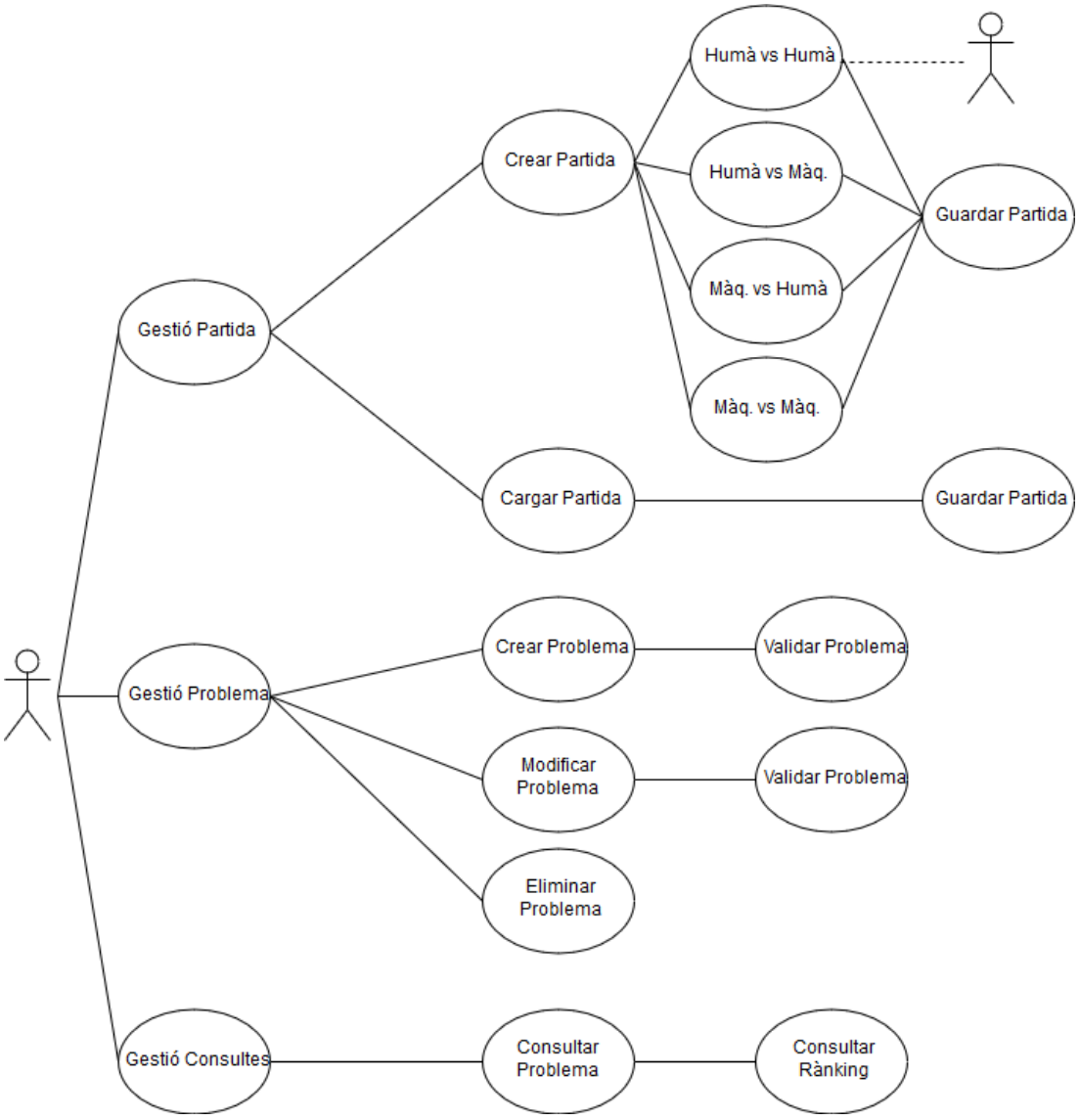
1ª entrega PROP

Jaume Malgosa, Calin-Constantin Pirau i Pau Charques

Maig 2019

Índex

1 Casos d'ús


Humà vs Humà
Crear Partida
Humà vs Màq.
Guardar Partida
Màq. vs Humà
Gestió Partida
Màq. vs Màq.
Cargar Partida
Guardar Partida
Crear Problema
Validar Problema
Gestió Problema
Modificar Problema
Validar Problema
Eliminar Problema
Gestió Consultes
Consultar Problema
Consultar Ràning

1.1 Descripció

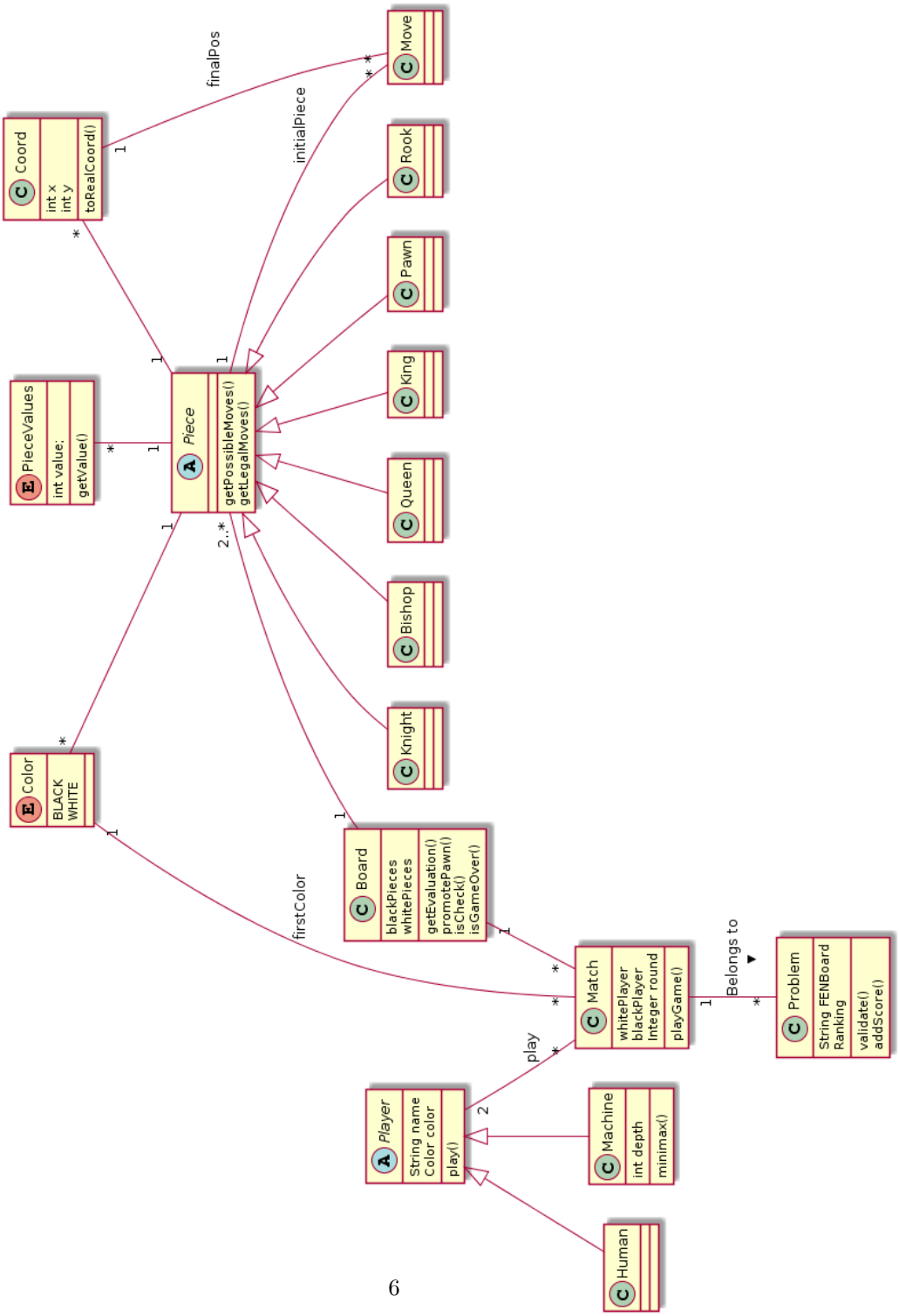
Els casos d'ús es divideixen principalment en tres parts: la gestió de la partida, que permet jugar a escacs, la gestió del problema, que ajuda a crear, modificar o eliminar problemes, i la gestió de les consultes, que crea la possibilitat de consultar tots els problemes, de consultar un problema en concret (i veure si es vol el seu Ranking), i consultar els jugadors de la base de dades. A continuació s'expliquen en detall:

- **Gestió Partida**
  - **Crear Partida**
    - Humà vs Humà: En aquest cas es dóna la possibilitat que dos jugadors humans puguin jugar. El programa demanarà les coordenades de la casella que contingui una peça del color del jugador i les coordenades de la casella on vulgui moure la peça, tot això fet en ambdós torns.
    - Humà vs Màq.: A l'humà li demanaran les coordenades inicials i de moviment i la màquina calcularà quin es el millor moviment que pot fer, a partir de l'algoritme minimax.
    - Màq. vs Humà: És el mateix que en l'apartat anterior però amb els colors canviats
    - Màq. vs Màq.: Ambdues màquines calculen els seus moviments, per tant l'usuari humà només podrà fer d'espectador.
    - Guardar Partida: Quan es surti de la partida, el programa la guardarà.
  - **Cargar Partida**
    - Guardar Partida: Quan es surti de la partida, el programa la guardarà.
- **Gestió Problema**
  - Crear Problema: Per crear un problema es demana primer la notació FEN de la taula d'escacs, seguit de quants moviments podrà fer el jugador per resoldre el problema. Seguit d'això el programa validarà si es pot resoldre el problema inserit
    - Validar Problema: Per poder validar el problema s'utilitza força bruta per poder veure si realment el jugador pot resoldre el problema.
  - Modificar Problema: Per modificar el problema el programa donarà el FEN existent a l'usuari i l'usuarí podrà editar la notació FEN, sempre i quan es pugui validar la seva modificació.
    - Validar Problema: Es valida amb força bruta el problema modificat.

  - Eliminar Problema: Es demanarà el ID del problema a eliminar, i s'eliminarà el problema i el seu Ranking.

- **Gestió Consultes**
  - Consultar Problema: Es demanarà el ID específic per veure un problema en concret.
    - Consultar Ranking: Després de consultar el problema, es pot consultar el seu Ranking.

2 Model UML


Coord
int x
int y
toRealCoord()
finalPos
initialPiece
Move
Rook
Pawn
King
Queen
Bishop
Knight
PieceValues
int value;
getValue()
Piece
getPossibleMoves()
getLegalMoves()
Color
BLACK
WHITE
Board
blackPieces
whitePieces
getEvaluation()
promotePawn()
isCheck()
isGameOver()
firstColor
Match
whitePlayer
blackPlayer
Integer round
playGame()
Belongs to
Problem
String FENBoard
Ranking
validate()
addScore()
play
Player
String name
Color color
play()
Machine
int depth
minimax()
Human

2.1 Descripció

El nostre diagrama l'hem fet pensant a entendre millor l'estructura feta en Java, per això hem afegit què tipus de classe en Java és cada part. Les classes que tenen una E són enums, les que tenen una A són abstract class, i les que tenen una C son classes normals. A continuació explicarem breument cada classe a part.

- **Color:** Color és una enumeració que s'usarà com atribut per determinar un equip. Conté els valors "WHITE" o "BLACK".
- **Coord:** Coord és una estructura formada per dos enters que determinen la posició en el taulell de joc.
- **PieceValues:** PieceValues és una enumeració que determina quin valor té cada peça dins del joc. S'usa tant per calcular la puntuació del guanyador com perquè els algorismes calculin els millors moviments.
- **Piece:** La classe Piece conté tota la informació de cada una de les peces del joc. Conté la posició, determinada per la classe Coord, l'equip, determinat per l'enumeració Color, i el seu valor, que prové de l'enumeració PieceValues. Aquesta classe s'estén en 6 classes filles, una per cada tipus de peça que existeix als escacs.
- **Move:** La classe Move està formada per una posició, continguda dins d'una Coord, i serveix per moure les peces. El moviment el determina la posició de l'atribut i la posició de la peça que es vol moure.
- **Board:** La classe Board està formada per un conjunt de peces, segons el problema, amb dos ArrayList. Cada ArrayList conté les peces de cada equip.
- **Match:** La classe Match està formada per un Board, un Color que determina a quin jugador li toca jugar, dos Players i una Score, que guardarà la puntuació del guanyador, i el Problem a solucionar.
- **Player:** La classe Player és la que conté el nom en forma de String i amb quin equip juga, determinat amb l'enumeració Color. Segons el tipus de jugador Humà o màquina es divideix en dues subclasses: Human: que llegeix els moviments de la terminal; Machine: que conté l'algorisme per trobar els millors moviments per guanyar.
- **Score:** La classe Score guarda el nom del jugador guanyador del Match i la seva puntuació.
- **Ranking:** La classe ranking està formada per les 5 millors Score d'un Problem en concret. Quan arriba una Score nova i és superior a qualsevol de les que hi ha a Ranking, la més petita guardada s'elimina i es posa la Score nova a la posició que li correspon.

- **Problem:** La classe Problem està formada per un FEN, la dificultat del problema, qui comença jugant i dues ArrayList amb la posició de les peces de cada equip, així com la quantitat de torns per fer el mat (N).

2.2 JUnit

Per provar les utilitats de JUnit (eina de test), hem aplicat les seves tècniques de programació.

JUnit permet programar a partir d'una sèrie de tests, a mesura que vas escrivint el test amb les sortides i crides a funcions desitjades es va generant el codi, evitant el hardcode. Es a dir, deixa que el JUnit et vagi dient el que et fa falta. D'aquesta manera cada petita porció de codi està provada en tot moment. Separant els tests per seccions permet identificar ràpidament els errors i solucionar-los.

En el nostre cas hem fet ús d'aquesta eina amb la classe Ranking i la classe Score de forma indirecta. Al principi vam definir els tests, no gaires, ja que la classe Ranking funciona més com una memòria que com a un codi a executar, i definint un atribut Ranking format per l'stub de Scores, d'aquesta manera asseguràvem que si hi havia un error era a Ranking. Quan vam aconseguir que funciones tot, vam canviar el stub per scores reals i executar els tests de nou.

No va haver-hi cap error, ja que Ranking estava ven programada. Amb això ens vam assegurar la correctesa de Ranking i Score.

3 Estructures de dades i algoritmes

L'estructura de dades utilitzada majoritàriament ha sigut la ArrayList, gràcies a la seva simplicitat i per les funcionalitats que conté. Ha sigut utilitzada amb més freqüència en la nostra classe Coord, per exemple per crear una ArrayList de tots els possibles moviments d'una peça en concret). Per guardar els problemes, jugadors i rankings s'ha utilitzat JUnit.

En quant a algorismes, els més importants han sigut el Minimax, utilitzat en l'algoritme del jugador màquina, i el Backtracking, utilitzat en el validador de problemes. A continuació s'expliquen els algoritmes amb més detall.

3.1 Minimax

Minimax és un mètode de decisió que s'utilitza per minimitzar les pèrdues i maximitzar les guanys del jugador en qüestió, això sabent que el joc és amb adversari i amb informació perfecta. Minimax és un algoritme recursiu, i es pot resumir en què busca el millor moviment sabent que el contrincant farà el pitjor moviment per al jugador.

En el nostre cas minimax compleix tots els requisits: és un joc amb adversari i té informació perfecta, o sigui no hi ha res abstracte, per tant el podem aplicar.

Per poder avaluar qui està guanyant en un cas determinat, s'ha de puntuar l'estat actual de la taula d'escacs, sumant el valor de les peces que queden blanques i restant el valor de les peces que queden negres. S'ha afegit una heurística per donar un valor en concret a cada peça, que són les següents:

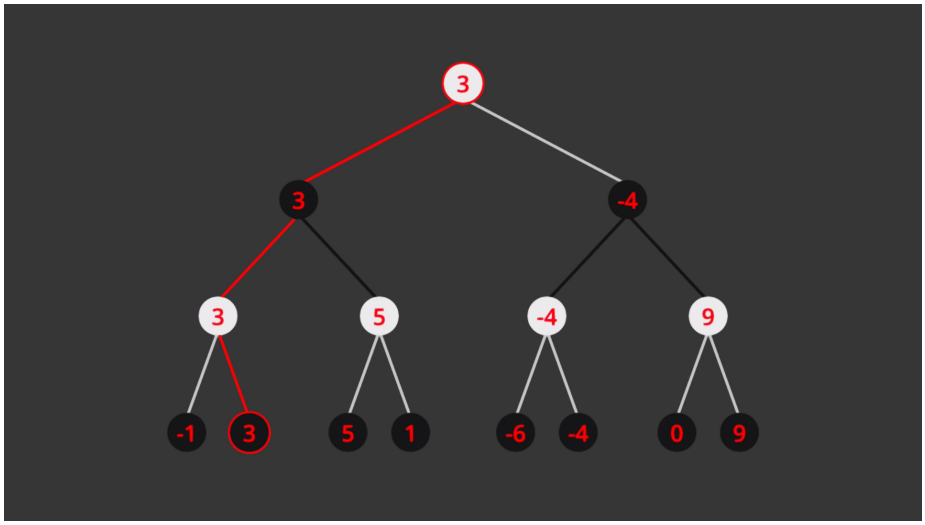
- Peó: 1 punt
- Torre: 5 punts
- Alfil: 3 punts
- Cavall: 3 punts
- Reina: 9 punts
- Rei: 900 punts

A més, s'ha afegit una funcionalitat que verifica si hi ha una situació en què s'hagi produït escac i mat, per poder així sumar 900 punts si el que està en escac i mat és el color negre, i restar 900 punts si és a l'inrevés.

Sabent com es puntuen els taulers, ara es pot fer que la màquina pugui prendre les seves pròpies decisions. Aquest algoritme incorpora una variable depth (profunditat), que s'utilitza per poder fer tots els possibles casos de moviments en un nombre limitat de rondes, que en aquest cas seria la profundidat. Per tal de fer-ho, es comença amb totes les peçes que tingui la màquina i tots els

possibles moviments que pugui fer cada peça, i es mouen imaginàriament a la taula. Així ja estaríem passant de profunditat 0 a 1. Si la profunditat és major o igual que 1, ara seria el torn del color contrari per a moure una peça imaginàriament. La màquina suposarà que l'adversari farà el pitjor moviment possible per a ella, ja que així guanyaria ell. Per tant, es fa com el primer torn, s'agafen totes les peces de l'adversari amb tots els seus moviments possibles i es fan moviments imaginaris. Aquest procés es repeteix fins que s'arriba a la profunditat màxima (perquè si no el temps de càlcul podria arribar a ser infinit), que és quan s'avalua el tauler. Amb la puntuació resultant, es retorna a la funció pare (ja que minimax es recursiu), i el mateix decideix quina és la millor jugada: si havia sigut el torn del color blanc, la puntuació més alta de tots els moviments fets és l'escollida; si el color que havia mogut era el negre, la puntuació més baixa és l'escollida. Finalment, quan arriba a la funció principal, s'agafa el moviment que més ajudi al color que tingui el torn.

A continuació s'adjunta una imatge amb un exemple d'arbre construït amb minimax (simplificat):



Les línies pintades de vermell indiquen el camí que ha fet el número 3 per arribar a ser el millor moviment. Es pot veure com el color blanc agafa sempre els moviments amb més puntuació i el color negre agafa els moviments amb menys puntuació.

3.2 Backtracking

Backtracking és un algoritme general que té com a funció trobar totes les possibles solucions d'un problema en concret. Aquest algoritme s'utilitza majoritàriament en problemes de prova i error, problemes que no tenen una solució mecànica o trivial. El major inconvenient del Backtracking és que depenent de la situació pot ser un algoritme lent, ja que el seu cost d'execució és exponencial.

En el nostre cas l'algoritme de Backtracking l'hem fet servir per poder validar els problemes que es creen o que es modifiquen. El que es busca en concret és que el problema creat (o modificat) es pugui solucionar en els N moviments designats pel problema. Si hi ha almenys un cas en el qual es pot resoldre, l'algoritme donarà llum verda al problema, i el mateix es guardarà a la base de dades. Si de tots els moviments que es puguin fer en els N del jugador principal no es pot fer escac mat el problema no té solució, i per tant no es guardarà en la base de dades.