

            print(f"Restart #{restarts}")

        else:

            board = best\_neighbor

            current\_conflicts = new\_conflicts

        draw\_board(board)

    print(" hurrrrrrrr.  the Solution founded ")

    while True:

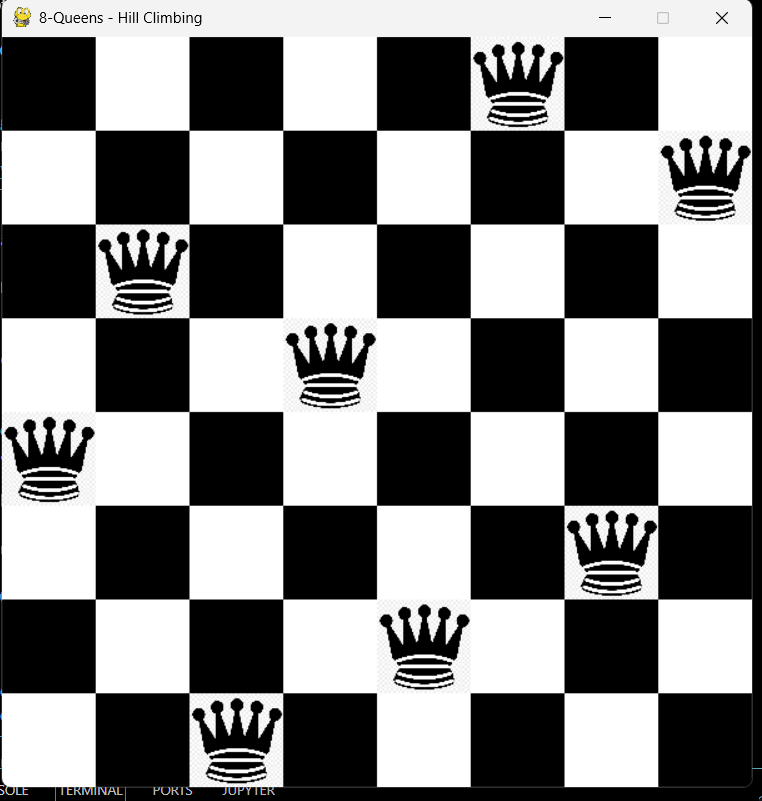
        for event in pygame.event.get():

            if event.type == pygame.QUIT:

                pygame.quit()

                return

hill\_climbing()



**8 Queen (Local Beam)**

import pygame

import numpy as np

import random

import time

pygame.init()

WIDTH, HEIGHT = 600, 600

BOARD\_SIZE = 8

CELL\_SIZE = WIDTH // BOARD\_SIZE

WHITE = (255, 255, 255)

BLACK = (0, 0, 0)

BEAM\_WIDTH = 8

QUEEN\_IMG = pygame.image.load("queen.png")

QUEEN\_IMG = pygame.transform.scale(QUEEN\_IMG, (CELL\_SIZE, CELL\_SIZE))

screen = pygame.display.set\_mode((WIDTH, HEIGHT))

pygame.display.set\_caption("8-Queens problem - ( Local Beam Search ) ")

def count\_conflicts(board):

    conflicts = 0

    for i in range(BOARD\_SIZE):

        for j in range(i+1, BOARD\_SIZE):

            if board[i] == board[j] or abs(i-j) == abs(board[i]-board[j]):

                conflicts += 1

    return conflicts

def generate\_initial\_boards(k):

    return [np.random.randint(0, BOARD\_SIZE, BOARD\_SIZE) for \_ in range(k)]

def get\_all\_neighbors(boards):

    neighbors = []

    for board in boards:

        for row in range(BOARD\_SIZE):

            original\_col = board[row]

            for new\_col in range(BOARD\_SIZE):

                if new\_col != original\_col:

                    new\_board = np.copy(board)

                    new\_board[row] = new\_col

                    neighbors.append(new\_board)

    return neighbors

# function to visulated the board

def draw\_board(board):

    screen.fill(WHITE)

    for row in range(BOARD\_SIZE):

        for col in range(BOARD\_SIZE):

            if (row + col) % 2 == 0:

                pygame.draw.rect(screen, BLACK, (col\*CELL\_SIZE, row\*CELL\_SIZE, CELL\_SIZE, CELL\_SIZE))

            if board[row] == col:

                screen.blit(QUEEN\_IMG, (col\*CELL\_SIZE, row\*CELL\_SIZE))

    pygame.display.update()

    time.sleep(0.2)

# main algorithm

def local\_beam(beam\_width=BEAM\_WIDTH):

    current\_boards = generate\_initial\_boards(beam\_width)

    bst\_board = None

    best\_conflicts = float('inf')

    while best\_conflicts > 0:

        for event in pygame.event.get():

            if event.type == pygame.QUIT:

                pygame.quit()

                return

        all\_neighbors = get\_all\_neighbors(current\_boards)

        if not all\_neighbors:

            current\_boards = generate\_initial\_boards(beam\_width)

            all\_neighbors = get\_all\_neighbors(current\_boards)

        scored\_neighbors = [(count\_conflicts(board), board) for board in all\_neighbors]

        scored\_neighbors.sort(key=lambda x: x[0])

        current\_boards = [board for (score, board) in scored\_neighbors[:beam\_width]]

        #updating the  best solution

        current\_best = min(current\_boards, key=lambda x: count\_conflicts(x))

        current\_best\_conflicts = count\_conflicts(current\_best)

        if current\_best\_conflicts < best\_conflicts:

            best\_conflicts = current\_best\_conflicts

            bst\_board = current\_best

        draw\_board(bst\_board)

    print("hurrrrr , the  Solution founded ")

    while True:

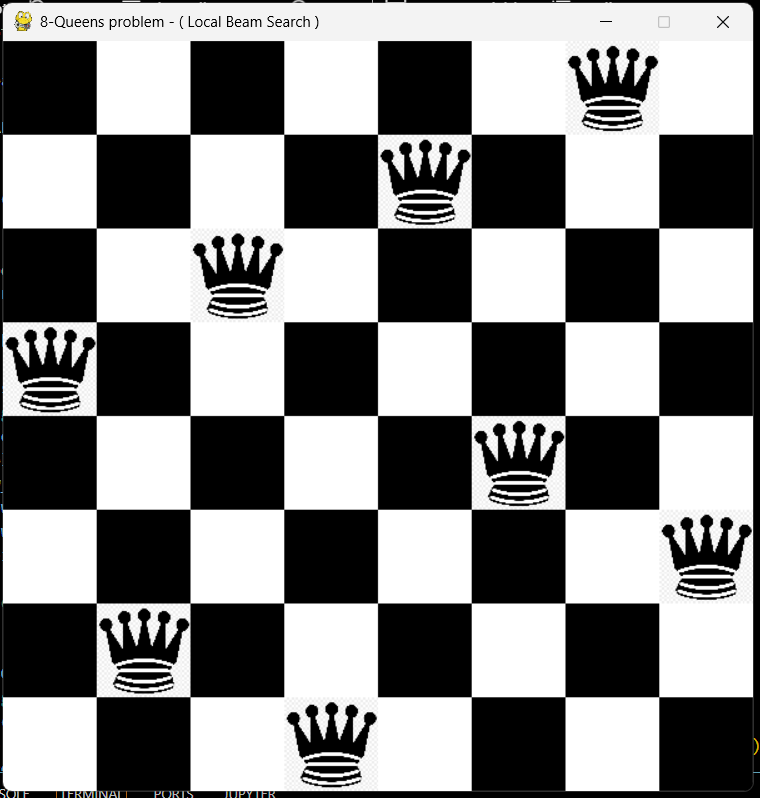
        for event in pygame.event.get():

            if event.type == pygame.QUIT:

                pygame.quit()

                return

local\_beam()



**8 Queen (Stimulate Annealing)**

import pygame

import numpy as np

import random

import math

import time

import sys

pygame.init()

WIDTH, HEIGHT = 600, 600

BOARD\_SIZE = 8

CELL\_SIZE = WIDTH // BOARD\_SIZE

WHITE = (255, 255, 255)

BLACK = (0, 0, 0)

QUEEN\_IMG = pygame.image.load("queen.png")

QUEEN\_IMG = pygame.transform.scale(QUEEN\_IMG, (CELL\_SIZE, CELL\_SIZE))

screen = pygame.display.set\_mode((WIDTH, HEIGHT))

pygame.display.set\_caption("8-Queens problem - ( Local Search Algorithm )")

def count\_conflicts(board):

    conflicts = 0

    for i in range(BOARD\_SIZE):

        for j in range(i + 1, BOARD\_SIZE):

            if board[i] == board[j] or abs(i - j) == abs(board[i] - board[j]):

                conflicts += 1

    return conflicts

def generate\_initial\_board():

    return np.random.randint(0, BOARD\_SIZE, BOARD\_SIZE)

def get\_best\_neighbor(board):

    current\_conflicts = count\_conflicts(board)

    best\_conflicts = current\_conflicts

    best\_neighbors = []

    for row in range(BOARD\_SIZE):

        original\_col = board[row]

        for new\_col in range(BOARD\_SIZE):

            if new\_col == original\_col:

                continue

            neighbor = np.copy(board)

            neighbor[row] = new\_col

            conflicts = count\_conflicts(neighbor)

            if conflicts < best\_conflicts:

                best\_conflicts = conflicts

                best\_neighbors = [np.copy(neighbor)]

            elif conflicts == best\_conflicts:

                best\_neighbors.append(np.copy(neighbor))

    if not best\_neighbors:

        return None, current\_conflicts

    return random.choice(best\_neighbors), best\_conflicts

def draw\_board(board):

    screen.fill(WHITE)

    for row in range(BOARD\_SIZE):

        for col in range(BOARD\_SIZE):

            if (row + col) % 2 == 0:

                pygame.draw.rect(screen, BLACK, (col \* CELL\_SIZE, row \* CELL\_SIZE, CELL\_SIZE, CELL\_SIZE))

            if board[row] == col:

                screen.blit(QUEEN\_IMG, (col \* CELL\_SIZE, row \* CELL\_SIZE))

    pygame.display.update()

    time.sleep(0.5)

def simuulated\_anneling():

    board = generate\_initial\_board()

    crt\_conflict = count\_conflicts(board)

    temp = 1000.0

    cooling\_rate = 0.99

    steps = 0

    running = True

    while running and temp > 0.1 and crt\_conflict > 0:

        for event in pygame.event.get():

            if event.type == pygame.QUIT:

                running = False

                pygame.quit()

                sys.exit()

        row = random.randint(0, BOARD\_SIZE - 1)

        new\_col = random.randint(0, BOARD\_SIZE - 1)

        while new\_col == board[row]:

            new\_col = random.randint(0, BOARD\_SIZE - 1)

        neighbor = np.copy(board)

        neighbor[row] = new\_col

        new\_confloct = count\_conflicts(neighbor)

        delta = new\_confloct - crt\_conflict

        if delta < 0 or random.random() < math.exp(-delta / temp):

            board = neighbor

            crt\_conflict = new\_confloct

        temp \*= cooling\_rate

        steps += 1

        draw\_board(board)

        time.sleep(0.1)

    if crt\_conflict == 0:

        print(" hurrrrr. Solution founded")

    else:

        print("Stopped without finding a solution.")

    draw\_board(board)

    time.sleep(2)

    pygame.quit()

    sys.exit()

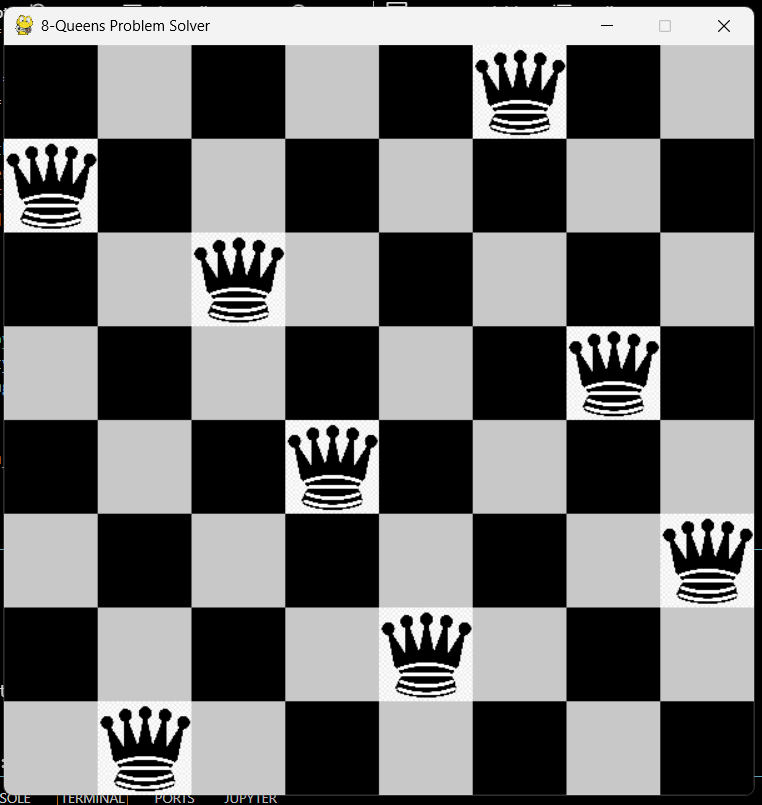
if \_\_name\_\_ == "\_\_main\_\_":

    print("Running Simulated Annealing...")

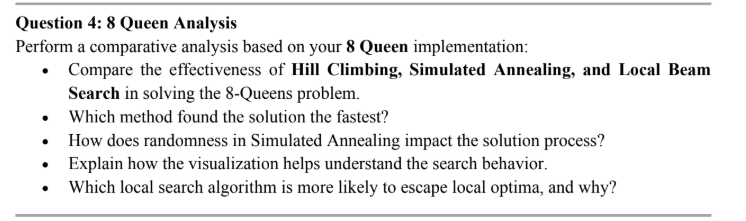
    simuulated\_anneling()

    pygame.quit()

    sys.exit()



**Question 6**



**1.Comparison of Effectiveness in Solving the 8-Queens Problem**

**Hill Climbing:** Greedily selects the neighbor with fewer conflicts, halting at local optima or after 1000 steps. It often stalls at 1-2 conflicts, solving ~20-30% of random starts. **Result: 1-3 conflicts, ~10-50 steps**. Least effective due to local optima traps.

**Simulated Annealing:** Probabilistically accepts worse moves (via exp(-delta / temp)), cooling from **1000 to 1e-5**. It escapes local optima, solving ~50-80% of cases. Result: 0 conflicts likely, ~**100-300 steps**. More effective than Hill Climbing.

**Local Beam Search:** Tracks **k=8 states**, uses randomness and restarts after 50 stagnant steps. Highly effective, solving **~70-90% of runs**. Result: 0 conflicts very likely, ~**50-200 steps**. Most effective due to parallel search and restarts.

**2. Which Method Found the Solution the Fastest?**

Hill Climbing is likely to find a solution the fastest for small problems like the 8-Queens problem because it greedily moves toward the best neighbor at each step.

However, Hill Climbing may get stuck in local optima, so it might not always find a valid solution (0 conflicts).

Simulated Annealing and Local Beam Search are slower due to their exploration mechanisms but are more likely to find a valid solution.

**3. How Does Randomness in Simulated Annealing Impact the Solution Process?**

Randomness in Simulated Annealing (random.randint for neighbor selection and **random.random() < math.exp(-delta / temp**) for acceptance) significantly shapes its behavior:

**Exploration**: Early high temperatures (temp = 1000) allow acceptance of worse moves (e.g., higher conflicts), enabling the algorithm to escape local optima that trap Hill Climbing. For example, moving from 2 conflicts to 4 might be accepted if **exp(-2 / 1000) ≈ 0.998** beats a random value.

**Convergence:** As temp decreases **(cooling\_rate = 0.95),** randomness diminishes, and the algorithm behaves more like Hill Climbing, settling into a solution. Late-game acceptance of worse moves is rare **(e.g., exp(-2 / 0.01) ≈ 0).**

**Impact:** This randomness prevents premature convergence, increasing the chance of finding a global optimum (0 conflicts). However, it can prolong the process if unlucky moves delay progress, unlike the deterministic Hill Climbing or guided Local Beam Search.

*Example: From a state with 3 conflicts, a random neighbor with 5 conflicts might be accepted early, leading to a path that eventually reaches 0, whereas Hill Climbing would reject it outright.*

**4. Explain How the Visualization Helps Understand the Search Behavior**

The visualization (drawing the board and updating it in real-time) helps in:

Observing how the algorithm progresses toward a solution.

Identifying when the algorithm gets stuck in local optima (e.g., Hill Climbing).

Seeing the impact of randomness (e.g., Simulated Annealing) and diversity (e.g., Local Beam Search) on the search process.

Providing a clear understanding of the trade-offs between exploration and exploitation.

**5. Which Local Search Algorithm Is More Likely to Escape Local Optima, and Why?**

**Hill Climbing:** Least likely. It only moves to better neighbors **(best\_conflicts < current\_conflicts),** trapping it in local optima **(e.g., 1 conflict)** with no escape mechanism.

**Simulated Annealing:** Moderately likely. Random acceptance of worse moves (exp(-delta / temp)) allows escape early on, though late-game cooling reduces this ability. Success depends on the cooling schedule and initial temp.

**Local Beam Search:** Most likely. Exploring k=8 states in parallel, combined with randomness in selection **(random.uniform(-0.1, 0.1) and random.random() < 0.3)** and restarts (**restart\_threshold = 50),** ensures it escapes local optima by diversifying the search and resetting when stuck.