```
import math
import time

board = [' ' for _ in range(9)]

def show_board():
    for i in range(3):
        print('| ' + ' | '.join(board[i*3:(i+1)*3]) + ' |')
    print("\n")

def check_winner(board, player):
    win_patterns = [
        (0, 1, 2), (3, 4, 5), (6, 7, 8),
        (0, 3, 6), (1, 4, 7), (2, 5, 8),
        (0, 4, 8), (2, 4, 6)
    ]
    return any(board[a] == board[b] == board[c] == player for a, b, c in win_patterns)

def board_full(board):
    return ' ' not in board

def minimax(board, depth, maximizing):
    if check_winner(board, 'Osaid'):
        return 1
    elif check_winner(board, 'Abubakar'):
        return -1
    elif board_full(board):
        return 0

    if maximizing:
        max_eval = -math.inf
        for i in range(9):
            if board[i] == ' ':
                board[i] = 'Osaid'
                eval = minimax(board, depth + 1, False)
                board[i] = ' '
                max_eval = max(max_eval, eval)
        return max_eval
    else:
        min_eval = math.inf
        for i in range(9):
            if board[i] == ' ':
                board[i] = 'Abubakar'
                eval = minimax(board, depth + 1, True)
                board[i] = ' '
                min_eval = min(min_eval, eval)
        return min_eval

def minimax_alpha_beta(board, depth, alpha, beta, maximizing):
    if check_winner(board, 'Osaid'):
        return 1
    elif check_winner(board, 'Abubakar'):
        return -1
    elif board_full(board):
        return 0

    if maximizing:
        max_eval = -math.inf
        for i in range(9):
            if board[i] == ' ':
                board[i] = 'Osaid'
                eval = minimax_alpha_beta(board, depth + 1, alpha, beta, False)
                board[i] = ' '
                max_eval = max(max_eval, eval)
                alpha = max(alpha, eval)
                if beta <= alpha:
                    break
        return max_eval
    else:
        min_eval = math.inf
        for i in range(9):
            if board[i] == ' ':
                board[i] = 'Abubakar'
                eval = minimax_alpha_beta(board, depth + 1, alpha, beta, True)
                board[i] = ' '
                min_eval = min(min_eval, eval)
                beta = min(beta, eval)
                if beta <= alpha:
                    break
        return min_eval

def best_move(algorithm='minimax'):
    best_score = -math.inf
    move = -1
    for i in range(9):
        if board[i] == ' ':
            board[i] = 'Osaid'
            if algorithm == 'alpha_beta':
                score = minimax_alpha_beta(board, 0, -math.inf, math.inf, False)
            else:
                score = minimax(board, 0, False)
            board[i] = ' '
            if score > best_score:
                best_score = score
                move = i
    return move

def play_game(algorithm='minimax'):
    print(f"Starting a game against the AI using the {algorithm} algorithm.")
    while True:
        show_board()
        x_move = int(input("Abubakar, enter your move (0-8): "))
        if board[x_move] == ' ':
            board[x_move] = 'Abubakar'
        else:
            print("Invalid move, try again.")
            continue

        if check_winner(board, 'Abubakar'):
            show_board()
            print("Congratulations, Abubakar wins!")
            break
        elif board_full(board):
            show_board()
            print("It's a draw!")
            break

        o_move = best_move(algorithm)
        board[o_move] = 'Osaid'

        if check_winner(board, 'Osaid'):
            show_board()
            print("Osaid (AI) wins! Better luck next time.")
            break
        elif board_full(board):
            show_board()
            print("It's a draw!")
            break
```

Test cases

```
import time

# Initialize test results
results = {
    "minimax": {"nodes_explored": 0, "time": 0},
    "alpha_beta": {"nodes_explored": 0, "time": 0}
}

# Track nodes explored
nodes_explored_minimax = 0
nodes_explored_alpha_beta = 0

# Wrap Minimax and Alpha-Beta to track nodes
def minimax_tracking(board, depth, maximizing):
    global nodes_explored_minimax
    nodes_explored_minimax += 1
    return minimax(board, depth, maximizing)

def minimax_alpha_beta_tracking(board, depth, alpha, beta, maximizing):
    global nodes_explored_alpha_beta
```

```
        nodes_explored_alpha_beta += 1
        return minimax_alpha_beta(board, depth, alpha, beta, maximizing)

# Function to test both algorithms and compare results
def test_algorithms():
    # Reset board for each test
    global board, nodes_explored_minimax, nodes_explored_alpha_beta
    board = [' ' for _ in range(9)]

    # Test Minimax
    nodes_explored_minimax = 0
    start_time = time.time()
    best_move(algorithm='minimax')
    end_time = time.time()
    results["minimax"]["nodes_explored"] = nodes_explored_minimax
    results["minimax"]["time"] = end_time - start_time

    # Test Alpha-Beta Pruning
    nodes_explored_alpha_beta = 0
    start_time = time.time()
    best_move(algorithm='alpha_beta')
    end_time = time.time()
    results["alpha_beta"]["nodes_explored"] = nodes_explored_alpha_beta
    results["alpha_beta"]["time"] = end_time - start_time

    # Print results
    print("Performance Comparison:")
    for algorithm, data in results.items():
        print(f"\n{algorithm.capitalize()} Algorithm:")
        print(f"  Nodes Explored: {data['nodes_explored']}")
        print(f"  Time Taken: {data['time']} seconds")

# Run the test
test_algorithms()
```

```
Performance Comparison:

Minimax Algorithm:
  Nodes Explored: 0
  Time Taken: 2.0772979259490967 seconds

Alpha_beta Algorithm:
  Nodes Explored: 0
  Time Taken: 0.12005233764648438 seconds
```