Jugador Virtual Hex

Melani Forsythe Matos
Carrera: Ciencias de la Computación
Grupo: C312

10 de abril de 2025

1. Objetivo

Desarrollar un jugador virtual competitivo para el juego de Hex capaz de enfrentar oponentes avanzados, tomando decisiones estratégicas de forma eficiente tanto en tableros pequeños como grandes.

2. Estrategia General

El jugador utiliza una combinación de:

- Algoritmo **Minimax** con poda **alfa-beta** para tomar decisiones.
- **Evaluación heurística** adaptativa.
- **Reconocimiento de patrones estratégicos**: puentes, doble amenaza.
- **Paralelización** del cálculo usando múltiples núcleos.
- **Caching** de estados con tabla de transposición.

3. Implementación Paso a Paso

3.1. 1. Método principal: `play()`

Este método decide la jugada a realizar:

```
def play(self, tablero):
    movimientos = tablero.get_possible_moves()
    profundidad = self.calcular_profundidad(len(movimientos))
    args = [(self.jugador_id, tablero_flat, tam, mov, profundidad) for
        mov in movimientos]
    resultados = executor.map(evaluar_movimiento_global, args)
    return max(resultados)[1]
```

El método evalúa todas las jugadas posibles en paralelo y selecciona la mejor.

3.2. 2. Algoritmo `minimax()` con poda alfa-beta

```
def minimax(self, tablero, profundidad, maximizador, alfa, beta):
    if profundidad == 0 or tablero.check_connection(...):
        return self.evaluar(tablero)
    if maximizador:
        ...
    else:
        ...
```

Busca la mejor jugada minimizando el error y evitando explorar ramas inútiles.

3.3. 3. Evaluación heurística

La función `evaluar()` es fundamental para el rendimiento e inteligencia del jugador virtual. Esta función analiza el estado actual del tablero y devuelve un valor numérico que indica qué tan buena es esa posición para el jugador actual. Cuanto mayor es el valor, más favorable es.

A continuación se describen cada uno de los componentes que conforman la heurística:

- **Distancia mínima a la victoria**: Se calcula usando una versión simplificada de Dijkstra. Se trata de determinar la distancia más corta entre los dos bordes que el jugador necesita conectar. Esta distancia se penaliza o bonifica dependiendo de si es el jugador o su oponente.

```
  def distancia_conexion(jugador_id):
      dist = [[inf] * tam for _ in range(tam)]
      return min_distancia
```

- **Amenaza del oponente**: Si el oponente está a una sola jugada de ganar, se devuelve un valor muy negativo. Esto fuerza al jugador a bloquear esa amenaza en lugar de hacer otra jugada.

```
  if distancia_oponente <= 1:
      return -1000
```

- **Control del centro**: Las casillas cerca del centro del tablero tienen más conexiones posibles, por lo tanto, se bonifican.

```
  for fila in range(tam):
      for col in range(tam):
          if tablero[fila][col] == jugador:
              score += max(0, 5 - distancia_al_centro)
```

- **Puentes seguros**: El jugador recibe puntos extra si hay dos fichas propias conectadas indirectamente con un hueco en el medio (estructura típica de Hex).

```
  if tablero[nf][nc] == jugador and tablero[mf][mc] == 0:
      bonus += 1
```

- **Amenazas dobles**: Si una jugada conecta dos grupos propios separados, se cuenta como una doble amenaza, lo cual complica la defensa del oponente.

```
if conexiones_vecinas >= 2:
    doble_amenaza += 1
```

- **Adaptación a la etapa de juego**:
  - En la **apertura**, se favorece el control del centro y el desarrollo.
  - En el **medio juego**, se balancea entre atacar y defender.
  - En el **final**, se prioriza la conexión directa y bloquear.

Función completa de evaluación:

```
return (d_oponente - d_propio) * peso +
       centro_score +
       puente_bonus * 3 +
       doble_amenaza * 5 +
       amenaza_oponente
```

Donde:

- d_{propio} y $d_{oponente}$ son distancias mínimas a la conexión (Dijkstra).
- `puente_bonus`: detecta estructuras tipo `X _ X`.
- `doble_amenaza`: jugadas que conectan múltiples grupos.
- `amenaza_oponente`: penaliza si el oponente está a una jugada de ganar.

3.4. 4. Caching con tabla de transposición

```
key = hash(str(tablero.tablero))
if key in transposition_table:
    return transposition_table[key]
```

Se evita evaluar múltiples veces la misma posición del tablero.

4. Características Avanzadas

- **Puentes seguros**: conexión indirecta que garantiza continuidad.
- **Amenazas dobles**: forzan al oponente a dividir su defensa.
- **Control del centro**: bonificación en las primeras jugadas.
- **Etapas del juego**: evaluación adaptativa (inicio, medio, final).

5. Igualdad de Condiciones

Se eliminó toda aleatoriedad, garantizando que los dos jugadores sigan exactamente la misma lógica. Ambos usan la misma IA y heurística, permitiendo comparaciones justas.

6. Experiencia y Ajustes

Durante las pruebas se observó:

- Jugador 2 parecía tener ventaja debido a las respuestas inmediatas: esto se equilibró mejorando la apertura del jugador 1 (jugada central).
- Jugadas repetitivas entre jugadores: solucionado añadiendo evaluación contextual (dobles amenazas, puentes).
- Se identificó la necesidad de caching y se optimizó.