Arimaa

Cabal Correa Andres Felipe - 2160339
Cuestas Parada Daniel Jose - 2067550
Moreno Gil Johan Alejandro - 2160052

Facultad de Ingeniería, Universidad del Valle - Sede Tuluá
Introducción a la Inteligencia Artificial

Mtr. Triana Madrid Joshua David

Diciembre 2024

Índice

Resumen

Se desarrolló un programa en el lenguaje de programación JavaScript en el que se aplicaron los conceptos fundamentales de IA vistos en el curso con el objetivo de programar comportamientos específicos a un Agente de software permitiéndolela toma de decisiones inteligentes usando el método de minimax, sobre el juego **arimaa**.

Introducción

Los modelos de inteligencia artificial clásicos establecieron las bases para el desarrollo de tecnologías modernas en el campo de la IA, para la apropiación de estos conceptos se plantea el desarrollo de un juego en el cual mediante del uso del método minimax se pretende hacer un IA vs Player.

Objetivo General

Desarrollar el juego arimaa en JavaScript donde las fichas doradas serán controladas por una IA y las plateadas por un jugador.

Objetivos específicos:

- Desarrollar un agente para que mueva las fichas doradas buscando minimizar la pérdida esperada en cada movimiento
- Desarrollar el juego arimaa con todas sus reglas
- Desarrollar una interfaz gráfica donde se pueda visualizar e interactuar con el tablero.

Metodología

juego

Para el desarrollo de este proyecto se implementó metodología de clases, para esto se identificaron clases para cada tipo de pieza, una superclase para todas las piezas y una clase para el juego. Cada clase cuenta con sus métodos que ayudan a la resolución o cumplimiento de alguna necesidad dentro del juego.

Se identificaron las reglas del juego y se crearon métodos que validen el cumplimiento de estas para los comportamientos de las piezas

IA (Inteligencia Artificial)

Primero se buscó obtener todas las permutaciones posibles de las piezas de un color sobre el tablero teniendo en cuenta tipo y dirección del movimiento, para simular estas permutaciones se crearon clones del objeto game principal. Seguido de esto hicimos la construcción del árbol minimax utilizando la simulaciones creadas anteriormente y simulando sobre estas simulaciones y así diferenciar los movimientos de min y max. Por último cuando se llega a los nodos finales se hace un evaluación del estado, para esta evaluación utilizamos las siguientes heurísticas:

- fichas en el centro del tablero
- Distancia de los conejos hacia la meta
- Si no está congelada o bloqueada por otra pieza

Estas heurísticas se calculan para el jugador actual y el oponente, con esto sumamos los cálculos propios y restamos los del enemigo.

Luego de construir el árbol minimax ya podemos hacer la búsqueda, para esto usamos una función que recibe un nodo y hace la búsqueda recursiva buscando un nodo con menor value para min y mayor valor para max, al finalizar la recursión nos retorna el nodo de decisión.

Desarrollo

Interfaz

Se creó un canvas de html para renderizar las fichas y el tablero, este recibe los eventos click sobre la interfaz donde primero se selecciona la ficha y luego se habilitan las opciones de movimiento, para luego ejecutar el movimiento deseado teniendo el from desde el active cell que es la pieza que seleccione primero, y el to sería la celda que se encuentra en las coordenadas paradas en donde le hice click.

```
canvas.addEventListener("click", (event: MouseEvent) => {
    const offset = parseOffsetToCoordinates(event, canvas);
    const cell = game.getCellAt(offset[0], offset[1]);

    if (game.isMoving) {
        const movement: GameMovement = {
            from: game.activeCell!,
            to: cell,
            player: game.currentPlayer,
            type: "simple",
        };

        const playerColor = game.currentPlayer.color;

        if (game.isMoving === "simple") {
            game.simpleMovement(movement);
        } else if (game.isMoving === "push") {
            if (!game.floatingPiece) {
                movement.type = "pre-push";
                game.pushMovement(movement);
            } else {
                movement.type = "push";
                game.pushPiece(movement);
            }
        } else if (game.isMoving === "pull") {
            if (!game.floatingPiece) {
                movement.type = "pre-pull";
                game.pullMovement(movement);
            } else {
                movement.type = "pull";
                game.pullPiece(movement);
            }
        }
        if (game.currentPlayer.color !== playerColor) {
            game.playIA();
        }

        disableMenu();
        return;
    }

    onCellClick(event, game, canvas);
});

canvas.addEventListener("mousemove", (event: MouseEvent) => onCellHover(event, game, canvas));
```

cada botón incluye la lógica necesaria para este tipo de movimiento, o también puedo deshacer un movimiento lo que nos borra el último elemento del historial y deja todos los estados como iniciales, desde estos eventos se habilitaban los movimientos válidos según las opciones que ofreciera el tipo de movimiento.

Ubicación de las fichas

La función randomfill coloca aleatoriamente diferentes tipos de piezas sobre él tablero, respetando las cantidades máximas de cada tipo y asegurándose de no sobrescribir posiciones ya ocupadas, también diferencia las posiciones iniciales en el tablero según el color del jugador (gold o silver).

```
public randomFill(): void {
    [this.playerGold, this.playerSilver].forEach((player) => {
        let rabbitCount = 8;
        let dogCount = 2;
        let catCount = 2;
        let horseCount = 2;
        let camelCount = 1;
        let elephantCount = 1;

        while (rabbitCount > 0 || dogCount > 0 || catCount > 0 || horseCount > 0 || camelCount > 0 || elephantCount > 0) {
            const y = Math.floor(Math.random() * this.board.length);
            const x = player.color === "silver" ? Math.floor(Math.random() * 2) : Math.floor(Math.random() * 2) + (this.board[0].length - 2);
            let piece: Piece | null = null;

            if (this.board[x][y] === 0 && rabbitCount > 0) {
                piece = new Rabbit(player.color, [x, y], this.board, this);
                rabbitCount--;
            } else if (this.board[x][y] === 0 && dogCount > 0) {
                piece = new Dog(player.color, [x, y], this.board, this);
                dogCount--;
            } else if (this.board[x][y] === 0 && catCount > 0) {
                piece = new Cat(player.color, [x, y], this.board, this);
                catCount--;
            } else if (this.board[x][y] === 0 && horseCount > 0) {
                piece = new Horse(player.color, [x, y], this.board, this);
                horseCount--;
            } else if (this.board[x][y] === 0 && camelCount > 0) {
                piece = new Camel(player.color, [x, y], this.board, this);
                camelCount--;
            } else if (this.board[x][y] === 0 && elephantCount > 0) {
                piece = new Elephant(player.color, [x, y], this.board, this);
                elephantCount--;
            }
            if (piece) {
                console.log(`Piece ${piece.toString()}`, piece.game.id, this.id);
                player.pieces.push(piece);
                this.placePiece(piece);
            }
        }
    });
}
```

Agentes

En el tablero del juego de tamaño 8x8 cada jugador controla 16 piezas, cada tipo de pieza tiene un peso diferente, las piezas son las siguientes:

- 1 Elefante peso:
- 1 Camello
- 2 Caballos

- 2 Perros
- 2 Gatos
- 8 Conejos

Cada pieza fue creada con los siguientes atributos

```
export class Piece {

    public color: ColorPiece;

    public icon;

    public weight: number;

    public position: number[];

    public name: "Rabbit" | "Horse" | "Camel" | "Elephant" | "Dog" | "Cat";

    public isFloating: boolean = false;

    public active: boolean = false;

    public game: Game;

```

Piece properties image

Reglas

Movimientos

En Arimaa las piezas se mueven solo en direcciones cardinales y avanzan una casilla a la vez, todas pueden moverse en todas las direcciones (adelante, atrás, izquierda y derecha) a excepción del conejo, este no puede dar pasos hacia atrás.

```
getAdjacentsMovements(position: number[]): coordinates[] {
        const [x, y] = position;
        let adjacents: coordinates[] = [];

        if (this.game.board[x - 1] && this.game.board[x - 1][y] !== undefined) adjacents.push([x - 1, y]);
        if (this.game.board[x + 1] && this.game.board[x + 1][y] !== undefined) adjacents.push([x + 1, y]);
        if (this.game.board[x][y - 1] !== undefined) adjacents.push([x, y - 1]);
        if (this.game.board[x][y + 1] !== undefined) adjacents.push([x, y + 1]);

        return adjacents;
    }
```

adjacents movements image

Un movimiento equivale a un paso por ficha, en cada turno el jugador tiene 4 movimientos que se pueden usar en 4 piezas diferentes o todos en la misma.

Congelación

Una pieza que esté adyacente a una pieza enemiga más fuerte se congela, a menos que también esté adyacente con una pieza amiga.

Para esto se busca en las casillas adyacentes si existe una pieza amiga (el color de la pieza debe ser igual al color del jugador) en caso de no haber piezas amigas y si enemigas se valida si alguna de estas piezas tiene un peso mayor a la pieza a evaluar

```
isFreezed(): boolean {
    let weighterPieces = 0;

    const adjacentTiles = this.getAdjacentsMovements(this.position);

    for (const tile of adjacentTiles) {
        const [x, y] = tile;

        if (!this.game.board[x][y] || this.game.board[x][y] === 1 || this.game.board[x][y] === 0) continue;
        const piece = this.game.board[x][y] as Piece;

        // If the adjacent piece is from the same color, then the piece is not freezed
        if (piece.color === this.color) return false;

        // If the adjacent piece is heavier, then the piece is freezed
        if (piece.weight >= this.weight) weighterPieces++;
    }

    return weighterPieces > 0;
}
```

isFreezed image

Empujar o Halar

Una pieza puede empujar o jalar a una pieza enemiga más débil (de menos peso) que esté a su lado, siempre que una casilla adyacente esté vacía y permita este movimiento.

En un tirón, una pieza entra en un cuadrado vacío adyacente y arrastra la pieza enemiga más débil al cuadrado de donde proviene. Para esto primero se buscar si en las posiciones adyacentes a la pieza existen piezas a las que pueda halar, debe existir alguna pieza enemiga en las posiciones adyacentes, la pieza enemiga debe ser de el color contrario al color del jugador y de un peso menor

```
getPullablePieces(): AvailableMovement[] {
        const adjacentTiles = this.getAdjacentsMovements(this.position);
        let pullablePieces: AvailableMovement[] = [];

        if (adjacentTiles.some((tile) => this.game.board[tile[0]][tile[1]] === 0)) {
            for (const tile of adjacentTiles) {
                const [x, y] = tile;

                if (this.game.board[x][y] === 0 || this.game.board[x][y] === 1) continue;

                const piece = this.game.board[x][y] as Piece;
                if (piece.color === this.color || piece.weight > this.weight) continue;

                pullablePieces.push({
                    coordinates: piece.position,
                    type: "pull",
                });
            }
        }
        return pullablePieces;
    }
```

get pullable pieces image

Luego se hace la validación de que esto se pueda ejecutar correctamente y poner a la pieza enemiga en un estado **flotante** lo cual nos indica que existe una ficha enemiga que se va a mover con un movimiento propio.

```
public pullMovement(movement: GameMovement): void {
    const { from, to, player } = movement;
    const [toX, toY] = to;
    const piece = this.getPieceAt(from)!;

    if (!this.availableMovements.some((movement) => movement.coordinates[0] === toX && movement.coordinates[1] === toY)) {
        showErrorMessage("Invalid movement: The piece can't move to that position");
        throw new Error("Invalid movement: The piece can't move to that position");
    }

    const enemyPiece = this.getPieceAt(to)!;
    this.floatingPiece = enemyPiece;

    this.completeMovement(player, movement, true);
    this.availableMovements = piece.getSimpleMovements();
    this.activeCell = from;
}
```

pull movement image

Para finalizar el pull acomoda la pieza y la pieza flotante en el tablero, yendo la pieza propia hacia la posición que se escoge por la interfaz (el To) y la pieza flotante hacia la posición de donde viene la pieza propia (el From).

```
public pullPiece(movement: GameMovement): void {
    const { from, to, player } = movement;
    const [fromX, fromY] = from!;
    const [toX, toY] = to;
    const piece = this.getPieceAt(from)!;
    const enemyPiece = this.floatingPiece!;

    if (!this.availableMovements.some((movement) => movement.coordinates[0] === toX && movement.coordinates[1] === toY)) {
        showErrorMessage("Invalid movement: The piece can't move to that position");
        throw new Error("Invalid movement: The piece can't move to that position");
    }

    this.board[enemyPiece.position[0]][enemyPiece.position[1]] = 0;
    this.board[fromX][fromY] = enemyPiece;
    enemyPiece.updatePosition([fromX, fromY]);

    piece.updatePosition([toX, toY]);
    this.board[toX][toY] = piece;
    this.floatingPiece = null;
    this.completeMovement(player, movement);
}
```

pull piece image

En un empujón la pieza enemiga más débil se mueve a una casilla adyacente y la pieza que empujo se mueve a la casilla que había ocupado. Para esto se utilizó la misma lógica que el pull, primero obteniendo las piezas a las que puedo empujar que estén adyacentes con las mismas condiciones que el pull.

```
getPushablePieces() {
    const adjacentTiles = this.getAdjacentsMovements(this.position);
    let pushablePieces: AvailableMovement[] = [];

    for (const tile of adjacentTiles) {
        const [x, y] = tile;

        if (this.game.board[x][y] === 0 || this.game.board[x][y] === 1) continue;

        const piece = this.game.board[x][y] as Piece;
        if (piece.color === this.color || piece.weight > this.weight) continue;

        pushablePieces.push({
            coordinates: piece.position,
            type: "push",
        });
    }

    return pushablePieces;
}
```

piezas empujables image

seguido se efectua el movimiento de la pieza que hace el push (empuje) en el tablero y la pieza a la que se le hace el push se pone como ficha flotante

```
public pushMovement(movement: GameMovement): void {
    const { from, to, player } = movement;
    const [fromX, fromY] = from!;
    const [toX, toY] = to;
    const piece = this.getPieceAt(from!)!;

    if (!this.availableMovements.some((movement) => movement.coordinates[0] === toX && movement.coordinates[1] === toY)) {
        showErrorMessage("Invalid movement: The piece can't move to that position");
        throw new Error("Invalid movement: The piece can't move to that position");
    }

    const enemyPiece = this.getPieceAt(to)!;

    this.board[fromX][fromY] = 0;
    this.board[toX][toY] = piece;
    piece.updatePosition([toX, toY]);
    this.floatingPiece = enemyPiece;

    this.completeMovement(player, movement, true);
    this.availableMovements = enemyPiece.getSimpleMovements();
}
```

push movement image

Para finalizar el movimiento de push se hace la actualización de la pieza que está flotante en el tablero acomodándose en una de las casillas adyacentes que estén libres.

```
public pushPiece(movement: PushMovement): void {
    const { to, player } = movement;
    const [toX, toY] = to;
    const piece = this.floatingPiece!;

    if (!this.availableMovements.some((movement) => movement.coordinates[0] === toX && movement.coordinates[1] === toY)) {
        showErrorMessage("Invalid movement: The piece can't move to that position");
        throw new Error("Invalid movement: The piece can't move to that position");
    }

    this.floatingPiece = null;
    this.board[toX][toY] = piece;
    piece.updatePosition([toX, toY],);

    this.completeMovement(player, movement);
}
```

push piece image

Capturas

En arimaa, una captura ocurre en las casillas trampa, una pieza que entra en una casilla trampa se retira del tablero a menos que haya una pieza amiga al lado de esa trampa. Primero, hicimos una función la cual nos retorna todas las trampas del tablero en las cuales sus posiciones son fijas.

```
private getTraps(): coordinates[] {
        return [
            [2, 2],
            [2, 5],
            [5, 2],
            [5, 5],
        ];
    }
```

get Traps image

Luego en cada ciclo del juego hacemos un recorrido por todas las casillas trampa del tablero, y si hay alguna pieza en alguna validamos si tiene una pieza amiga o no en las casillas adyacentes, en caso de no tener alguna pieza amiga a su lado, borramos esta pieza del tablero.

```
private deleteTrappedPieces() {
    this.getTraps().forEach((trap) => {
        const [x, y] = trap;
        const piece = this.getPieceAt(trap);

        if (piece) {
            const adjacentCells = piece.getAdjacentsMovements([x, y]);
            if (
                !adjacentCells.some((tile) => {
                    const cPiece = this.getPieceAt(tile);
                    return cPiece && cPiece.color === piece.color;
                })
            ) {
                this.board[x][y] = 1;
            }
        }
    });
}
```

delete trapped pieces image

Fin del juego

Una partida de arimaa no puede terminar en empate para esto hay 3 formas diferentes de ganar:

Objetivo

El objetivo principal del juego consiste en mover cualquier conejo amigo hasta la última posición del lado contrario, es decir si los conejos dorados aparecen en la posición length -1 de la matriz, entonces el jugador dorado gana cuando un conejo llegue a la fila en la posición 0 de la matriz, y al contrario para el jugador plateado,

```
private checkWinByRabbitsAtEnd(): void {
    if (this.currentPlayer.color === "gold" && this.currentPlayer.turns === 0) {
        const goldenRabbits = this.board[0].some((cell) => {
            return cell instanceof Rabbit && cell.color === "gold";
        });

        if (goldenRabbits) alert("Gold wins! Golden rabbits reached the end.");
    }

    if (this.currentPlayer.color === "silver" && this.currentPlayer.turns === 0) {
        const silverRabbits = this.board[this.board.length - 1].some((cell) => {
            return cell instanceof Rabbit && cell.color === "silver";
        });

        if (silverRabbits) alert("Silver wins! Silver rabbits reached the end.");
    }
}
```

check win by rabbit at end image

Eliminación

Se puede ganar cuando se captura al último conejo enemigo restante, para esto simplemente hacemos una búsqueda de los conejos en el tablero, si no existe ninguno gana el jugador contrario.

```
private checkWinByRabbitsTrapped(): void {
    const pieces = this.getAllPieces();

    const goldenRabbits = pieces.some((cell) => {
        return cell instanceof Rabbit && cell.color === "gold";
    });

    if (!goldenRabbits) alert("Silver wins! Gold rabbits are trapped.");

    const silverRabbits = pieces.some((cell) => {
        return cell instanceof Rabbit && cell.color === "silver";
    });

    if (!silverRabbits) alert("Gold wins! Silver rabbits are trapped.");
}
```

check win by elimination of rabbits image

Inmovilización

Se puede ganar haciendo un movimiento que deje a todas las fichas del oponente sin movimiento lo que implica un gran bloqueo, esto se valida obteniendo todas las fichas de un color y si todas se encuentran inmovilizadas entonces gana el jugador del color contrario, para esto hay una función auxiliar que va a validar si una ficha esta inmovilizada por congelación o porque no tiene disponible ningún tipo de movimiento (simple, pull o push).

```
isImmobilized(): boolean {
        if (this.isFreezed()) return true;
        console.log("isImmobilized", this.getSimpleMovements(), this.getPushablePieces(), this.getPullablePieces());
        const availableMovements = this.getSimpleMovements();
        const pushablePieces = this.getPushablePieces();
        const pullablePieces = this.getPullablePieces();

        return availableMovements.length === 0 && pushablePieces.length === 0 && pullablePieces.length === 0;
    }
```

is Immobilized image

```
private checkWinByImmobilization(): void {
    const pieces = this.getAllPieces();
    const silverPieces = pieces.filter((cell) => cell.color === "silver");
    const goldPieces = pieces.filter((cell) => cell.color === "gold");

    const ableGoldPieces = silverPieces.some((piece) => !piece.isImmobilized());

    if (!ableGoldPieces) alert("Silver wins! Gold pieces are immobilized.");

    const ableSilverPieces = goldPieces.some((cell) => !cell.isImmobilized());
    if (!ableSilverPieces) alert("Gold wins! Silver pieces are immobilized.");
}
```

check win by immobilization image

Construcción del árbol minimax

Para esto implementamos una permutación de todas las piezas de un color y simulando todos sus posibles tipos y direcciones de movimientos asegurándonos de no evaluar estados previamente calculados, con esto se crearon clones de las estancias del juego y de estas copias obtuvimos todas las copias de las fichas y se fue construyendo un camino que serian los pasos que dio la ficha, donde guardamos el tipo de movimiento y la posición. Utilizamos un arreglo de los tipos de movimientos y así validamos cada uno de estos. Por ejemplo, si me encuentro validando un movimiento simple haré la permutación de la ficha con el movimiento simple ignorando los otros 2 tipos de movimientos para esta copia.

Con todas las simulaciones de un estado de un tablero, hicimos la construcción del árbol minimax haciendo un cambio entre simulaciones empezando con max y luego para min se hace el cálculo para cada estado final de las simulaciones resultantes de max, luego cuando nos aseguramos que llegamos al límite de profundidad hacemos una evaluación de este nodo final para calcular su valor mediante las heurísticas.

Para la toma de la decisión minimax se hizo un método que recibe un nodo e itera sus nodos hijos de forma recursiva haciendo una búsqueda del mejor nodo siendo los de mayor valor para max y los de menor valor para min, asegurándose de no evaluar nodos ya visitados.

Referencias

Arimaa información y reglas
https://es.m.wikibooks.org/wiki/Arimaa/Informaci%C3%B3n/Reglas