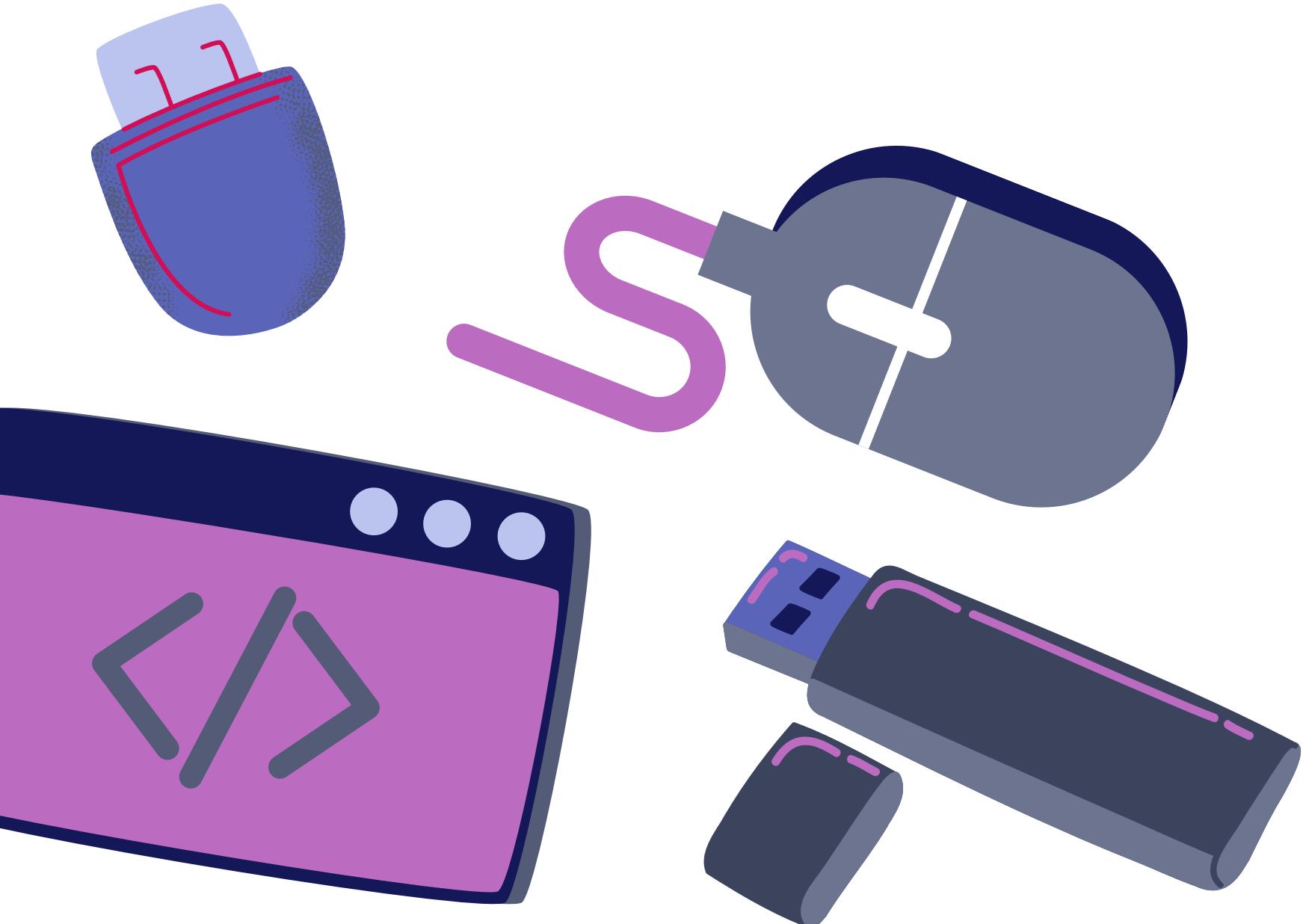


Equipo: 3

# ALGORITMOS DE BÚSQUEDA ENTRE ADVERSARIOS

# ÍNDICE



---

**01. Introducción**

---

**02. Algoritmo de búsqueda**

---

**03. Algoritmo Min-Max**

---

**04. Poda Alfa-Beta**

---

**05. Decisiones en Tiempo Real**

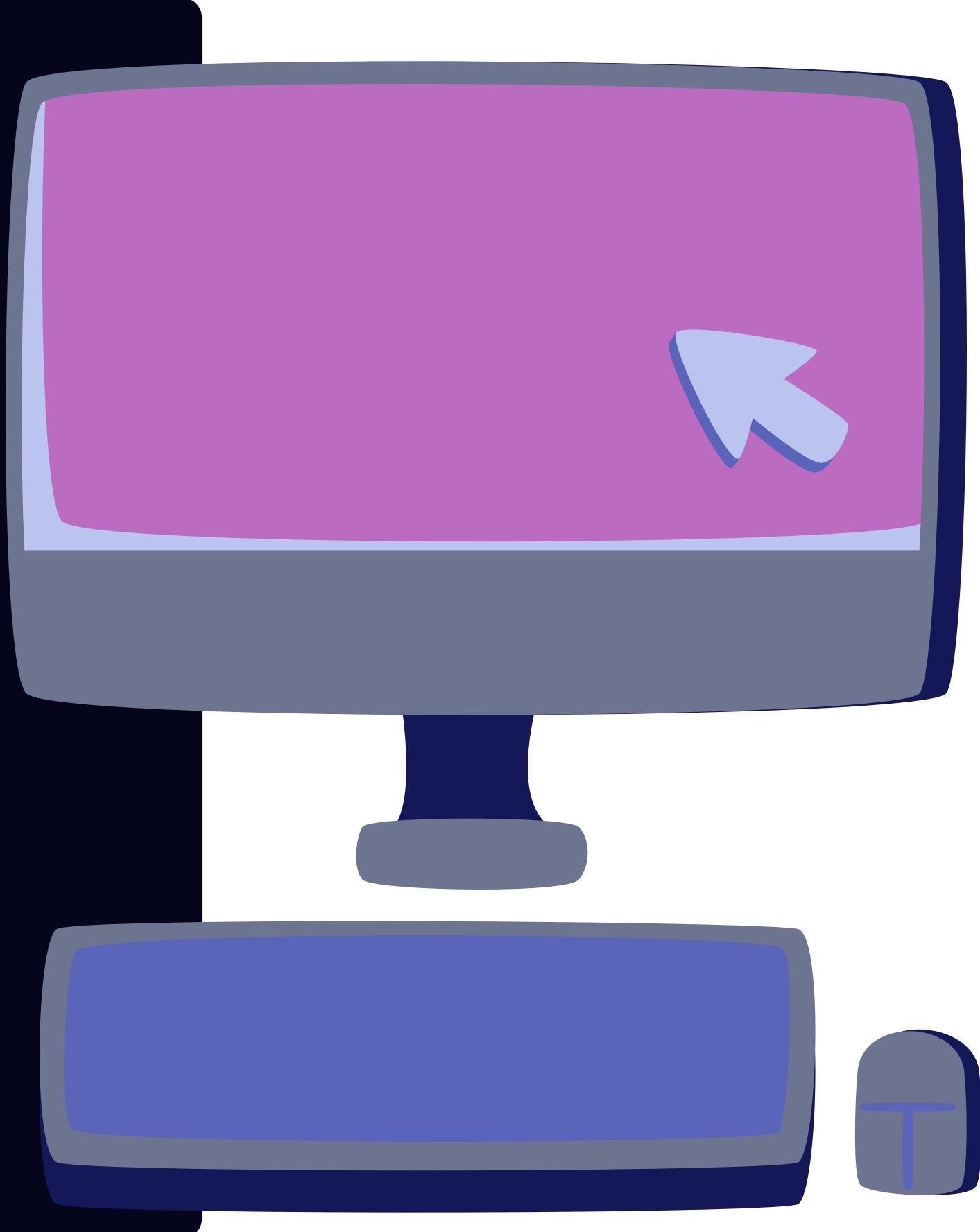
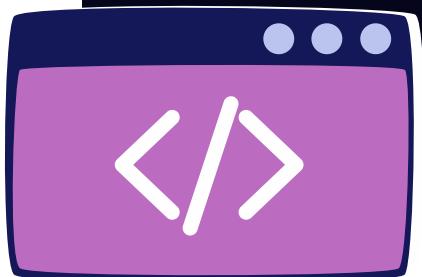
---

**06. Juegos de Posibilidad**

---

# Introducción

- Los algoritmos de búsqueda son fundamentales en inteligencia artificial.
- Permiten la toma de decisiones estratégicas en entornos competitivos.
- Ejemplos: Ajedrez, Damas, Go, donde los jugadores compiten directamente.



# ¿QUÉ ES UN ALGORITMO DE BÚSQUEDA?

- **Definición:** Un algoritmo que evalúa posibles movimientos para encontrar la mejor opción.
- **Objetivo:** Maximizar las posibilidades de éxito mientras se minimizan las del oponente.

# JUEGOS

- **Suma cero**
- **Deterministas**
- Secuenciales y simultaneos

# EJEMPLO GATO

```
[1]: def print_board(board):
    for row in board:
        print(" | ".join(row))
    print("-" * 5)

[2]: def check_winner(board):
    # Check rows, columns, and diagonals for a winner
    for row in board:
        if row[0] == row[1] == row[2] != " ":
            return row[0]
    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] != " ":
            return board[0][col]
        if board[0][0] == board[1][1] == board[2][2] != " ":
            return board[0][0]
        if board[0][2] == board[1][1] == board[2][0] != " ":
            return board[0][2]
    return None

[9]: def tic_tac_toe():
    board = [[" " for _ in range(3)] for _ in range(3)]
    current_player = "X"

    for turn in range(9):
        print_board(board)
        print(f"Turno del jugador {current_player}. Introduce fila y columna (0, 1 o 2):")
        row, col = map(int, input().split())

        if board[row][col] == " ":
            board[row][col] = current_player
            winner = check_winner(board)
            if winner:
                print_board(board)
                print(f"¡El jugador {winner} ha ganado!")
                return
```

```
    print(f"¡El jugador {winner} ha ganado!")
    return
    current_player = "O" if current_player == "X" else "X"
else:
    print("Esa posición ya está ocupada. Intenta de nuevo.")
    turn -= 1

print_board(board)
print("¡Es un empate!")
```

```
: tic_tac_toe()
```

```
| |
-----
```

```
| |
-----
```

```
| |
-----
```

```
X | |
-----
```

```
| |
-----
```

```
| |
-----
```

```
Turno del jugador X. Introduce fila y columna (0, 1 o 2):
```

```
X | |
-----
```

```
| 0 |
-----
```

```
| |
-----
```

```
Turno del jugador O. Introduce fila y columna (0, 1 o 2):
```

```
X | |
-----
```

```
| 0 |
-----
```

```
| |
-----
```

```
Turno del jugador X. Introduce fila y columna (0, 1 o 2):
```

```
X | |
-----
```

```
| 0 |
-----
```

```
| |
-----
```

# EJEMPLO GATO

```
[10]: tic_tac_toe()

| |
-----
| |
-----
| |

Turno del jugador X. Introduce fila y columna (0, 1 o 2):
X | |
-----
| |
-----
| |

Turno del jugador O. Introduce fila y columna (0, 1 o 2):
X | |
-----
| 0 |
-----
| |
-----
Turno del jugador X. Introduce fila y columna (0, 1 o 2):
X | |
-----
| 0 |
-----
| | X

Turno del jugador O. Introduce fila y columna (0, 1 o 2):
X | 0 |
-----
| |
-----
| | X

Turno del jugador X. Introduce fila y columna (0, 1 o 2):
X | 0 |
-----
| 0 |
-----
| | X

Turno del jugador O. Introduce fila y columna (0, 1 o 2):
X | 0 |
-----
```

```
| 0 |
-----
|   | X
-----
Turno del jugador O. Introduce fila y columna (0, 1 o 2):
X | 0 |
-----
| 0 |
-----
|   | X
-----
Turno del jugador X. Introduce fila y columna (0, 1 o 2):
X | 0 |
-----
X | 0 |
-----
|   | X
-----
Turno del jugador O. Introduce fila y columna (0, 1 o 2):
Esa posición ya está ocupada. Intenta de nuevo.
X | 0 |
-----
X | 0 |
-----
|   | X
-----
Turno del jugador O. Introduce fila y columna (0, 1 o 2):
X | 0 |
-----
X | 0 |
-----
| 0 | X
-----
¡El jugador O ha ganado!
```

# DIFERENCIAS

## JUEGOS DE ADVERSARIOS

- Objetivo: Cada jugador busca maximizar su propia ganancia a expensas del oponente.
- Interacción: Los jugadores son rivales; las decisiones de uno afectan directamente a los otros.
- Ejemplos: Ajedrez, Damas, Póker.
- Estrategia: Enfocada en anticipar y contrarrestar las acciones del oponente.

## JUEGOS DE COOPERACIÓN

- Objetivo: Los jugadores trabajan juntos hacia un objetivo común, maximizando beneficios para todos.
- Interacción: Se fomenta la colaboración; las decisiones se toman en función del bienestar grupal.
- Ejemplos: Juegos de rol, proyectos en grupo, algunos videojuegos cooperativos.
- Estrategia: Enfocada en coordinar acciones y lograr metas comunes.



# ¿Cómo funcionan?



- Los algoritmos como Minimax y Alpha-Beta Pruning son fundamentales.
- Evalúan todas las opciones posibles en un "árbol de decisiones" para predecir y responder a movimientos de un oponente.

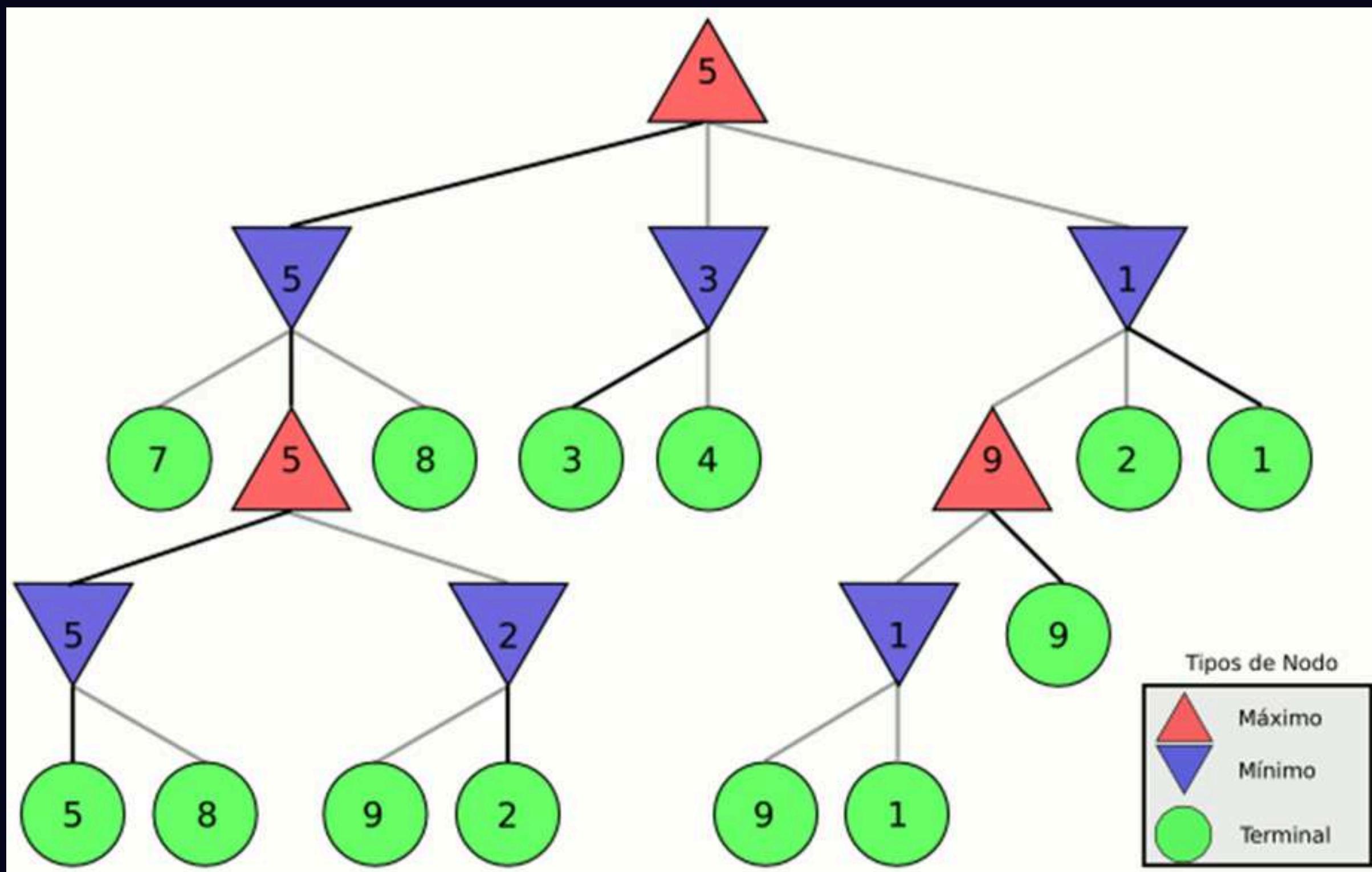


## EJEMPLO

En un juego de ajedrez, la IA intenta prever las jugadas del jugador y responde con la mejor jugada posible.

# ALGORITMO MIN-MAX

# ¿Qué es?



El algoritmo minimax es una técnica utilizada principalmente en teoría de juegos y en inteligencia artificial para tomar decisiones en entornos de adversarios, como el ajedrez o el juego del gato. Su objetivo es minimizar la posible pérdida en el peor de los casos y maximizar la ganancia en el mejor. Funciona evaluando todos los movimientos posibles en un árbol de decisiones hasta una profundidad definida y eligiendo la opción óptima que maximice la probabilidad de ganar o minimizar la posibilidad de perder.

# ¿Cómo funciona?

• **Construcción del árbol de decisiones:** El algoritmo genera un árbol que representa todos los movimientos posibles desde el estado actual del juego, donde cada nodo es un estado alcanzable.

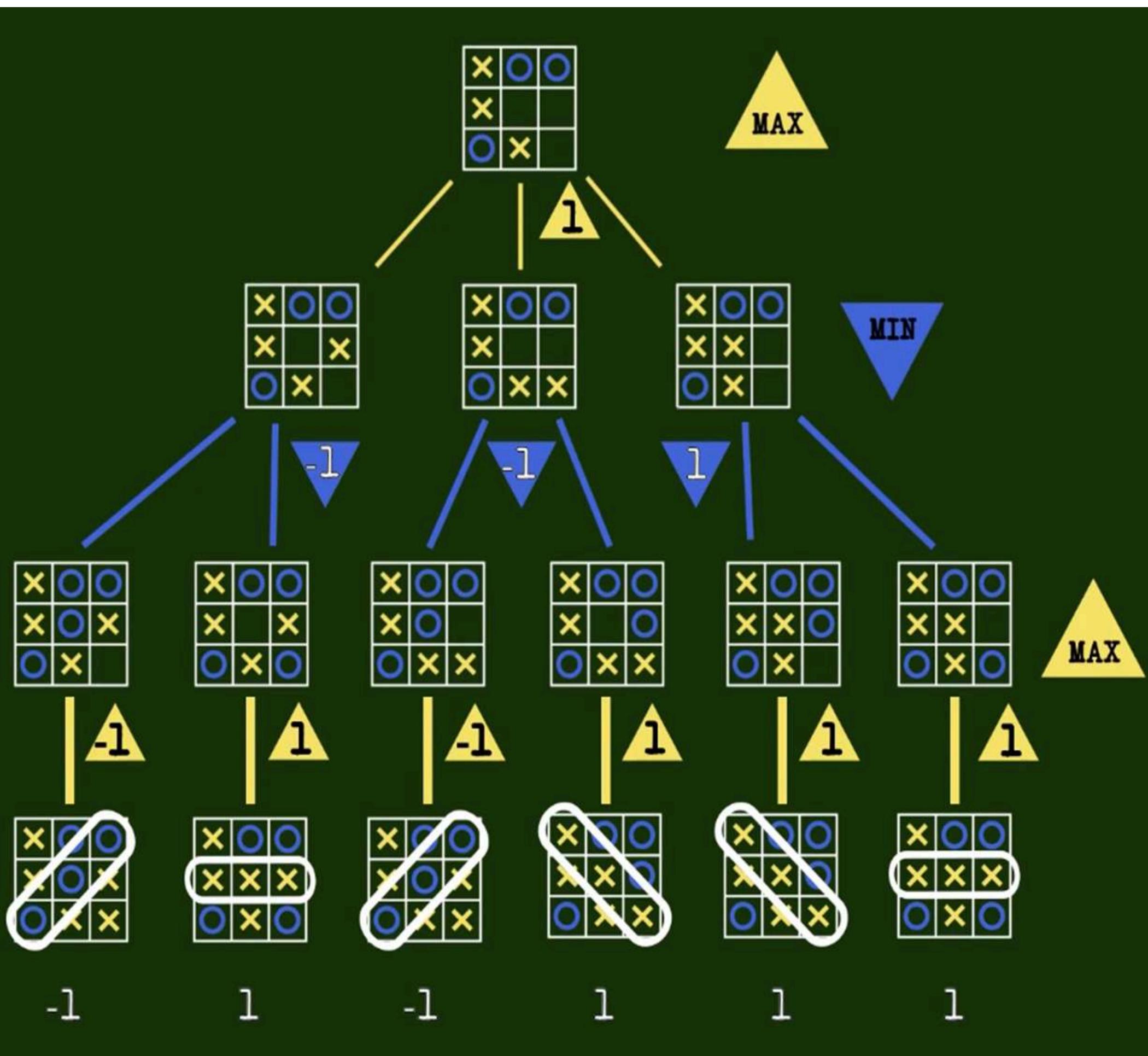
## • Turnos de jugador y adversario:

Cada nivel del árbol corresponde a un turno de uno de los jugadores. Si es el turno del jugador que usa minimax, se elige el movimiento que maximiza su puntaje; si es el turno del oponente, se elige el movimiento que minimiza el puntaje del jugador.

• **Evaluación de nodos terminales:** Cuando se alcanza un nodo terminal, se evalúa y asigna un valor que representa su utilidad para el jugador.

Este valor se establece de acuerdo con el resultado del juego en ese nodo: una victoria para el jugador, una derrota, o un empate, con valores numéricos que usualmente son 1, -1 y 0 respectivamente.

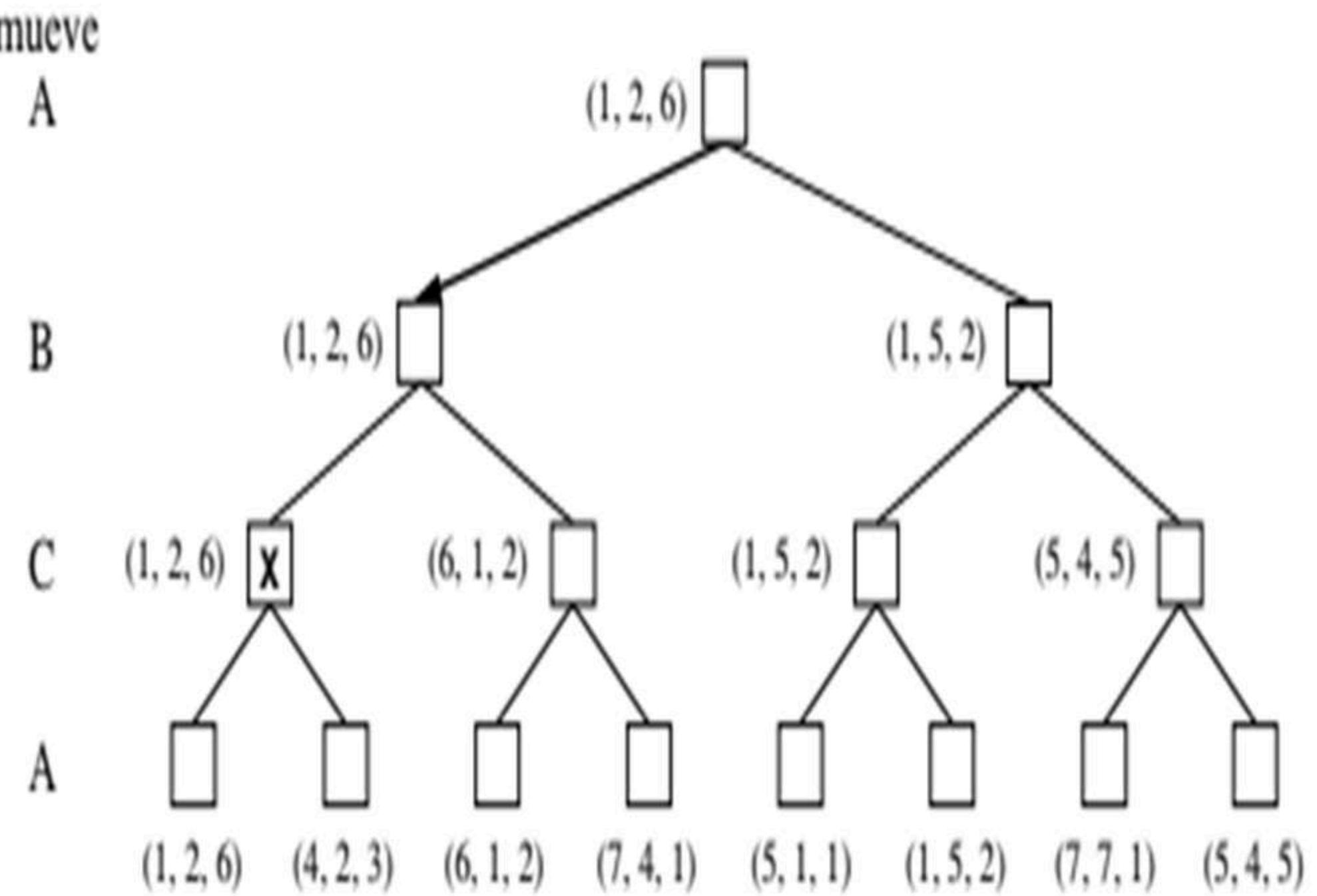
• **Propagación de valores:** Se retrocede en el árbol propagando los valores hacia el nodo raíz. El algoritmo minimax selecciona la acción con el valor óptimo en el nodo inicial.



# JUEGOS MULTI-JUGADOR

Primero, tenemos que sustituir el valor para cada nodo con un vector de valores. Por ejemplo, en un juego de tres jugadores con jugadores A, B y C, un vector  $(VA, VB, VC)$  asociado con cada nodo. Para los estados terminales, este vector dará la utilidad del estado desde el punto de vista de cada jugador.

- **Turnos de múltiples jugadores:** En lugar de alternar solo entre un jugador maximizador y uno minimizador, el árbol incorpora nodos de cada jugador. Esto significa que, en cada turno, el jugador actual tomará la mejor decisión para sí mismo, sin necesariamente optimizar para los demás.
  - **Evaluación de nodos terminales:** Se evalúan los nodos terminales con una función de puntaje personalizada para cada jugador, indicando qué tan favorable es ese estado para cada uno de ellos.  
Esto se expresa en un vector de valores en lugar de un solo valor, donde cada elemento representa la utilidad para cada jugador. Por ejemplo, en un juego de tres jugadores, un nodo terminal puede tener un valor como [1,2,6], donde cada número es la puntuación para cada jugador respectivamente.



# Ejemplo en python

```
# Definición de la clase Nodo
class Nodo:
    def __init__(self, valor=None):
        self.valor = valor # Valor del nodo (puede ser una evaluación de un estado del juego)
        self.hijos = [] # Lista de nodos hijos

# Implementación del algoritmo Minimax
def minimax(nodo, profundidad, maximizando_jugador, nombre=""):
    # Imprime el estado actual del nodo que se está evaluando
    print(f"Evaluando nodo {nombre} a profundidad {profundidad}")

    # Condición de parada: si alcanzamos la profundidad 0 o el nodo no tiene hijos (es un nodo terminal)
    if profundidad == 0 or not nodo.hijos:
        print(f"Nodo {nombre} es terminal o profundidad 0 con valor {nodo.valor}")
        return nodo.valor # Devuelve el valor del nodo

    # Si es el turno del jugador que maximiza (MAX)
    if maximizando_jugador:
        max_evaluacion = float('-inf') # Inicializamos la evaluación máxima con el peor valor posible
        for i, hijo in enumerate(nodo.hijos):
            evaluacion = minimax(hijo, profundidad - 1, False, nombre + "-" + str(i))
            max_evaluacion = max(max_evaluacion, evaluacion) # Actualizamos la evaluación máxima
        return max_evaluacion
    else:
        # Si es el turno del jugador que minimiza (MIN)
        min_evaluacion = float('inf') # Inicializamos la evaluación mínima con el peor valor posible
        for i, hijo in enumerate(nodo.hijos):
            evaluacion = minimax(hijo, profundidad - 1, True, nombre + "-" + str(i))
            min_evaluacion = min(min_evaluacion, evaluacion) # Actualizamos la evaluación mínima
        return min_evaluacion
```

```
# Creación del árbol de ejemplo
A = Nodo()
B = Nodo()
C = Nodo()
D = Nodo()
E = Nodo()
F = Nodo()
G = Nodo()
H = Nodo(4)
I = Nodo(2)
J = Nodo(5)
K = Nodo(-3)
L = Nodo(3)
M = Nodo(-2)
N = Nodo(-4)
O = Nodo(5)

...
Estructura del árbol:
      A
     /   \
    B     C
   / \   / \
  D   E   F   G
 / \ / \ / \ / \
H I J K L M N O

Los valores terminales son:
H=4, I=2, J=5, K=-3, L=3, M=-2, N=-4, O=5
...
# Definimos las relaciones entre nodos
A.hijos = [B, C]
B.hijos = [D, E]
C.hijos = [F, G]
D.hijos = [H, I]
E.hijos = [J, K]
F.hijos = [L, M]
G.hijos = [N, O]
```

```
# Ejecutamos el algoritmo Minimax
resultado = minimax(A, 3, True, "A")
print(f"El valor óptimo es: {resultado}")
```

```
Evaluando nodo A a profundidad 3
Evaluando nodo A-0 a profundidad 2
Evaluando nodo A-0-0 a profundidad 1
Evaluando nodo A-0-0-0 a profundidad 0
Nodo A-0-0-0 es terminal o profundidad 0 con valor 4
Evaluando nodo A-0-0-1 a profundidad 0
Nodo A-0-0-1 es terminal o profundidad 0 con valor 2
Evaluando nodo A-0-1 a profundidad 1
Evaluando nodo A-0-1-0 a profundidad 0
Nodo A-0-1-0 es terminal o profundidad 0 con valor 5
Evaluando nodo A-0-1-1 a profundidad 0
Nodo A-0-1-1 es terminal o profundidad 0 con valor -3
Evaluando nodo A-1 a profundidad 2
Evaluando nodo A-1-0 a profundidad 1
Evaluando nodo A-1-0-0 a profundidad 0
Nodo A-1-0-0 es terminal o profundidad 0 con valor 3
Evaluando nodo A-1-0-1 a profundidad 0
Nodo A-1-0-1 es terminal o profundidad 0 con valor -2
Evaluando nodo A-1-1 a profundidad 1
Evaluando nodo A-1-1-0 a profundidad 0
Nodo A-1-1-0 es terminal o profundidad 0 con valor -4
Evaluando nodo A-1-1-1 a profundidad 0
Nodo A-1-1-1 es terminal o profundidad 0 con valor 5
El valor óptimo es: 4
```

## Número de evaluaciones: 15

# PODA ALFA-BETA

# Poda Alfa-Beta



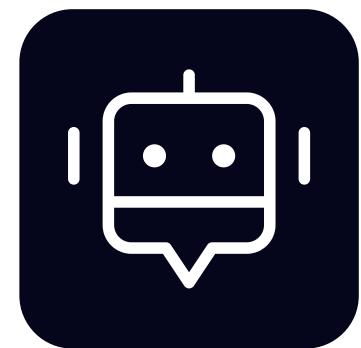
## Definición

Técnica que mejora la eficiencia del algoritmo MinMax al eliminar ramas innecesarias.



## Funcionamiento

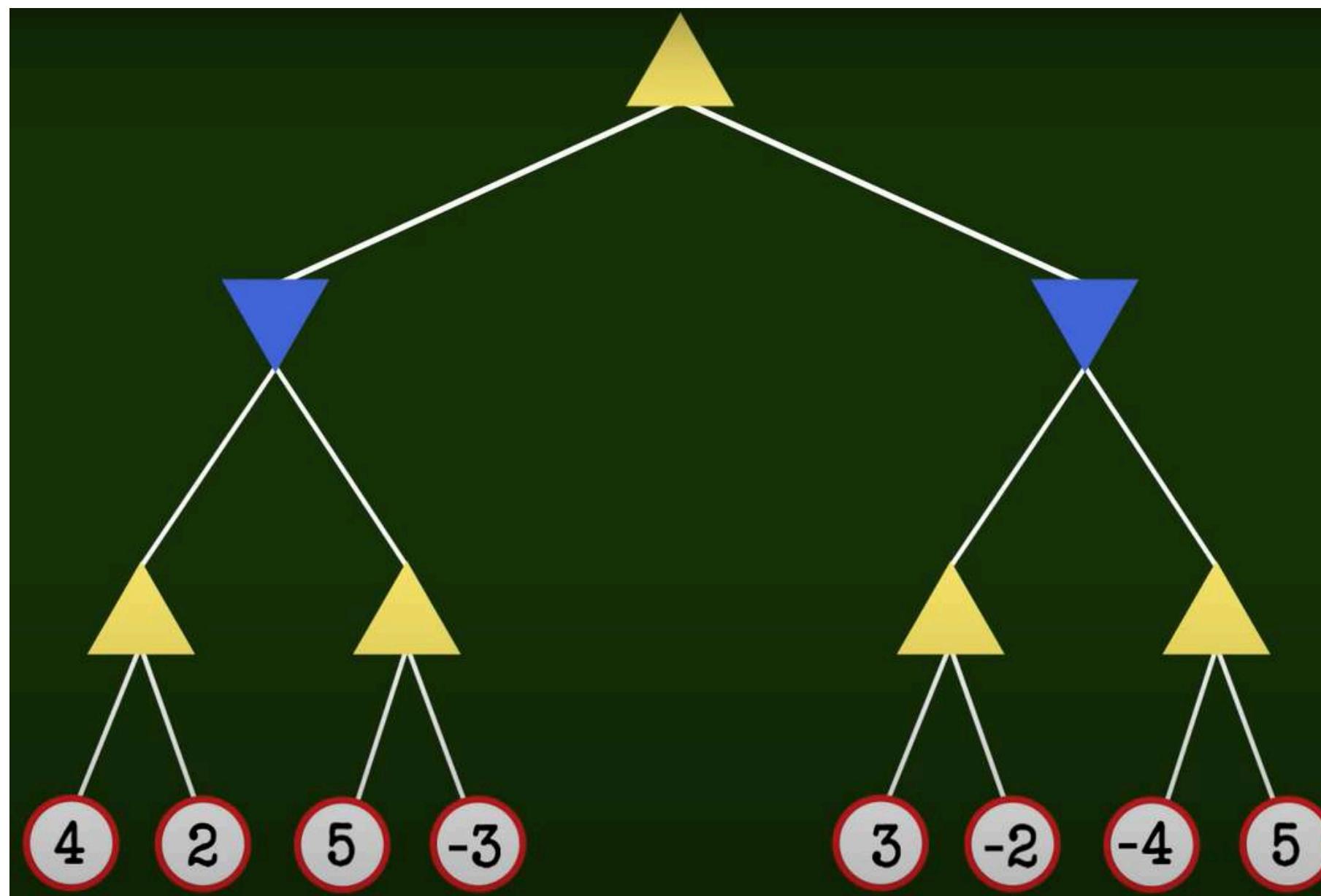
Al reducir la búsqueda, permite explorar más profundidad en el árbol.



## Estructura

- **Alfa:** Mejor puntaje que el jugador Max puede garantizar.
- **Beta:** Mejor puntaje que el jugador Min puede garantizar.

# Procedimiento



Al igual que en el algoritmo Min-max, se crea un árbol de decisiones con los posibles movimientos, y se sigue el mismo procedimiento de revisión, solo que se agregan ciertas cláusulas:

Se añaden los valores de alfa y beta como un intervalo, que inicializan en los peores casos, es decir, el valor de alfa es de  $-\infty$  y el de beta de  $\infty$ .

Posteriormente, al llegar a un nodo con un valor, se compara con lo que se necesite en el nodo padre directo:

- Si el valor que se busca es el máximo, el valor se compara y asigna a alfa, mientras que para un nodo con valor mínimo, se le asigna a beta.
- Después de comparar y asignar el valor de alfa, se revisa si se cumple la siguiente expresión:  
$$\alpha \geq \beta$$

En caso de que esto sea real, no se revisa los nodos restantes y se eliminan (poda). En caso de no cumplir con eso, se sigue con la búsqueda.

# Ejemplo en python

```
# Definición de la clase Nodo
class Nodo:
    def __init__(self, valor=None):
        self.valor = valor # Valor del nodo (puede ser una evaluación de un estado del juego)
        self.hijos = [] # Lista de nodos hijos

    # Implementación del algoritmo de poda alfa-beta
    def poda_alfa_beta(nodo, profundidad, alfa, beta, maximizando_jugador, nombre=""):
        # Imprime el estado actual del nodo que se está evaluando
        print(f"Evaluando nodo {nombre} a profundidad {profundidad} con alfa={alfa} y beta={beta}")

        # Condición de parada: si alcanzamos la profundidad 0 o el nodo no tiene hijos (es un nodo terminal)
        if profundidad == 0 or not nodo.hijos:
            print(f"Nodo {nombre} es terminal o profundidad 0 con valor {nodo.valor}")
            return nodo.valor # Devuelve el valor del nodo

        # Si es el turno del jugador que maximiza (MAX)
        if maximizando_jugador:
            max_evaluacion = float('-inf') # Inicializamos la evaluación máxima con el peor valor posible
            for i, hijo in enumerate(nodo.hijos):
                evaluacion = poda_alfa_beta(hijo, profundidad - 1, alfa, beta, False, nombre + "-" + str(i))
                max_evaluacion = max(max_evaluacion, evaluacion) # Actualizamos la evaluación máxima
            alfa = max(alfa, max_evaluacion) # Actualizamos alfa
            if beta <= alfa:
                print(f"Poda en nodo {nombre} con alfa={alfa} y beta={beta}")
                break # Poda beta: no necesitamos evaluar más nodos
            return max_evaluacion
        else:
            # Si es el turno del jugador que minimiza (MIN)
            min_evaluacion = float('inf') # Inicializamos la evaluación mínima con el peor valor posible
            for i, hijo in enumerate(nodo.hijos):
                evaluacion = poda_alfa_beta(hijo, profundidad - 1, alfa, beta, True, nombre + "-" + str(i))
                min_evaluacion = min(min_evaluacion, evaluacion) # Actualizamos la evaluación mínima
            beta = min(beta, min_evaluacion) # Actualizamos beta
            if beta <= alfa:
                print(f"Poda en nodo {nombre} con alfa={alfa} y beta={beta}")
                break # Poda alfa: no necesitamos evaluar más nodos
            return min_evaluacion
```

```
# Creación del árbol de ejemplo
A = Nodo()
B = Nodo()
C = Nodo()
D = Nodo()
E = Nodo()
F = Nodo()
G = Nodo()
H = Nodo(4)
I = Nodo(2)
J = Nodo(5)
K = Nodo(-3)
L = Nodo(3)
M = Nodo(-2)
N = Nodo(-4)
O = Nodo(5)

...
Estructura del árbol:

      A
      /   \
     B     C
    / \   / \
   D   E   F   G
  / \ / \ / \ / \
 H I J K L M N O

Los valores terminales son:
H=4, I=2, J=5, K=-3, L=3, M=-2, N=-4, O=5
...
# Definimos las relaciones entre nodos
A.hijos = [B, C]
B.hijos = [D, E]
C.hijos = [F, G]
D.hijos = [H, I]
E.hijos = [J, K]
F.hijos = [L, M]
G.hijos = [N, O]
```

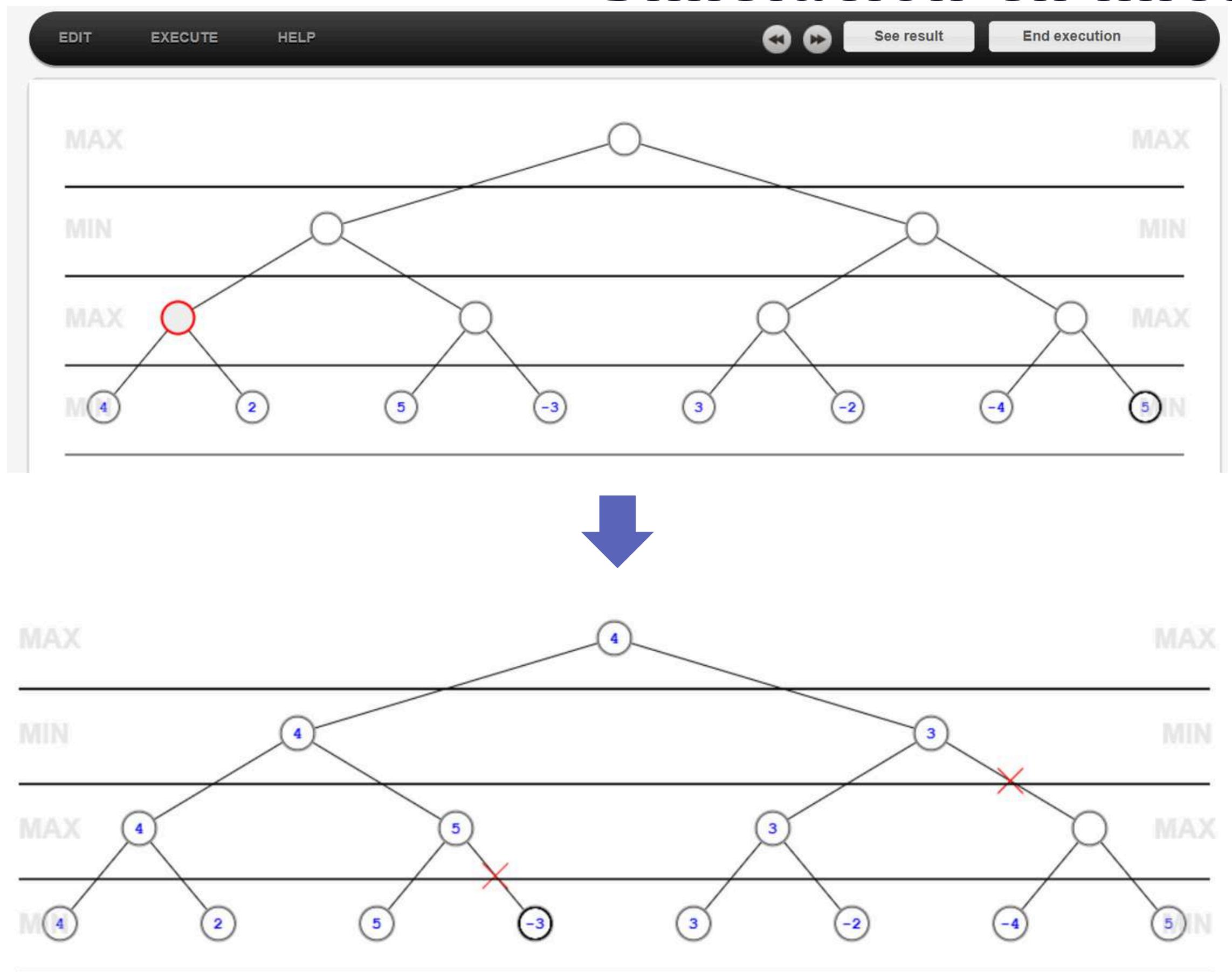
```
# Ejecutamos el algoritmo de poda alfa-beta
resultado = poda_alfa_beta(A, 3, float('-inf'), float('inf'), True, "A")
print(f"El valor óptimo es: {resultado}")

Evaluando nodo A a profundidad 3 con alfa=-inf y beta=inf
Evaluando nodo A-0 a profundidad 2 con alfa=-inf y beta=inf
Evaluando nodo A-0-0 a profundidad 1 con alfa=-inf y beta=inf
Evaluando nodo A-0-0-0 a profundidad 0 con alfa=-inf y beta=inf
Nodo A-0-0-0 es terminal o profundidad 0 con valor 4
Evaluando nodo A-0-0-1 a profundidad 0 con alfa=4 y beta=inf
Nodo A-0-0-1 es terminal o profundidad 0 con valor 2
Evaluando nodo A-0-1 a profundidad 1 con alfa=-inf y beta=4
Evaluando nodo A-0-1-0 a profundidad 0 con alfa=-inf y beta=4
Nodo A-0-1-0 es terminal o profundidad 0 con valor 5
Poda en nodo A-0-1 con alfa=5 y beta=4
Evaluando nodo A-1 a profundidad 2 con alfa=4 y beta=inf
Evaluando nodo A-1-0 a profundidad 1 con alfa=4 y beta=inf
Evaluando nodo A-1-0-0 a profundidad 0 con alfa=4 y beta=inf
Nodo A-1-0-0 es terminal o profundidad 0 con valor 3
Evaluando nodo A-1-0-1 a profundidad 0 con alfa=4 y beta=inf
Nodo A-1-0-1 es terminal o profundidad 0 con valor -2
Poda en nodo A-1 con alfa=4 y beta=3
El valor óptimo es: 4
```

Número de evaluaciones:

11

# Simulación en linea



En la simulación se puede escoger entre usar el algoritmo MinMax o el Poda Alfa-Beta.



# Decisiones en tiempo real



## DESAFÍOS

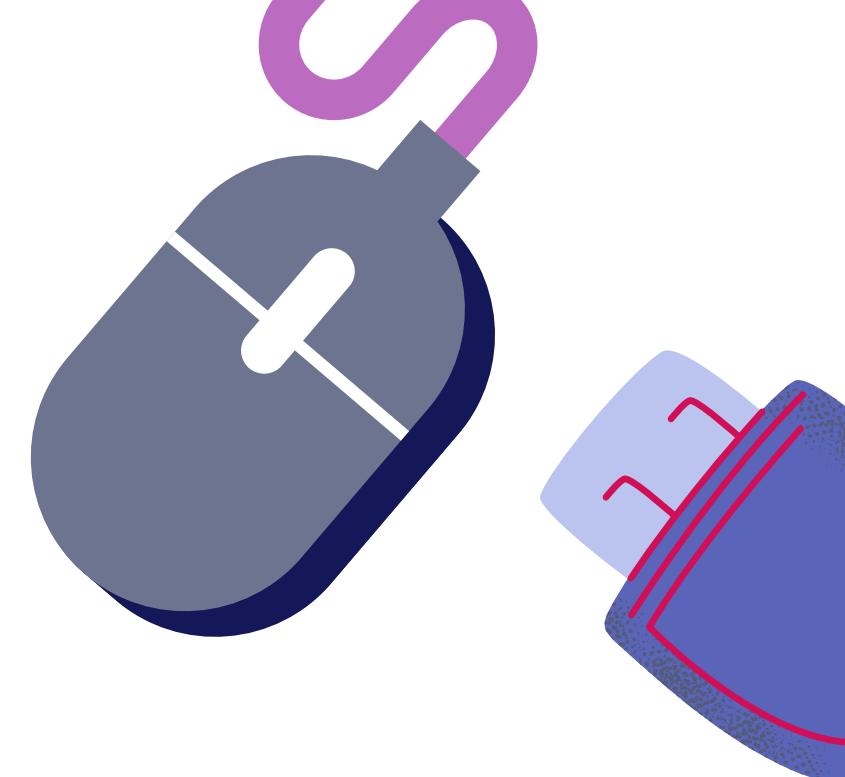
En juegos de tiempo real, los jugadores deben decidir rápidamente sin un análisis completo.

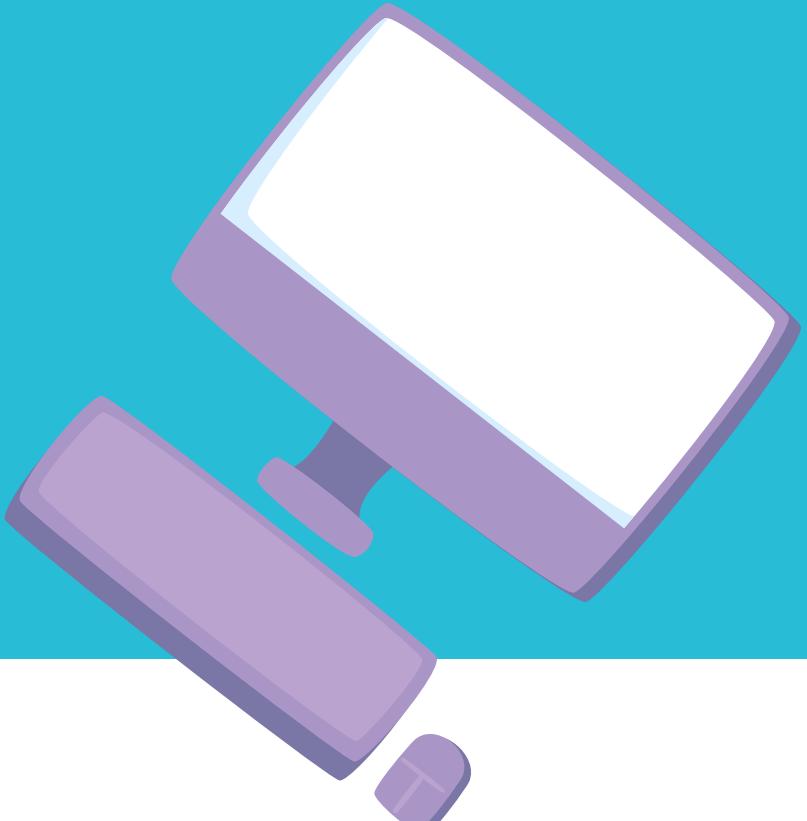
## ESTRATEGIAS

- Reducción de la profundidad de búsqueda.
- Uso de funciones heurísticas para evaluaciones rápidas.

## EJEMPLO

Juegos como StarCraft requieren decisiones rápidas y estratégicas.

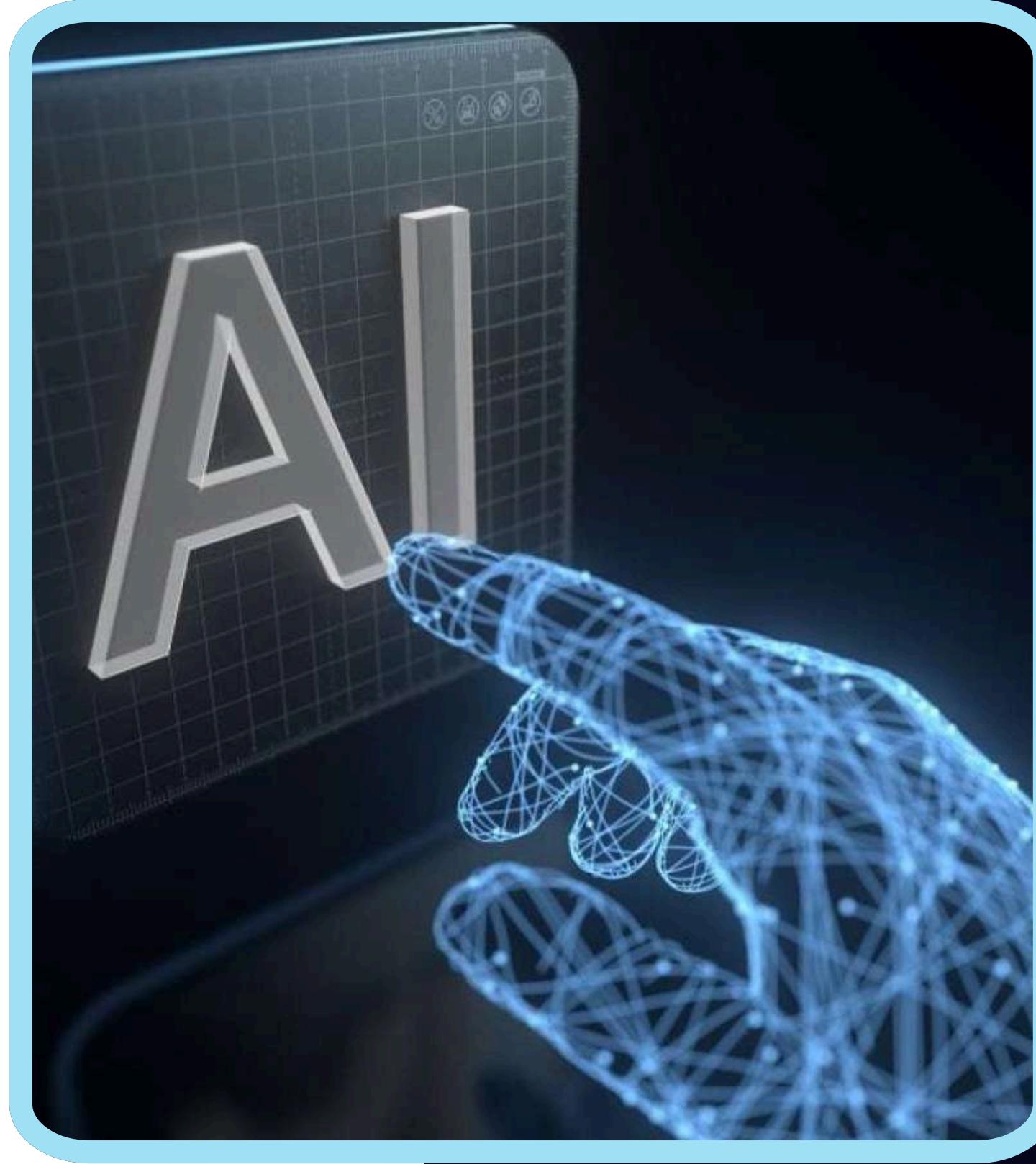




# ¿Qué es la Toma de Decisiones en Tiempo Real?

La toma de decisiones en tiempo real es la habilidad de un sistema de inteligencia artificial para evaluar y responder rápidamente a cambios en su entorno, especialmente en videojuegos y aplicaciones que requieren respuestas inmediatas.





# Estrategias Heurísticas

- **Definición:** Funciones que evalúan rápidamente la calidad de una posición sin un análisis exhaustivo.
- **Uso:** Ayudan a tomar decisiones rápidas en situaciones de tiempo limitado.
- **Ejemplos:** Heurísticas en ajedrez (control del centro, desarrollo de piezas).



# APLICACIONES EN

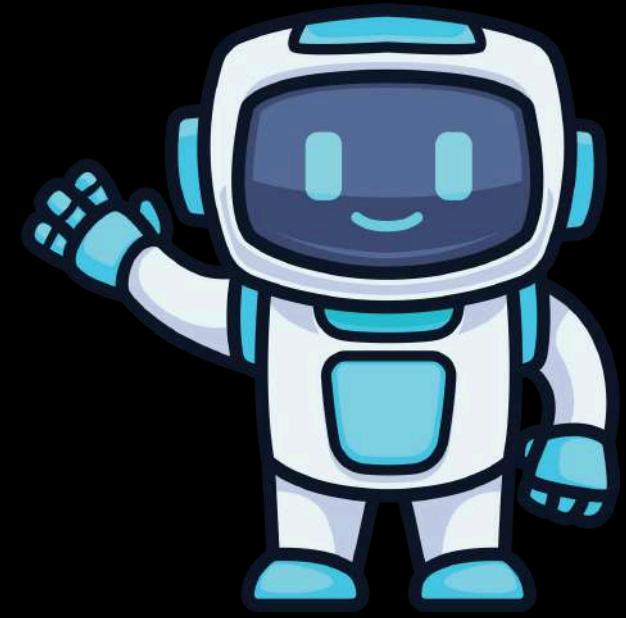
# VIDEOJUEGOS



La IA en juegos como StarCraft analiza recursos, tropas y movimiento en tiempo real.

Cada decisión de la IA se basa en lo que hace el jugador, creando una experiencia de adversario más dinámica y desafiante.

PLAY



# Aplicaciones COTIDIANAS

- Vehículos autónomos: En un cruce de tráfico, un automóvil autónomo debe prever las acciones de otros conductores o peatones en segundos y tomar decisiones rápidas para evitar colisiones.
- Robots en competiciones: Los robots de combate o de fútbol deben reaccionar en milisegundos ante los movimientos de sus oponentes, usando estrategias de evasión o ataque en tiempo real.



# Ejemplo en python

```
import tkinter as tk
import time
import random

# Variables globales
start_time = None
reaction_time = None

# Función para cambiar el color a verde en un tiempo aleatorio
def change_to_green():
    global start_time
    root.config(bg="green")
    start_time = time.time() # Registrar el momento exacto en que cambia a verde

# Función que se ejecuta cuando el usuario hace clic
def on_click(event):
    global reaction_time
    if start_time is not None: # Solo mide si el color ya cambió a verde
        reaction_time = time.time() - start_time
        result_label.config(text=f"Tiempo de reacción: {reaction_time:.3f} segundos")
    reset_button.pack(pady=10)

# Función para iniciar el juego
def start_game():
    result_label.config(text="Esperando...")
    root.config(bg="red")
    reset_button.pack_forget() # Oculta el botón de reinicio
    root.after(random.randint(2000, 5000), change_to_green) # Cambia el color a verde al azar
```

```
# Configuración de la interfaz gráfica
root = tk.Tk()
root.title("Juego de Tiempo de Reacción")
root.geometry("400x300")

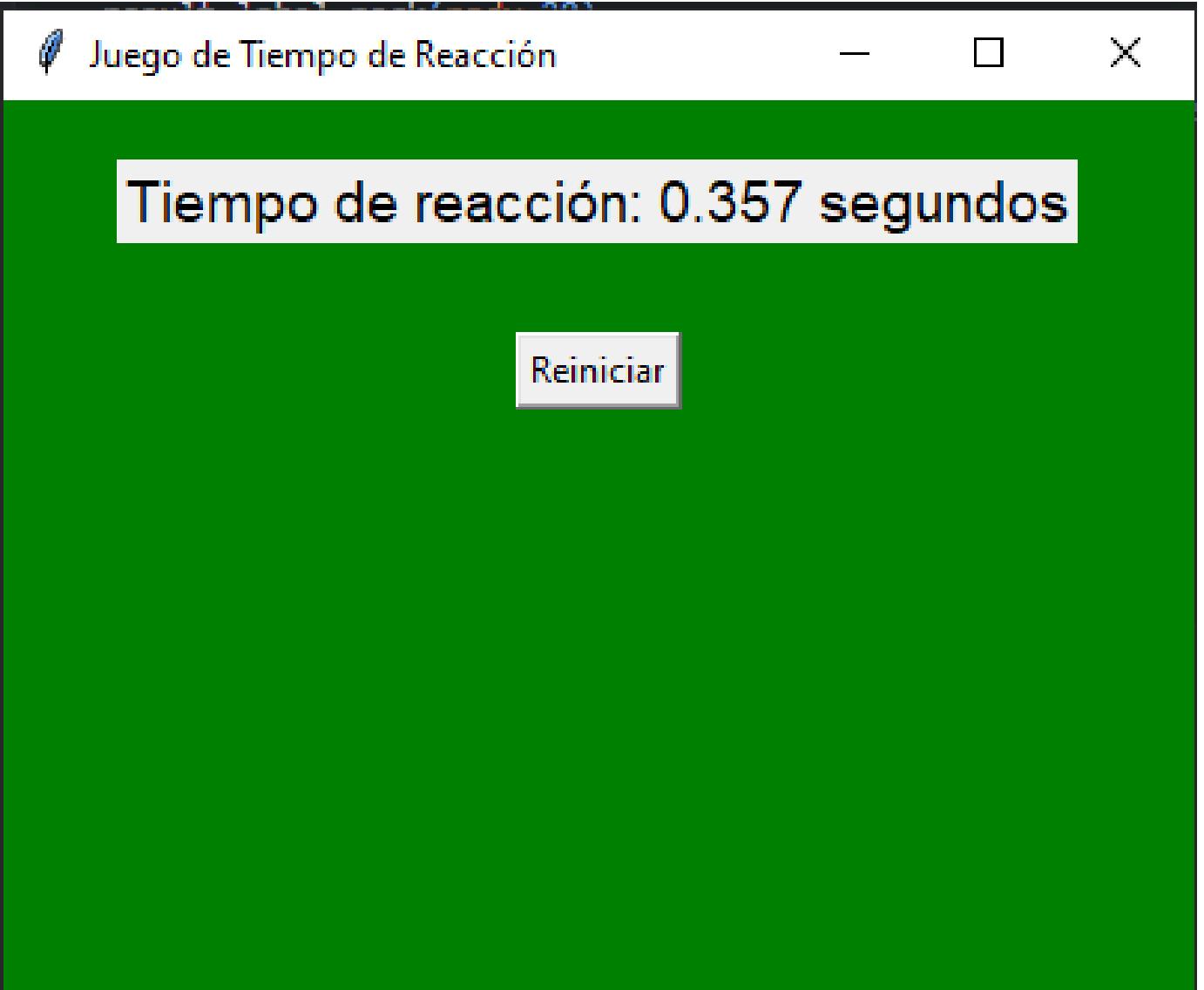
result_label = tk.Label(root, text="Haz clic cuando la pantalla se ponga verde", font=("Arial", 14))
result_label.pack(pady=20)

reset_button = tk.Button(root, text="Reiniciar", command=start_game)

# Configurar el color de fondo inicial y vincular el clic a la función
root.config(bg="red")
root.bind("<Button-1>", on_click)

start_game() # Comenzar el juego

root.mainloop()
```



```

for i in range(num_of_enemies):
    enemy_x.append(random.randint(0, screen_width - 64))
    enemy_y.append(random.randint(50, 150))
    enemy_x_change.append(enemy_speed)
    enemy_y_change.append(40)

# Bala
bullet_x = 0
bullet_y = player_y
bullet_y_change = 1
bullet_state = "ready" # Estado 'ready' significa que la bala no se ve en pantalla

# Puntuación
score_value = 0
font = pygame.font.Font('freesansbold.ttf', 32)
text_x = 10
text_y = 10

# Fin del juego
game_over_font = pygame.font.Font('freesansbold.ttf', 64)

def show_score(x, y):
    score = font.render(f"Puntuación: {score_value}", True, (255, 255, 255))
    screen.blit(score, (x, y))

def game_over_text():
    over_text = game_over_font.render("GAME OVER", True, (255, 255, 255))
    screen.blit(over_text, (200, 250))

def player(x, y):
    screen.blit(player_img, (x, y))

def enemy(x, y, i):
    screen.blit(enemy_img, (x, y))

def fire_bullet(x, y):
    global bullet_state

```

```

        bullet_x = x
        bullet_y = y
        bullet_state = "fire"

    def is_collision(enemy_x, enemy_y, bullet_x, bullet_y):
        distance = math.sqrt(math.pow(enemy_x - bullet_x, 2) + math.pow(enemy_y - bullet_y, 2))
        return distance < 27

    # Ciclo principal del juego
    running = True
    while running:
        screen.fill((0, 0, 0)) # Color de fondo
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False

        # Movimiento de la nave
        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_LEFT:
                player_x_change = -0.5
            if event.key == pygame.K_RIGHT:
                player_x_change = 0.5
            if event.key == pygame.K_SPACE:
                if bullet_state == "ready":
                    fire_bullet(bullet_x, bullet_y)

        if event.type == pygame.KEYUP:
            if event.key == pygame.K_LEFT or event.key == pygame.K_RIGHT:
                player_x_change = 0

        # Movimiento de la nave
        player_x += player_x_change

```

```

        enemy_x[i] += enemy_x_change[i]
        if enemy_x[i] <= 0 or enemy_x[i] >= screen_width - 64:
            enemy_x_change[i] *= -1
            enemy_y[i] += enemy_y_change[i]

        # Colisión
        collision = is_collision(enemy_x[i], enemy_y[i], bullet_x, bullet_y)
        if collision:
            bullet_y = player_y
            bullet_state = "ready"
            score_value += 10
            enemy_x[i] = random.randint(0, screen_width - 64)
            enemy_y[i] = random.randint(50, 150)

            enemy(enemy_x[i], enemy_y[i], i)

        # Movimiento de la bala
        if bullet_y <= 0:
            bullet_y = player_y
            bullet_state = "ready"

```

```

        if bullet_state == "fire":
            fire_bullet(bullet_x, bullet_y)
            bullet_y -= bullet_y_change

        # Aumenta la velocidad de los enemigos a medida que aumenta la puntuación
        if score_value % 50 == 0 and score_value != 0:
            for k in range(num_of_enemies):
                enemy_x_change[k] *= 1.05

    player(player_x, player_y)
    show_score(text_x, text_y)
    pygame.display.update()

pygame.quit()

```



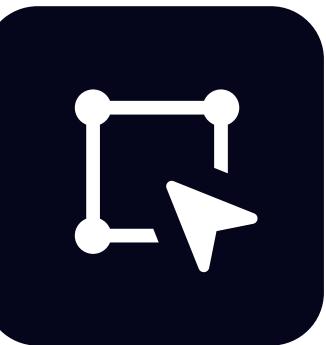
# ALGORITMO JUEGOS DE POSIBILIDAD

# JUEGOS DE POSIBILIDAD



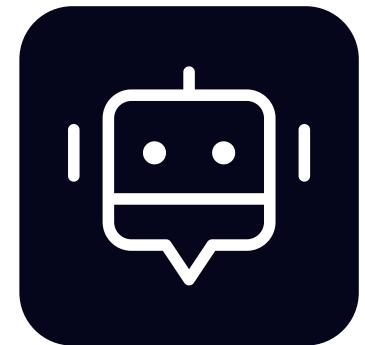
## Definición

Juegos donde la incertidumbre juega un papel crucial.



## Importancia

Necesidad de evaluar estrategias bajo condiciones de incertidumbre.



## Ejemplos

Póker, Blackjack, donde el resultado depende de la información oculta.

# Estrategias de juegos de posibilidad

01

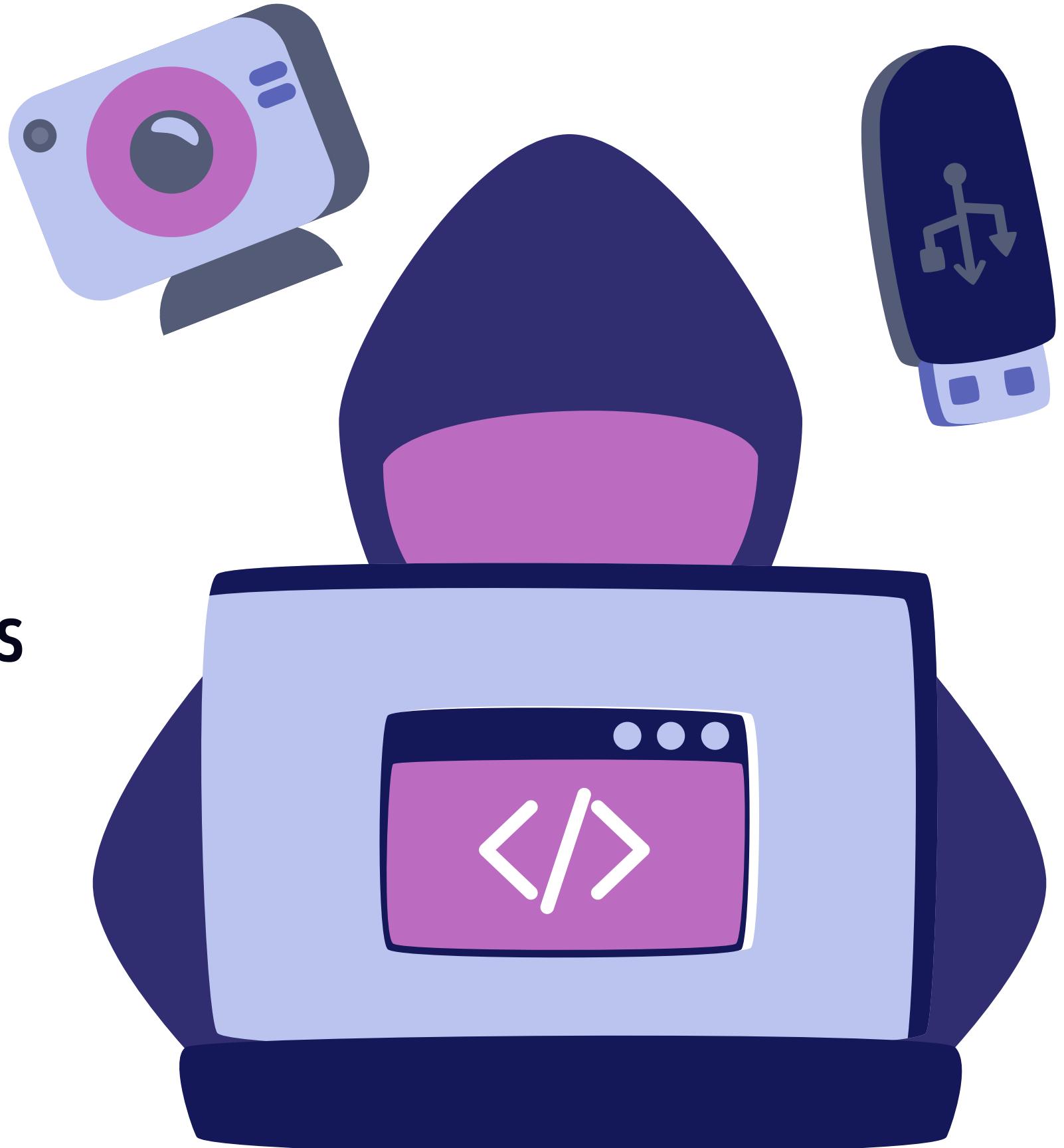
ANÁLISIS

Estrategias que toman en cuenta la información incompleta.

02

TEORÍA DE JUEGOS

Aplicación de la teoría para maximizar oportunidades y minimizar riesgos.



# COMPARACIÓN ENTRE JUEGOS DETERMINISTAS Y DE POSIBILIDAD

## JUEGOS DETERMINISTAS

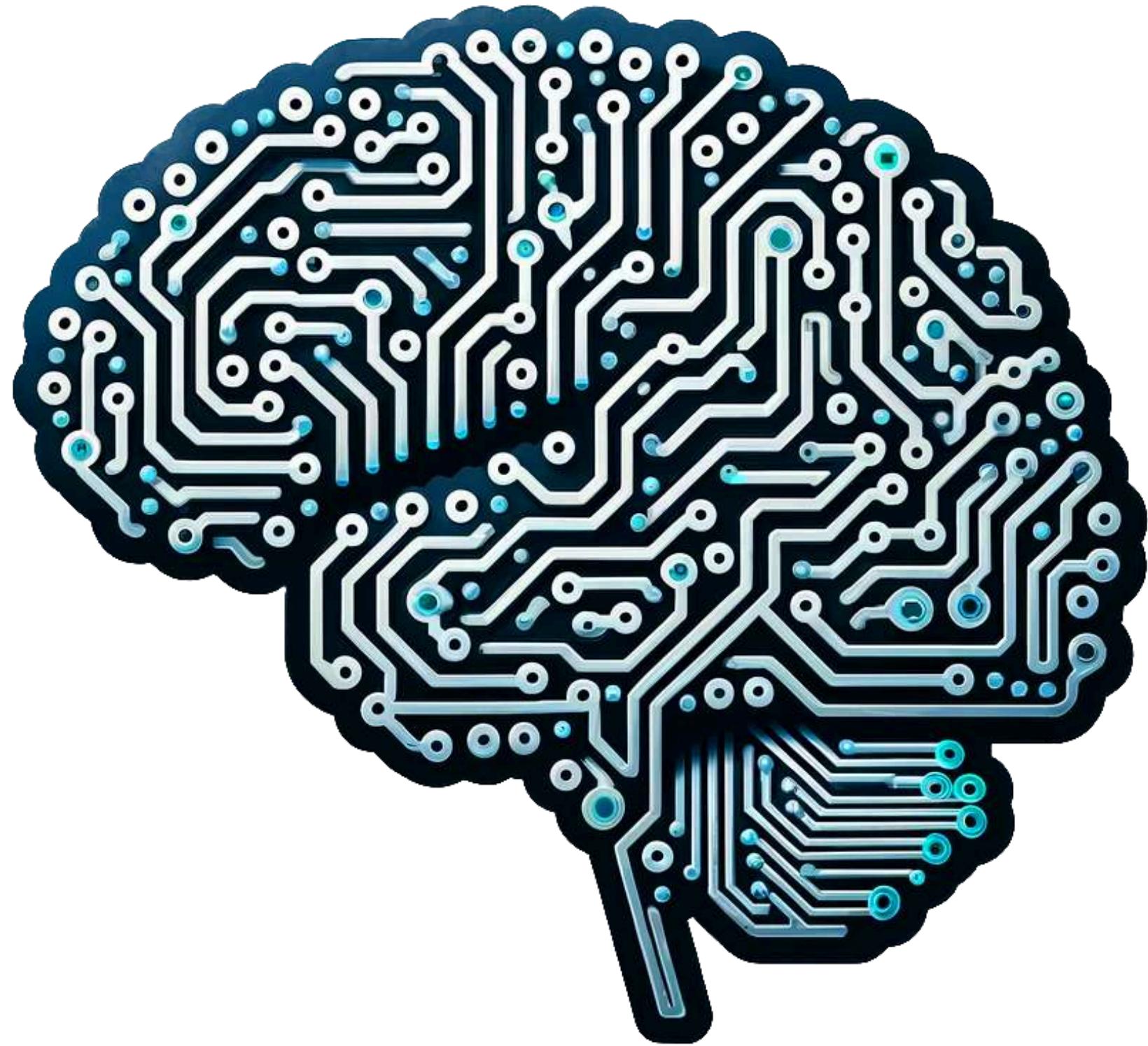
- Resultados predecibles
- Ejemplo: Ajedrez

## JUEGOS DE POSIBILIDAD

- Resultados inciertos
- Ejemplo: Cartas



# FUTURO DE LOS ALGORITMOS DE BÚSQUEDA



## TENDENCIA

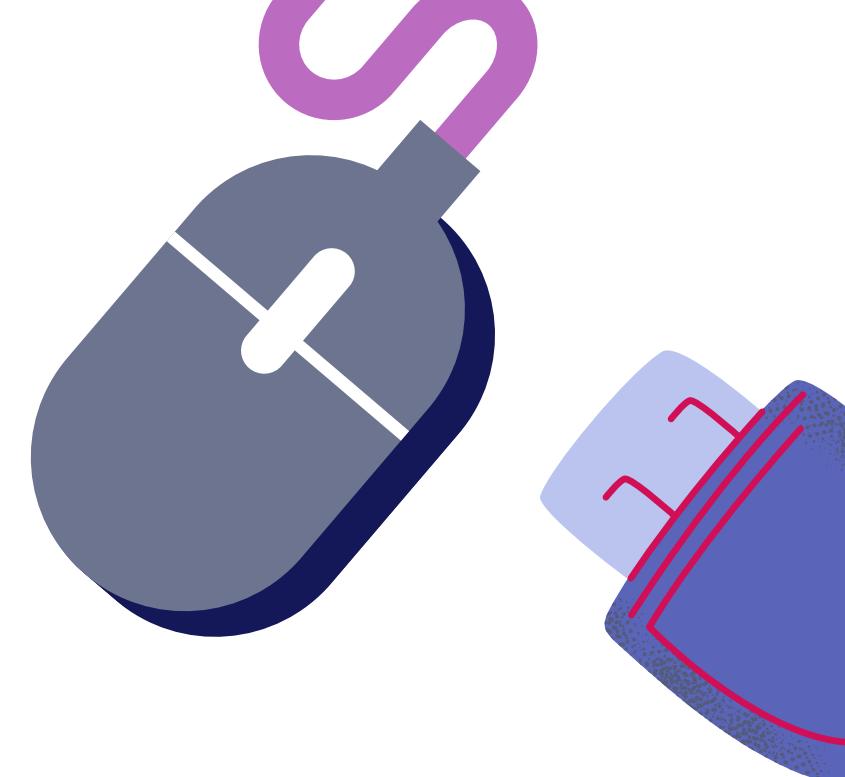
Evolución hacia algoritmos más eficientes y adaptativos.

## AVANCES TECNOLÓGICOS

Aprendizaje automático y su integración con algoritmos de búsqueda.

## IMPACTO

Transformaciones en la industria del entretenimiento y más allá.

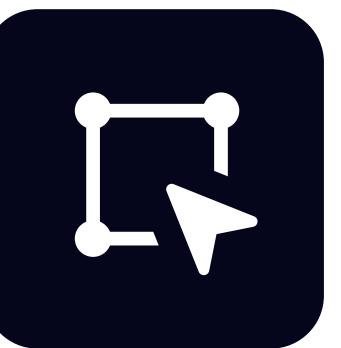


# CONCLUSIONES



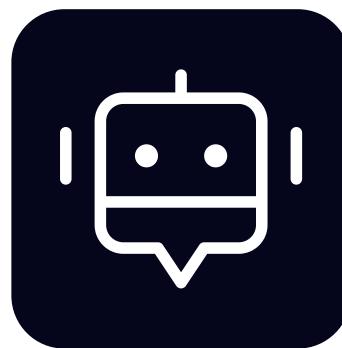
## SOBRE ALGORITMOS

- **MinMax:** Evalúa movimientos anticipando respuestas del oponente.
- **Poda Alfa-Beta:** Optimiza la búsqueda reduciendo nodos evaluados.
- **Decisiones en Tiempo Real:** Clave para agentes inteligentes en entornos competitivos.



## MIN-MAX o PODA

La combinación de MinMax y poda Alfa-Beta es crucial para el rendimiento en juegos.



¿?

Preguntas

# GRACIAS

# REFERENCIAS

- Stuart Russell, Peter Norvig. INTELIGENCIA ARTIFICIAL. UN ENFOQUE MODERNO. Segunda edición. PEARSON EDUCACIÓN, S.A., Madrid, 2004.
- Bruno López Takeyas. PODA ALFA-BETA. Instituto Tecnológico de Nuevo Laredo.  
<https://nlaredo.tecnm.mx/takeyas/Apuntes/Inteligencia%20Artificial/Apuntes/IA/Alfa-Beta.pdf>
- Raphael Rocha da Silva. Minimax simulator. [https://raphsilva.github.io/utilities/minimax\\_simulator/#](https://raphsilva.github.io/utilities/minimax_simulator/#)