Estructures de dades i algorismes

Estructures de dades

Matriu de peces -> Atribut del taulell -> Peca[][] Tauler
Les peces tenen 2 atributs ArrayList que són les posicions amenaçades per la peça i les posicions de les peces que l'estan amenaçant a ella.

En la classe Estadistica, s'utilitza ArrayList en la majoria dels casos de Strings, per poder guardar una línea completa del fitxer en el mateix ordre (idProblema idUsuari mat temps), estructura escollida perquè les dades es van guardant quan es troba, per tant, es necesita una estructura de mida variable. Aquesta estructura també s'utilitza per guarda la classe privada Marca, per poder ordenar segons dos valors, la qual després es converteix a una llista.

Entrega 2

En la classe CrearProblema s'utilitzen 2 matrius: una de Strings per guardar els tipus de les peces que es mostren en el tauler i l'altre per guardar els colors d'aquestes peces.

En la classe Problema es guarda una matriu de booleans per provar primer els moviments que fan escac. Es guarda a cada fila un array de boolean per cada moviment de una peça i hi ha una fila per cada peça del color que toca moure.

Optimització del escac mitjançant la revisió de ArrayList de amenacis i amenaçades.
Ús / actualització dels ArrayList de amenaces i amenaçades de cada peça en moure el tauler.

Algorismes

validar_problema

Mètode de la classe Problema, valida si un problema es pot resoldre en N jugades. Això vol dir que ha d'existir com a mínim una seqüència de moviments del jugador que comença (atacant) de manera que si es fan, no hi ha cap combinació de moviments del jugador que defensa que pot evitar el mat. Per comprovar-ho el mètode aplica un algorisme de força bruta: mira totes les peces del jugador atacant i per cadascuna d'elles prova tots els moviments possibles de la peça. Per cadascun d'aquests moviments prova tots els moviments vàlids de la defensa i si algun d'aquests fa que el problema no tingui solució en N-1 jugades (crida recursiva amb el tauler després de fer els moviments dels 2 jugadors) aleshores es descarta el moviment del atacant i es passa a provar el següent. Si es proven tots els moviments de la defensa i amb tots el problema amb N-1 jugades segueix sent vàlid aleshores es determina que el problema és vàlid i deixen de provar més moviments de l'atacant.

En aquesta segona entrega, per millorar l'eficiència primer es proven tots els moviments que fan escac(que normalment són els que portaran al mat). Per fer-ho, en comptes de fer un primer recorregut sobre els moviments per ordenarlos, simplement es deixen per després els que no fan escac i són vàlids, marcant-los en un array de booleans.

escac

Funció que usa un algoritme de referenciació/desreferenciació de les direccions de les peces que es troben a la matriu de peces a l'hora de poder simular un moviment fictici del rei, guardant en peces temporals les referències de les peces que antigament es trobaven en aquelles posicions per tal de restaurar-les un cop s'hagin acabat les comprovacions. Podem aconseguir això sense machacar realment les peces que es troben a les diferents posicions gràcies al llenguatge, doncs ens ofereix un sistema d'allotjament dobjectes el qual es basa en punters i en emmagatzemar els objectes referenciats en el heap.

generar_situació

Funció que usa un algorisme similar, però no està pensada per donada una posició destí del rei, simular aquest moviment i després restaurarlo, sinó que aquesta funció es capaç de simular totes les possibles posicions destí de les peces d'un color, i a continuació, després de simular la situació s'avalua el arraylist d'amenaces del rei del color especificat. A diferència de la funció escac, aquesta usa els arraylist d'amenaces i amenaçades, i per tant aquest algorisme funciona fent una deep copy del taulell passat com a paràmetre implícit (totalment independent) però no recalcula les amenaces de tot el taulell sino que invoca a la operació pública mover pieza que ja s'encarrega de simular el moviment en cas que aquest sigui possible i així no hem de restaurar cap estat inicial del taulell simplement el tornem a machacar quan volguem repetir el procés.
La funció escac hauria d'assimilar-se més a generar_situacio doncs considero que es bastant més eficient, però el fet que fos més complexa al requerir dels arraylist d'amenaces, per això per a la segona entrega, escac estarà refactoritzada emprant també les estructures d'amenaces com fa generar_situacio.

minimax

L'algoritme Minimax de la teoria de jocs és un tipus de 'backtracking' amb una profunditat limitada i a on a cada node se li dóna un valor. Segons el valor del node els moviments de les peces pròpies es voldrà maximitzar i l'oponent intentarà minimitzar el guany que es pot obtenir. El seu principal objectiu és fer escac i mat a l'adversari, en cas de no poder arribar a la profunditat necessari per fer-ho intenta trobar l'estat del taulell que més el benefici. En alguns casos per aconseguir fer mat s'han de sacrificar peces o fer jugades que de primera no semblen les més òptimes, per tant no sempre és possible arribar al mat en el mínim de jugades.
L'algorisme parteix de la funció moviment, la qual fa el primer moviment i es crida a minimax, per tant, la profunditat real es la que es dona més un. Un cop fet el primer

moviment s’entra en minimax i es comprova que el taulell no està en un estat de final de partida o que és l'últim torn de la máquina, si no és el cas es comproven totes les peces del tauler, si es vol maximitzar llavors es faran moviments sobre les peces de la màquina, es crida a minimax (especificant que el següent torn es de l’oponent i per tant s’ha de minimitzar) el qual retornarà un valor(trobat en el node de la profunditat inferior) el cual si és major al màxim de tots el moviment d’aquell estat s’agafa com nou màxim i finalment es desfà el moviment. Al completar tots els moviments en minimax en la funció moviment s'escollirà el que dona un major valor.

alfa-beta

Es basa en el mateix principi que el ‘Minimax’, però amb certes millores de rendiment que permet que la resposta a un cert problema sigui més acurada.
S’aplica la tècnica de la poda alpha-beta, la qual redueix el número de nodes avaluats per l’algoritme de minimax, es basa que si arribes a un node i es troba que segons el valor d’aquest ja saps que ja tens un valor que maximitzant o minimitzant no serà l’escollit. A més per augmentar les podes que fa l'algorisme s’ordenen els moviments abans de probarlos, aquesta es una ordenació simple per evitar que el temps de càlcul d’aquest al final no empitjori, on es posen els moviments que fan escac com a primers moviments a probar.
Per tant, amb aquest algorisme es redueixen el nombre de nodes a evaluar i permet tenir una profunditat major a la del minimax, a més de tenir una augmenta d'eficiència i reducció del temps d’espera.

Threads

Threads → en fer pausa si el torn actual pertany a la màquina, mitjançant l'ús de les funcions thread.wait () i thread.notify () encapsulades dins d'un bloc syncronized (thread) per per / reprendre l'execució del fil secundari que executa el moviment de la màquina.
Per a la simulació, a més es posa aquest thread com un thread amb la màxima prioritat per a dedicar més temps de CPU a aquest fil, així aprofitant la planificació del SO i fent executar el thread secundari amb més freqüència.
Per implementar la funcionalitat de tornar al menú dins de la simulació, ús thread.interrupt () i dins el thread, faig check corresponent (if (thread.isInterrupted ()) {/ * finish thread mitjançant l'ús d'un flag * / }