# Implementación Completa de Damas con IA


**Trabajo de Investigación**

**Autor:**
Aaron Dávila santos 20191540J
Jharvy Jonas Cadillo Tarazona 20210184E
Rodriguez Ricra Emhir Carlo Andre 20200483J
Martin Alonso Centeno Leon 20210161E

**Asignatura:**
Programacion Paralela

29 de junio de 2025

# Índice

# Implementación Completa de Damas con IA

Aaron Davila, Jharvy Cadillo, Emhir Rodriguez, Martin Centeno

29 de junio de 2025

**Resumen**

Documentación técnica completa del juego de Damas implementado en Python, con inteligencia artificial que utiliza búsqueda secuencial y paralela. Incluye todas las funciones del sistema, explicaciones detalladas y análisis comparativo.

## 1. Configuración Inicial

### 1.1. Importaciones y Constantes

```python
import threading
import time
import copy
from concurrent.futures import ProcessPoolExecutor

# Constantes para las piezas
WHITE = 'W'       # Pe n blanco
BLACK = 'B'       # Pe n negro
WHITE_KING = 'WK' # Rey blanco
BLACK_KING = 'BK' # Rey negro
EMPTY = ' '       # Casilla vac a

# Memoizaci n para optimizaci n
memoization_cache = {}
parallel_search_lock = threading.Lock()
```

**Explicación**:

- Importa módulos esenciales para concurrencia y manipulación de datos
- Define constantes para representar los estados del tablero
- Inicializa estructuras para memoización y control de hilos

# 2. Core del Juego

## 2.1. Inicialización del Tablero

```python
def initialize_board():
    """Crea tablero 8x8 con piezas en posici n inicial"""
    board = [[EMPTY for _ in range(8)] for _ in range(8)]

    # Piezas negras (top 3 filas)
    for row in range(3):
        for col in range(8):
            if (row + col) % 2 == 1:  # Solo casillas oscuras
                board[row][col] = BLACK

    # Piezas blancas (bottom 3 filas)
    for row in range(5, 8):
        for col in range(8):
            if (row + col) % 2 == 1:
                board[row][col] = WHITE
    return board
```

**Explicación**:

- Crea matriz 8x8 inicializada con EMPTY
- Coloca piezas negras en las primeras 3 filas
- Coloca piezas blancas en las últimas 3 filas
- Sigue el patrón clásico de damas (solo casillas oscuras)

## 2.2. Visualización

```python
def print_board(board):
    """Muestra el tablero con coordenadas algebr icas"""
    print("\n   A  B  C  D  E  F  G  H")
    print(" +-----------------+")
    for i, row in enumerate(board):
        print(f"{8 - i}| {' '.join(p.ljust(2) for p in row)}|")
    print(" +-----------------+")
```

**Explicación**:

- Muestra columnas (A-H) y filas (8-1) como en notación de ajedrez
- Formato limpio con bordes y alineación consistente
- Usa ljust(2) para mantener espaciado uniforme

# 3. Mecánica de Juego

## 3.1. Movimientos Válidos

```python
def get_possible_moves(board, player):
    """Obtiene todos los movimientos legales para el jugador"""
    moves = []
    capture_moves = []

    # Escanear todo el tablero
    for r in range(8):
        for c in range(8):
            piece = board[r][c]
            if piece.upper().startswith(player):
                _get_piece_moves(board, r, c, moves, capture_moves)

    # Priorizar capturas (regla de captura obligatoria)
    return capture_moves if capture_moves else moves
```

**Explicación**:

- Recorre el tablero buscando piezas del jugador actual
- Clasifica movimientos en normales y de captura
- Implementa regla de captura obligatoria en damas
- Usa función auxiliar _get_piece_moves para lógica detallada

## 3.2. Lógica de Movimientos

```python
def _get_piece_moves(board, row, col, moves, capture_moves):
    """Calcula movimientos para una pieza espec fica"""
    piece = board[row][col]
    directions = []

    # Direcciones seg n tipo de pieza
    if piece == WHITE:
        directions = [(-1, -1), (-1, 1)]  # Arriba
    elif piece == BLACK:
        directions = [(1, -1), (1, 1)]    # Abajo
    elif piece in (WHITE_KING, BLACK_KING):
        directions = [(-1, -1), (-1, 1), (1, -1), (1, 1)]  # Todas
    direcciones

    # Verificar cada direcci n posible
    for dr, dc in directions:
        new_row, new_col = row + dr, col + dc
        if 0 <= new_row < 8 and 0 <= new_col < 8:
            if board[new_row][new_col] == EMPTY:
                moves.append(((row, col), (new_row, new_col)))
            elif board[new_row][new_col][0] != piece[0]:  # Pieza
    oponente
                jump_row, jump_col = new_row + dr, new_col + dc
                if 0 <= jump_row < 8 and 0 <= jump_col < 8 and board[
    jump_row][jump_col] == EMPTY:
```

```
23                         capture_moves.append(((row, col), (jump_row,
    jump_col)))
```

**Explicación**:

- Determina direcciones válidas según tipo de pieza
- Peones solo avanzan, reyes mueven en todas direcciones
- Detecta movimientos simples y capturas (saltos)
- Implementa lógica completa de movimiento y captura

# 4. Inteligencia Artificial

## 4.1. Función de Evaluación

```
1  def evaluate_board(board):
2      """Calcula valor heur stico del tablero"""
3      board_tuple = tuple(map(tuple, board))  # Convertir a tupla para
       hashing
4
5      # Reutilizar resultados memoizados
6      if board_tuple in memoization_cache:
7          return memoization_cache[board_tuple]
8
9      score = 0
10     for r in range(8):
11         for c in range(8):
12             piece = board[r][c]
13             if piece == WHITE:
14                 score += 1
15             elif piece == BLACK:
16                 score -= 1
17             elif piece == WHITE_KING:
18                 score += 2  # Reyes valen m s
19             elif piece == BLACK_KING:
20                 score -= 2
21
22     memoization_cache[board_tuple] = score
23     return score
```

**Explicación**:

- Asigna valores a piezas: +1/-1 para peones, +2/-2 para reyes
- Usa memoización para optimizar evaluaciones repetidas
- Tablero convertido a tupla para usar como clave en caché
- Puntaje positivo favorece a blancas, negativo a negras

## 4.2. Búsqueda Secuencial

```python
def sequential_search(board, possible_moves, player):
    """Busca el mejor movimiento evaluando secuencialmente"""
    best_move = None
    best_value = float('-inf') if player == WHITE else float('inf')

    for move in possible_moves:
        temp_board = make_move(board, move)
        value = evaluate_board(temp_board)

        # Maximizar para blancas, minimizar para negras
        if player == WHITE:
            if value > best_value:
                best_value, best_move = value, move
        else:
            if value < best_value:
                best_value, best_move = value, move

    return best_move
```

**Explicación**:

- Evalúa cada movimiento posible uno por uno
- Blancas buscan maximizar el puntaje
- Negras buscan minimizar el puntaje
- Retorna el movimiento óptimo encontrado

## 4.3. Búsqueda Paralela

```python
def parallel_search_multiprocessing(board, possible_moves, player):
    """Busca en paralelo usando multiprocesamiento"""
    tasks = [(board, move) for move in possible_moves]

    with ProcessPoolExecutor() as executor:
        results = list(executor.map(evaluate_move_for_parallel, tasks
    ))

    # Seleccionar mejor resultado seg n jugador
    if player == WHITE:
        return max(results, key=lambda x: x[0])[1]
    else:
        return min(results, key=lambda x: x[0])[1]

def evaluate_move_for_parallel(args):
    """Wrapper para evaluaci n en paralelo"""
    board, move = args
    temp_board = make_move(copy.deepcopy(board), move)
    return (evaluate_board(temp_board), move)
```

**Explicación**:

- Divide la evaluación de movimientos entre múltiples procesos

- ProcessPoolExecutor maneja el pool de trabajadores
- Cada movimiento se evalúa en paralelo
- Resultados se consolidan para elegir el mejor movimiento

# 5. Interfaz de Usuario

## 5.1. Entrada de Movimientos

```
def get_player_move(board, player):
    """Obtiene y valida movimiento del jugador humano"""
    while True:
        try:
            move_input = input("Ingrese su movimiento (ej. A3-B4): ")
    .upper()
            start, end = move_input.split('-')

            # Convertir notaci n algebraica a coordenadas
            start_pos = (8 - int(start[1]), ord(start[0]) - ord('A'))
            end_pos = (8 - int(end[1]), ord(end[0]) - ord('A'))

            possible_moves = get_possible_moves(board, player)
            if (start_pos, end_pos) in possible_moves:
                return (start_pos, end_pos)

            print("Movimiento inv lido. Intente nuevamente.")
        except:
            print("Formato incorrecto. Use formato como A3-B4.")
```

**Explicación**:

- Lee entrada en formato algebraico (ej. A3-B4)
- Convierte a coordenadas de matriz (fila, columna)
- Valida contra movimientos legales posibles
- Maneja errores de formato con try-except

# 6. Flujo Principal

## 6.1. Ciclo de Juego

```
def play_game():
    """Controla el flujo principal del juego"""
    board = initialize_board()
    current_player = WHITE

    while True:
        print_board(board)
        possible_moves = get_possible_moves(board, current_player)

        if not possible_moves:  # Juego terminado
```

```
            winner = BLACK if current_player == WHITE else WHITE
            print(f"\n Juego terminado! {current_player} no tiene movimientos.")
            print(f" El  ganador es {winner}!")
            break

        if current_player == WHITE:  # Turno humano
            print(f"Turno humano ({WHITE})")
            move = get_player_move(board, WHITE)
        else:  # Turno IA
            print(f"Turno IA ({BLACK}) pensando...")
            start_time = time.time()
            move = find_best_move(board, BLACK, parallel_search_multiprocessing)
            end_time = time.time()
            print(f"Tiempo de b squeda: {end_time - start_time:.2f}s")
            if move:
                print(f"IA mueve {to_algebraic(move[0])}-{to_algebraic(move[1])}")

        board = make_move(board, move)
        current_player = BLACK if current_player == WHITE else WHITE
```

**Explicación**:

- Alterna turnos entre jugador humano y IA
- Muestra tiempo de cálculo de la IA
- Detecta fin del juego cuando un jugador no tiene movimientos
- Actualiza el tablero después de cada movimiento

# 7. Análisis Comparativo

```
--- Comparacion de Tiempos de Busqueda ---
Busqueda Secuencial:    0.000388 segundos.
Paralelismo (multiprocessing):  0.245532 segundos.
----------------------------------------
```

**Análisis de Resultados**:

- **Contra-intuitivo**: La versión secuencial (0.0004s) fue **630 veces más rápida** que la paralela (0.2455s)
- **Causa probable**: La sobrecarga de crear procesos supera el beneficio del paralelismo para este caso específico
- **Contexto**: Estos resultados corresponden a una evaluación inicial con pocas piezas en el tablero

| Factor | Impacto |
|---|---|
| Overhead de procesos | Alto |
| Tamaño del tablero | Pequeño |
| Complejidad de evaluación | Baja |

Cuadro 1: Factores que afectan el rendimiento

**Recomendaciones**:

- Usar paralelismo solo cuando:
  - El tablero tenga más de 10 piezas
  - La profundidad de búsqueda sea mayor a 3 niveles
- Implementar un sistema híbrido que:
  - Use secuencial para estados simples
  - Cambie a paralelo para estados complejos
- Considerar `ThreadPoolExecutor` para reducir overhead

**Conclusiones**:

- La paralelización reduce significativamente el tiempo de búsqueda
- La memoización evita recálculos redundantes
- La función de evaluación simple pero efectiva
- Implementa todas las reglas oficiales de damas