Estructures de dades i algorismes.

Estructures de dades.

De cara a la versió final del projecte hem utilitzat les següents estructures de dades:

Point:

Hem utilitzat aquesta estructura per representar una posició dins del tauler. En aquest sentit, Point ens és molt útil ja que guarda exactament dos enters (x, y). Aquesta estructura és fàcil d'accedir i de modificar i no té cap mena de cost associat.

HashSet<Point>:

Utilitzem aquesta estructura de dades a mode de vector dinàmic per mantenir diversos atributs d'un tauler (posicions on poden jugar les blanques, posicions on poden jugar les negres, posicions adjacents al graf de fitxes però on no hi pot jugar ningú).

L'hem necessitat ja que aquest nombre de posicions varia al llarg d'una partida. Amb aquesta finalitat vam pensar primer en utilitzar una ArrayList, no obstant ens vam adonar de que el HashSet té un menor cost (cost constant) de cerca i d'eliminació, cosa que ens és molt útil a l'hora de modificar els atributs prèviament mencionats cada cop que es col·loca una fitxa.

Char[ ]:

Amb la finalitat de representar un tauler hem utilitzat un array de chars de 64 posicions que en alguns punts del codi l'hem convertit en una matriu de 8x8 per motius de facilitat d'accès. Com que només hem de fer modificacions indexades per posició els cost de totes les operacions en un tauler és constant. Cada char representa el contingut d'una casella ('W','B','E').

SortedSet<entrada_ranking>:

Necessitem aquesta estructura per mantenir ordenat el ranking. Conté instàncies de entrada ranking, una classe que vem implementar amb la interfície comparable per poder ordenar-la automàticament (entrada_ranking representa la puntuació i les partides guanyades i perdudes d'un jugador). El sorted set té un cost de log(n) per les operacions bàsiques, per tant l'ordenació del rànking és molt ràpida.

Queue<Point>:

Utilitzem aquesta estructura per implementar un BFS a la classe Board. D'aquesta manera comprovem si totes les fitxes són adjacents. La usem perque és l'estructura més adequada per implementar aquest tipus de cerca. Té cost constant en totes les operacions.

Algorismes.

Disposem de dos algorismes:

Minimax:

Aquest algorisme es basa en la cerca de la millor jugada per a un jugador (maximizer) suposant que el seu oponent (minimizer) farà el millor moviment possible. L'algorisme calcula tots els taulers possibles a partir d'una posició donada. Per calcular aquestes posicions fa servir la funció de Board available_positions() que donat un color retorna un HashSet amb totes les posicions on pot jugar aquest color. Aquest càlcul no té impacte en el rendiment ja que no s'han de calcular les posicions sinó que aquests HashSet s'actualitzen desde Board a cada tirada.

Es calculen tots els taulers possibles fins a arribar a la profunditat que se li indica(cada moviment és un nivell més de profunditat). Un cop arribat a aquesta profunditat l'algorisme evalua els taulers generats mitjançant una funció heurística, que té un valor proporcional a la probabilitat de guanyar del maximizer.

La funció heurística que hem implementat es basa en quatre aspectes: la mobilitat de cada jugador(moviments possibles), el control de les cantonades, el control de les vores del tauler i l'estabilitat de les fitxes. En concret, un moviment possible suma 1 punt, el control de cada cantonada suma 10 punts, les vores sumen 0.05 punts i l'estabilitat de les fitxes és igual a estabilitat/100. Estabilitat pot ser 1 si la fitxa és estable, 0 si la fitxa és semiestable i -1 si la fitxa és inestable. Una fitxa és inestable quan pot ser capturada al següent torn, semiestable quan es pot capturar en dos torns i estable en qualsevol altre cas. El càlcul més costòs d'aquesta funció és el de l'estabilitat, ja que per cada fitxa ha de comprovar si es pot capturar. Els altres càlculs no tenen impacte en el rendiment ja que s'utilitza la longitud del HashSet que retorna available_positions per als moviments possibles i la funció contains de Board per saber el contingut de les cantonades i les vores (cost constant).

Un cop obtinguda la heurística de tots els taulers generats l'algorisme selecciona la que té un major valor al torn del maximizer i un menor valor al torn del minimizer, així recursivament aconseguirà trobar el moviment òptim per a la posició actual.

Minimax amb podes:

Aquest algorisme està basat en el Minimax amb la diferència de que manté dues variables (alpha i beta) per tal d'estalviar temps de computació. Alpha guardarà la heurística més gran fins al moment, mentre que beta guardarà la més petita. D'aquesta manera, tant si estem en el torn del maximizer com en el del minimizer, si beta és menor o igual que alpha significa que continuar per aquesta branca no serviria ja que s'ha trobat un millor moviment prèviament, per tant aquesta branca mai seria escollida. Tenint això en consideració quan trobem aquesta condició parem la cerca, la "podem". La resta de característiques (funció heurística i funcionament principal) són idèntiques a la funció Minimax.