

Práctica de PROP

Joan Manuel Ramos Refusta

Àlex Aguilera Martínez

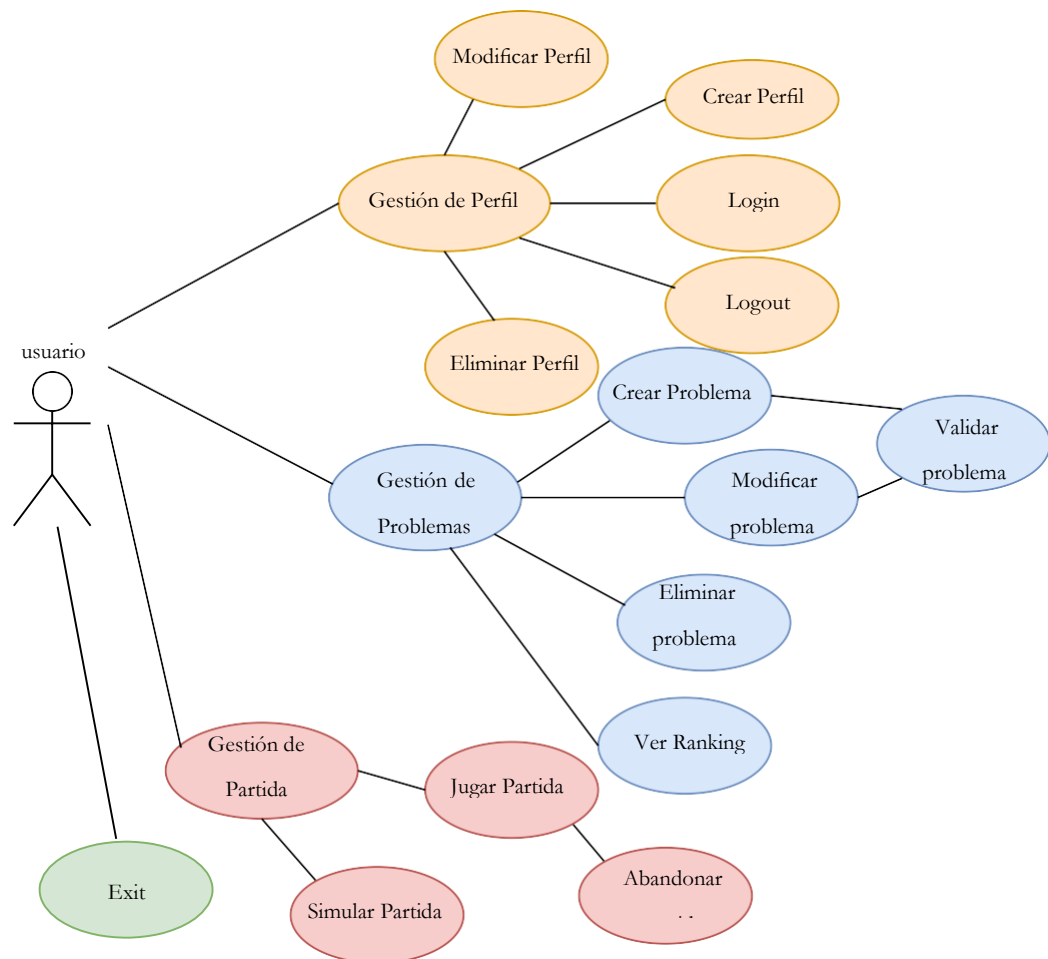
Jordi Santacreu Carranco

Versión 1.0

Índice

Índice.....	2
1-Diagrama casos de uso	3
2-Diagrama estático.....	7
3-Relación clases con miembros del grupo.....	10
4-Breve descripción de algoritmos y estructuras.....	11
4.1-Arraylist	11
4.2-List	11
4.3-Map	11
4.4-MinMax	11
4.5-Alpha-Beta	12

1-Diagrama casos de uso



Gestión perfil

El usuario puede gestionar su perfil, creando un nuevo perfil, iniciar sesión en una cuenta existente, modificando el perfil, eliminándolo o cerrar la sesión iniciada.

Modificar perfil

El usuario, una vez haya iniciado su sesión, puede modificar el nombre o la contraseña, siempre que los datos introducidos no sean iguales a los que tenía previamente. Si no se produce ningún error durante proceso, la aplicación nos dirá que el cambio se ha producido correctamente.

Crear perfil

El usuario crea un nuevo perfil válido, se introduce un nombre y una contraseña válidos, siempre que el nombre de usuario no haya sido creado previamente.

Login

El usuario introduce el nombre y contraseña de su perfil de usuario y si los datos son correctos iniciará sesión correctamente, en caso contrario, la aplicación nos informará de que alguno de los datos es erróneo.

Logout

El usuario desconectará su sesión y accederá de nuevo al menú de inicio de sesión.

Eliminar perfil

El usuario, una vez haya iniciado su sesión, puede eliminar su perfil, destruyendo así todos los datos registrados en la aplicación (ranking y problemas creados incluidos). Y al acabar se le informa de ello.

Gestión de problemas

El usuario puede gestionar los problemas, creando uno nuevo, modificándolo siendo el creador, puede ver el ranking de cada uno de los problemas, validar un problema cuando lo haya creado o modificado y eliminarlo.

Crear problema

El usuario crea un nuevo problema desde cero, escogiendo la dificultad, el tema y la posición y color de las fichas en el tablero. El problema nuevo no debe de existir previamente y se guardarán los datos del creador en dicho problema.

Modificar problema

El usuario modifica un problema siempre y cuando éste haya sido su creador y ningún usuario lo haya superado aún.

Eliminar problema

El usuario elimina un problema siempre y cuando éste haya sido su creador.

Validar problema

El usuario, una vez haya creado o modificado un problema, lo valida de forma que todos los usuarios ven el problema en la lista de problemas.

Ver ranking

El usuario puede acceder a los rankings de puntuación de cada problema existente. En ella verá todos los usuarios que han superado el problema ordenados de forma ascendente según la puntuación obtenida.

Esta puntuación se obtiene mediante el número de intentos (que será el número de veces que hacemos un movimiento erróneo, es decir, un movimiento que no conduce a la solución), y de ser igual este número de intentos, el tiempo que se haya empleado en conseguir la solución final desempatará.

Gestión partida

El usuario puede elegir entre simular una partida (entre máquinas) o jugar partido entre dos jugadores o contra una máquina.

Jugar partida

El usuario empieza una partida, seleccionando un problema a resolver, en caso de querer jugar contra otro jugador, el segundo jugador deberá de introducir sus datos. Una vez se haya comprobado que los datos sean correctos, la aplicación preguntará quién quiere atacar/defender (con el color) en el problema y dará a escoger entre Jugador1 o Jugador2 y empezará la partida.

En caso de que queramos jugar contra la máquina, una vez acabada la partida, si el jugador ha superado el problema, nuestra partida será evaluada por la puntuación del ranking y asumiremos un puesto en el mismo.

Simular partida

El usuario selecciona dos máquinas distintas y K problemas a superar por estas máquinas. La simulación hará que estas dos máquinas ataquen y defiendan los K problemas, de los problemas que hayan superado, se hará una evaluación y saldrá el ganador teniendo en cuenta la máquina que ha pasado más problemas.

Abandonar partida

El usuario, dentro de una partida, podrá abandonarla, volviendo al menú Inicial.

Exit

Salimos de la aplicación y hacemos Logout.

2-Diagrama estático

- **Jugador:** Es la clase implementada para jugar una partida, es una clase abstracta que tiene como subclases usuario y máquina necesarios para diferenciar si juega uno u otro.

Clases

- Pair Getnextmove(Problema p) :con esta función obtendremos donde se va a mover nuestro jugador , estas coordenadas serán validas i de esta manera el movimiento ser correcto. Esta operación es abstracta
- Get/setter del nombre y el color

- **Usuario:** Subclase de la clase jugador implementada para jugar por personas en nuestro juego de ajedrez y para gestionar los rankings de los problemas. Tiene como atributos nombre (heredada de jugador) y una contraseña para poder iniciar sesión.

Clases

-Get/setter de la contraseña

-Pair Getnextmove(Problema p) :con esta función obtendremos donde se va a mover nuestro jugador , estas coordenadas serán válidas i de esta manera el movimiento ser correcto

- **Máquina:** Subclase de la clase jugador implementada para la IA. Usada para la evaluación de K problemas y para que el usuario intente vencerla. Tiene como atributos nombre (heredada de jugador), dificultad (fácil o difícil) y una profundidad de cara al minimax. (algoritmo usado para la IA).

ClasFunciones

-Get dificultad : devuelve la dificultad.

--Pair Getnextmove(Problema p) :con esta función obtendremos donde se va a mover nuestro jugador , estas coordenadas serán válidas i de esta manera el movimiento ser correcto .

- **Ranking:** Es la clase implementada para guardar todos aquellos usuarios que superen un problema. Hay un ranking por problema. Este ranking ordena según el tiempo logrado por el usuario de manera que se tendrá siempre ordenado del menor al máximo tiempo.

Funciones

-Ordenar(): Esta función ordena nuestro hash map , esta ordenación la haremos por el valor del map i de manera ascendente , es decir , del más pequeño al más grande.

-Eliminarusuario(string nom) : Esta función eliminar del map el nodo el cual tiene como key la variable nom .

-Get/set para obtener sus atributos

- **Partida:** Es la clase implementada para poder jugar a un problema el cual se carga cuando iniciamos la partida. Esta cuenta con dos jugadores que pueden ser dos máquinas, dos jugadores o jugador contra máquina. El objetivo es lograr hacer mate en un problema en el número de movimientos que establece el mismo.

Funciones

-playJugadores(): Esta función la utilizaremos para crear una partida la cual juegan dos humanos a un problema , el primer jugador puede elegir su color y el otro obtiene el color que no ha escogido su rival , el usuario que gana es quien realiza un jaque mate en los números de movimientos preestablecido.

-playJugadorvsMaquina(): Esta función la utilizaremos para crear una partida entre un humano y una máquina , el objetivo de esta función es que el usuario pueda vencer a la máquina en los movimientos establecidos en el problema inicialmente .

-playMaquinavsMaquina(boolean validar):Esta función crearemos una partida donde enfrentaremos dos maquina con la misma dificultad , los colores de las máquinas han sido seleccionadas de forma aleatoria , la maquina que tenga el turno inicial será la que tiene mas posibilidades en ganar la partida.

mover(boolean color,String cord1,String cord2) : Está función nos dirá si el movimiento realizado es correcto o si el movimiento que estamos realizando nos ponemos en una situación de jaque o si la coordenada introducida es incorrecta.

Problema: Es la clase implementada para cargar nuestro tablero mediante el fen que proporcione el usuario. Además, tiene otros atributos como son el número de movimientos para hacer mate, el turno inicial (obtenido del fen) y el ranking de los usuarios que han intentado el problema. Tiene también, métodos necesarios para el momento de jugar como son el mate/checkmate y el mover ficha.

Función

-actualizarRanking(String nombre , double tiempo):Actualiza el tiempo del usuario cuando haya pasado el problema .

-introducirElemento(String nombre,double tiempo) : En esta función introduce una key y un valor al ranking.

-printTablero(): Esta función imprime la matriz generada por el fen que ha sido introducido anteriormente.

-matrixToFen(): Esta función convierte la matriz de fichas que tenemos como board lo pasamos a un string que será el fen

-FenToMatrix(string fen) : Esta función convierte el fen que le pasamos como parámetros a la matriz de ficha que tenemos declarada en la clase .

-moveFicha(String s1, String s2): dadas dos posiciones , mueve la ficha de coord. C1 a c2 siempre y cuando c2 se pueda acceder y no haya una ficha igual al color al que movemos y este dentro del tablero . Si hay una ficha rival en C2 , nos la comemos .

- **Ficha:** Es la clase implementada para hacer de superclase de todos los tipos de fichas del ajedrez y para fijar los posibles movimientos de las mismas. Es una clase abstracta ya que los diferentes tipos tienen posibles movimientos diferentes.

Funciones

- ArrayList<Coordenada> posiblesMovimientos(Problema p, Coordenada c): Dado un problema y una coordenada cualquiera dentro del tablero del problema, devuelve todos los posibles movimientos que puede hacer una ficha según que tipo sea.

- Get/set para devolver o poner valores a los atributos de la clase

- **Peón:** Subclase de la clase ficha que implementa los posibles movimientos de una ficha peón del ajedrez: moverse dos en caso de estar en la segunda/penúltima fila, avanzar uno o comer otra ficha en diagonal.

Funciones

- ArrayList<Coordenada> posiblesMovimientos(Problema p, Coordenada c): Dado un problema y una coordenada cualquiera dentro del tablero del problema, devuelve todos los posibles movimientos que puede hacer una ficha según que tipo sea.

- Get/set para devolver o poner valores a los atributos de la clase

- **Reina:** Subclase de la clase ficha que implementa los posibles movimientos de una ficha reina del ajedrez: básicamente puede hacer la suma de los movimientos de un alfil y de una torre, es decir, se puede mover tanto de manera horizontal como vertical o diagonal. Es la ficha más poderosa.

Funciones

- ArrayList<Coordenada> posiblesMovimientos(Problema p, Coordenada c): Dado un problema y una coordenada cualquiera dentro del tablero del problema, devuelve todos los posibles movimientos que puede hacer una ficha según que tipo sea.

- Get/set para devolver o poner valores a los atributos de la clase

- **Caballo:** Subclase de la clase ficha que implementa los posibles movimientos de una ficha caballo del ajedrez: se puede mover dos casillas horizontalmente y una casilla verticalmente o dos casillas en posición vertical y una horizontal cuadrada. Es decir, haciendo Ls por el tablero.

Funciones

- ArrayList<Coordenada> posiblesMovimientos(Problema p, Coordenada c): Dado un problema y una coordenada cualquiera dentro del tablero del problema, devuelve todos los posibles movimientos que puede hacer una ficha según que tipo sea.

- Get/set para devolver o poner valores a los atributos de la clase

- **Alfil**: Subclase de la clase ficha que implementa los posibles movimientos de una ficha alfil del ajedrez: se puede mover sobre el tablero en una línea recta diagonal.

Funciones

- ArrayList<Coordenada> posiblesMovimientos(Problema p, Coordenada c):Dado un problema y una coordenada cualquiera dentro del tablero del problema , devuelve todos los posibles movimientos que puede a ver una ficha según qué tipo sea .

-Get/set para devolver o poner valores a los atributos de la clase

- **Torre**: Subclase de la clase ficha que implementa los posibles movimientos de una ficha alfil del ajedrez: se puede mover en línea recta con movimientos horizontales y verticales sobre el tablero.

Funciones

- ArrayList<Coordenada> posiblesMovimientos(Problema p, Coordenada c):Dado un problema y una coordenada cualquiera dentro del tablero del problema , devuelve todos los posibles movimientos que puede a ver una ficha según qué tipo sea .

-Get/set para devolver o poner valores a los atributos de la clase

- **Rey**: Subclase de la clase ficha que implementa los posibles movimientos de una ficha rey del ajedrez. Es la pieza fundamental del tablero solo se puede desplazar en una casilla tanto en vertical, horizontal o en diagonal. Si el rey puede ser comido por otra pieza es jaque y si no puede hacer movimiento para evitarlo es jaque mate.

Funciones

- ArrayList<Coordenada> posiblesMovimientos(Problema p, Coordenada c):Dado un problema y una coordenada cualquiera dentro del tablero del problema , devuelve todos los posibles movimientos que puede a ver una ficha según qué tipo sea .

-Get/set para devolver o poner valores a los atributos de la clase

- Hemos implementado además algunas clases extra como son **Minimax**, **Coordenada** y **Pair** por necesidad y para el correcto funcionamiento de nuestro programa. Aunque Coordenada y Pair sean clases muy similares, hemos optado en implementar las dos, debido a que Coordenada necesita dos métodos que son StringToCoordenada y CoordenadaToString para representar de manera más entendible el tablero de nuestro programa con letras y números

3-Relación clase con miembros del grupo

La relación entre las clases con los miembros del grupo ha sido la siguiente:

- Àlex: Ha hecho todas las clases relacionadas con las fichas del tablero y la clase MiniMax la cual encontramos el algoritmo que usamos para jugar contra otro usuario (IA), la clase Máquina y sus respectivos drivers.
- Joan: Ha hecho la clase Problema, la clase Jugador, los controladores de problemas y usuarios y también sus respectivos drivers.
- Jordi: Ha hecho la clase Ranking, Partida, Jugador, el JUnit y sus respectivos drivers. Al igual que las clases extra implementadas Coordinada y Pair con sus drivers.

4-Breve descripción de los algoritmos y estructuras

4.1-Arraylist

Es una clase que permite almacenar datos en memoria de manera similar a la de un array con la ventaja de que es una estructura flexible que permite añadir elementos sin tener en cuenta su tamaño. Permite añadir y eliminar atributos del ArrayList siempre que el usuario quiera. Algunas funcionalidades; para añadir un nuevo elemento se hace con `array.add(variable)` y si queremos eliminar un elemento lo haremos mediante `array.remove (atributo)`, para saber el tamaño del ArrayList en un momento dado se usa `array.size()`...

4.2-List

Son un tipo de estructura que permite almacenar grandes cantidades de datos. Proporciona control sobre la posición en la que puede insertar un elemento y permite acceder a los elementos por su índice y también buscar elementos. Algunas funcionalidades; para añadir elementos haremos `listnom.add (elemento)` y para eliminar `listnom.delete(elemento)`. Es, además, una colección ordenada motivo por el cual ha sido usada en el ranking para ordenar.













4.3-Map

El map es una estructura ordenada que nos permite almacenar pares "clave/valor"; de tal manera que para una clave solamente tenemos un valor. La hemos utilizadi tanto para hacer el ranking en nuestro programa como los `CtrlProblemas` y `CtrlUsuarios`. Es una estructura que permite ordenar sus claves de forma automática si lo deseamos, pero en este caso tiene la peculiaridad que también permite ordenar por valor mediante un comparador, esta característica nos ha sido bastante útil en el momento de implementar nuestro ranking.

4.4-MinMax

Es el algoritmo utilizado en esta práctica para poder implementar nuestra máquina (IA). El algoritmo MiniMax es el algoritmo más utilizado para problemas con exactamente 2 adversarios y movimientos alternos. Básicamente, consiste en identificar a cada jugador como MIN y como MAX, este último es el que siempre inicia el juego, y marcar como objetivo encontrar el conjunto de movimientos que proporcione la victoria a MAX (máquina) suponiendo que el contrincante (MIN) escogerá los peores para MAX.

Es necesario una función de evaluación heurística que devuelva valores elevados para las buenas situaciones y valores bastante peores para las situaciones favorables para el oponente, de esta manera cada movimiento tendrá un valor numérico y podremos escoger el mejor de todos los movimientos. Nosotros, hemos escogido los siguientes valores para cada una de las fichas de nuestro tablero:

	10		-10		50		-50
	30		-30		150		-150
	30		-30		900		-900

Además de utilizar esta función heurística, se usa una estrategia DFS de profundidad limita (ya que partir de 4, se empieza a notar el coste exponencial del algoritmo) para explorar las jugadas posibles. Lo que quiere decir, que, al no poder estudiar todas las posibles situaciones hasta el fin de partida, puede que haga alguna jugada que sea bastante mala. El pseudocódigo es el siguiente:

```

Función valorMin(estado) retorna entero
    valor_min:entero;
    si estado_terminal(estado) entonces
        retorna evaluación(estado);
    sinó
        valor_min := +infinito;
        para cada mov en movimientos_posibles(estado) hacer
            valor_min := min(valor_min, valorMax(aplicar(mov, estado)));
        fpara
        retorna valor_min;
    fsi
fFunción

Función valorMax(estado) retorna entero
    valor_max:entero;
    si estado_terminal(estado) entonces
        retorna evaluación(estado);
    sinó
        valor_max := -infinito;
        para cada mov en movimientos_posibles(estado) hacer
            valor_max := max(valor_max, valorMin(aplicar(mov, estado)));
        fpara
        retorna valor_max;
    fsi
fFunción

Función MiniMax(estado) retorna movimiento
    mejor_mov: movimiento;
    max, max_actual: entero;
    max = -infinito;
    para cada mov en movimientos_posibles(estado) hacer
        max_actual = valorMin(aplicar(mov, estado));
        si max_actual > max entonces
            max = max_actual;
            mejor_mov = mov;
    fsi
    fpara
    retorna mejor_mov;
fFunción

```

Esta ha sido la opción escogida de cara a la máquina de nivel fácil, ya que es la más sencilla y no es la opción más inteligente. También añadir que en nuestro código nos hemos ahorrado usar las funciones min y max, haciéndolo en una única función recursiva.

4.5-AlphaBetaMinimax

Por lo dicho anteriormente, se puede afirmar que la calidad de nuestras jugadas y por tanto “inteligencia” de la IA, vendrá determinada por la profundidad a la que lleguemos

en cada exploración. Es por ello, que la idea es implementar mejoras para poder tener más espacio de búsqueda, una optimización es el AlphaBetaMinimax.

Este consiste en una técnica llamada ramificación y poda con la cual abandona una solución parcial en el momento que se sepa que hay otra solución mejor.

Concretamente, se guardan dos valores denominados alfa y beta, de ahí el nombre.

Alfa representa la cota inferior del valor que puede asignarse en último término a un nodo de MAX, y mientras que beta representa la cota superior del valor que puede asignarse como última opción en un nodo de MIN.

La función minimax (minimaxDecision, en nuestro código) se mantiene igual y cambiaría las funciones max y min, ya que debe evaluar alfa y beta de la siguiente manera:

```
si alfa >= beta entonces
    retorna ( beta )
fsi
```

Esta es la opción escogida de cara a la máquina con dificultad difícil. Pero, intentaremos de cara a la segunda entrega mejorar aún más la profundidad del AlphaBeta con otras opciones que hemos visto por Internet.