Projectes de Programació

Othello

Estructures de dades i algorismes

subgrup-prop5.1 - Versió 1.0

Albert Bernal: @albert.bernal.macias

Oriol Marco: @oriol.marco.deumal

Adrián Rubio: @adrian.rubio.i

31/05/2021

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

ÍNDEX

1. Descripció de les estructures de dades i algorismes utilitzats

La breu descripció dels atributs i mètodes de cada classe de la capa de domini es troba al Javadoc.

1.1. Estructures de dades

L'estructura de dades principal és la que representa un taulell d'Othello. Aquest està descrit per una estructura de dades senzilla, *Cell*[][] *map*, un array de cel·les de tamany (*height*+2)*(*width*+2) que comptarà amb un valor per cada cel·la referent al seu estat d'entre els següents:

- **Border**, si es tracta d'una cel·la frontera, de fora del taulell, on no es poden posar fitxes. Ha estat afegida per a facilitar el control dels límits.
- **Empty**, si es tracta d'una cel·la buida.
- **Black**, si es tracta d'una cel·la ocupada per una fitxa negra.
- **White**, si es tracta d'una cel·la ocupada per una fitxa blanca.
- **Valid**, si es tracta d'una cel·la buida en la qual el jugador actual pot fer un moviment.

Per a tenir una ponderació de les puntuacions de cada posició del taulell també hem utilitzat una estructura de dades senzilla, *int*[][] *boardPoints*, un array d'enters de tamany (*height*+2)*(*width*+2) que comptarà amb un valor per cada cel·la en funció de la importància que té aconseguir-la. Aquest és omplert a la funció *fillPoints*(). Els valors possibles[1] són els següents:

- **-1**, si es tracta d'una cel·la frontera, de fora del taulell, on no es poden posar fitxes. Ha estat afegida per a facilitar el control dels límits i per a ser coherent amb l'estructura de taulell, fet que facilita la programació i comprensió del codi.
- **1**, si es tracta d'una cel·la del centre del taulell.
- **10**, si es tracta d'una cel·la del quadrat extern del taulell, exceptuant cel·les cantonada i les contigües a aquesta.
- **-10**, si es tracta d'una cel·la contigua a una cel·la cantonada. Aquest valor es deu a que posar una fitxa en aquesta cel·la pot donar l'opció al rival a ocupar una cantonada, fet que la màquina hauria d'evitar.
- **1000**, si es tracta d'una cel·la cantonada, de vital importància ja que, un cop ocupades, són inamovibles per l'oponent. Aquest valor provoca que sempre que la màquina pugui col·locar-hi una fitxa, ho faci.

[1] Valors extrets de https://www.instructables.com/Othello-Artificial-Intelligence/

-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	1000	-10	10	10	10	10	-10	1000	-1
-1	-10	-10	1	1	1	1	-10	-10	-1
-1	10	1	1	1	1	1	1	10	-1
-1	10	1	1	1	1	1	1	10	-1
-1	10	1	1	1	1	1	1	10	-1
-1	10	1	1	1	1	1	1	10	-1
-1	-10	-10	1	1	1	1	-10	-10	-1
-1	1000	-10	10	10	10	10	-10	1000	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Figura 1. Taula de ponderacions per casella d'un taulell 8x8

1.2 Algorismes

Els problemes de major complexitat que presenta Othello són el control de les normes del joc i la interpretació d'aquestes per a fer el millor moviment seguint una lògica determinada. A continuació s'explica quins mètodes hem utilitzat per a resoldre'ls.

1.2.1 Legalitat de moviments

El control de les normes del joc està implementat a la classe Board. L'única funció no auxiliar és *checkPosition* que, donades una posició i un torn, determina si és legal que el jugador hi faci un moviment i cap a quina direcció capturaria fitxes. Per a fer-ho segueix els següents passos:

1. Inicialitza un vector de posicions legals que indiquen en quina direcció creix la cerca de captura de fitxes, tindrà per tant 8 posicions.
2. Inicialitza un vector que conté les normes del joc i recorrent-lo junt amb el vector de direccions ens indicarà si en aquest taulell cal comprovar la direcció consultada.
3. Per a cadascuna de les direccions que calgui comprovar, mira si es legal capturar fitxes en aquella direcció, és a dir, aplica les normes del joc i, si ho és, afegeix la direcció que estava comprovant al vector de posicions que ens passen com a paràmetre.
4. Retorna cert si la posició pot capturar fitxes en qualsevol de les direccions.

Altres funcions que faciliten la interacció d'altres classes amb Board són:

- **getLegals**: Retorna una llista dels moviments legals per a un jugador.
    1. Recorre el taulell. Per a cada casella:

       1.1. Comprova si és legal fer-hi un moviment. Si ho és, afegeix la posició de la casella al conjunt de moviments legals.

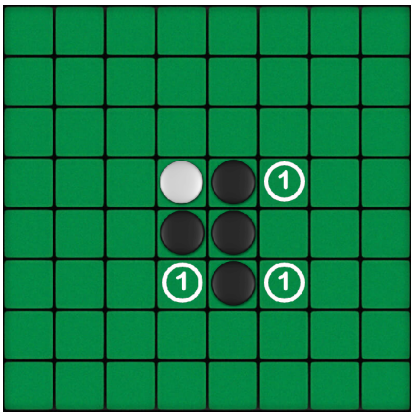
- **makeMoveTurn**: Fa un moviment si aquest és legal.
    1. Comprova que el moviment desitjat es vol fer dins dels límits del taulell.
    2. Mitjançant la funció privada *updateBoard*, comprova que la posició sigui un moviment vàlid i, si ho és, captura les fitxes corresponents al moviment, actualitzant el taulell.
    3. Retorna cert si el moviment és vàlid i, per tant, s'ha fet el moviment i actualitzat el taulell.

1.2.2 Algorismes de presa de decisions

El procés de decisió al torn de la màquina està implementat a la classe Algorithm. Totes les funcions no auxiliars reben un taulell en un estat concret i el torn del jugador actual, i actualitzen el taulell segons la lògica de l'algorisme seleccionat. Els possibles algorismes són els següents:

- **Random**: Fa un moviment aleatori del conjunt de moviments legals.
    1. Obté un vector de posicions legals. Si no hi ha moviments possibles, retorna.
    2. Calcula un enter aleatori *rand* entre 0 i el tamany del vector.
    3. Fa el moviment a la cel·la que es troba en la posició *rand* del vector.

    Exemple:



    Donada aquesta situació, Random triarà qualsevol de les tres posicions aleatòriament ja que no aplica cap tipus de lògica.

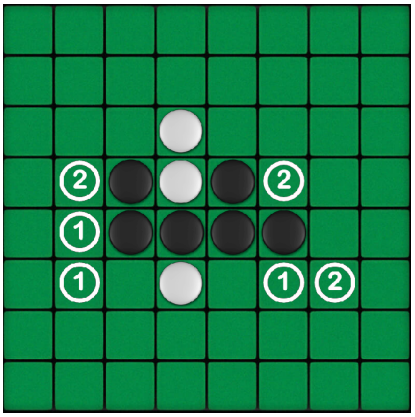
- **Greedy**: Fa el moviment que més augmentarà la puntuació del jugador. En cas d'haver més d'un que obtindria el mateix increment, fa el que correspon a la fila menor i, en cas de tenir la mateixa fila, el de columna menor.

    1. Obté un vector de posicions legals. Si no hi ha moviments possibles, retorna.
    2. Recorre el vector. Per a cada possible moviment:

        2.1. Crea un taulell temporal i prova el moviment.

        2.2. Calcula la puntuació del jugador en aquest escenari. Si aquesta és superior a les calculades anteriorment, es guarda el moviment i actualitza la màxima puntuació.
    3. Fa el moviment guardat.

    Exemple:



    Donada aquesta situació, Greedy triarà la posició de fila 4 columna 2, ja que així obtindrà el màxim increment en la puntuació possible. Tot i aconseguir el mateix increment, no triaria les caselles 4-6 ni 6-7 ja que aquestes tenen un valor de fila i columna major.

- **Minimax**: Fa un moviment buscant minimitzar les pèrdues en el pitjor cas, és a dir, tria, d'entre tots els possibles escenaris en els *n* torns següents (profunditat de la cerca), quin serà el millor moviment que pot fer suposant que l'oponent fa el pitjor moviment per a ell.

    1. Obté un vector de posicions legals. Si no hi ha moviments possibles, retorna.
    2. Recorre el vector per a comprovar si hi ha una cantonada disponible.[2]
    3. Si no ha trobat cap cantonada, recorre el vector. Per a cada possible moviment:

        3.1. Crea un taulell temporal i prova el moviment.

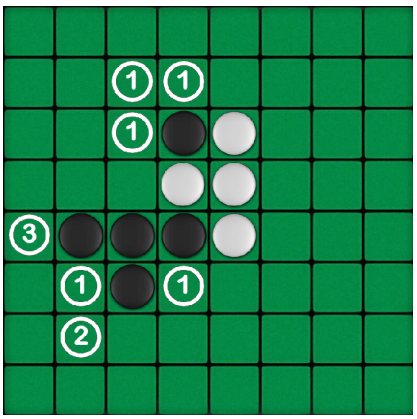
[2] Aquest pas ha estat afegit com a optimització per a evitar fer el costós càlcul recursiu en el cas de poder fer un moviment a una cantonada. En el cas de minimax sense ponderacions, també millora la decisió ja que, sense la comprovació, les cantonades no tindrien prioritat.

3.2. Partint d'aquest taulell, genera un arbre de profunditat *depth* de possibles escenaris, provant tots els moviments possibles en cada cas.

3.3. Comprova cada escenari (cada branca) i tria quin aconsegueix que la diferència entre la puntuació del jugador i la de l'oponent (suposant que aquest últim fa sempre el pitjor moviment per al jugador) sigui major.

4. Fa el moviment que ha obtingut el millor escenari final en *depth* torns.

Exemple:



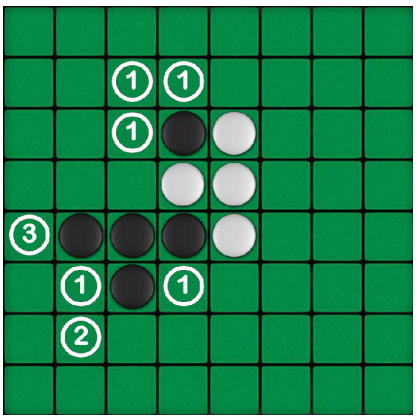
Donada aquesta situació, Minimax obtindrà els següents valors heurístics:

	1	2	3	4	5	6	7	8
1								
2			4	2				
3			4	○	○			
4				○	○			
5	4	○	○	○	○			
6		2	○	0				
7		2						
8								

Per tant triarà la casella 2-3, ja que compta amb el valor més alt i, en cas d'empat, escull com en el cas de Greedy, la que correspon a la fila meno i, en cas de tenir la mateixa fila, el de columna menor.

- **Weighted Minimax**: Fa un moviment buscant minimitzar les pèrdues en el pitjor cas, també tenint en compte ponderacions per cada casella. Els passos que segueix són els mateixos que Minimax amb la diferència de que, per a seleccionar el millor escenari, enlloc de sumar 1 a la puntuació per cada casella d'un color, suma el valor de la casella a *boardPoints*. Això resulta en què la màquina té més preferència per triar algunes cel·les que d'altres i per evitar que l'oponent es faci amb elles.

  Exemple:



Donada aquesta situació, Weighted Minimax obtindrà els següents valors heurístics:

	1	2	3	4	5	6	7	8
1								
2			2	2				
3			2	○	○			
4				○	○			
5	13	○	○	○	○			
6		2	○	0				
7		-2						
8								

A diferència de Minimax normal, donada aquesta situació, Weighted Minimax triarà la casella 5-1, ja que la seva decisió es veurà influenciada pel valor de 10 que té a la matriu de ponderacions. Tot i escollir la casella de major puntuació en aquest cas, en altres ocasions pot no triar-la perquè ha trobat algun altre escenari que minimitzarà encara més les pèrdues, com per exemple un cas en el que, suposant sempre que l'oponent fa el pitjor per a ell, el jugador aconsegueix una cantonada.

1.2.3 Gestió de games i rankings

La funcionalitat principal de la classe ranking és proporcionar una llista ordenada de jugadors segons diferents paràmetres. Hem utilitzat la forma més simple que coneixíem que ens ho permet fer, una llista de *players*: *ArrayList<Player> ranking*. Permet ordenar el ranking segons identificador, nombre de victòries, nombre de derrotes i nombre d'empats.

El ranking es construeix a partir del registre de partides de la classe GameLog, representat per una llista de *games*: *ArrayList<Game> log*. Es recorre comprovant qui ha jugat cada partida del registre i quin ha sigut el resultat. En funció d'aquest, es modifiquen els atributs dels jugadors fins a tenir una estructura coherent amb el *log*.

1.3 Cost dels algorismes

Donades *h*, *w* i *d*, sent altura, amplada i diagonal del taulell, les complexitats dels algorismes són les següents:

- **checkPosition**: Donat que recorre les direccions horitzontal, vertical i diagonal respecte a una posició del taulell:

  $$checkPosition \in O(h+w+2d)$$

  En el cas més comú tindrem un taulell quadrat on, per tant, les dimensions valen el mateix:

  $$checkPosition \in O(4n)$$

- **getLegals**: Donat que recorre el taulell i crida *checkPosition* a cada casella, el seu cost ve definit pel producte del cost de *checkPosition* i les dimensions del taulell:

  $$getLegals \in O(h*w*(h+w+d))$$

  En taulells quadrats:

  $$getLegals \in O(3n^3)$$

- **makeMoveTurn**: Donat que comprova la legalitat del moviment i actualitza les caselles de l'oponent capturades, aquest últim un valor que sempre serà molt menor a les dimensions del taulell:

  $$makeMoveTurn \in O(h+w+2d)$$

  En taulells quadrats:

  $$makeMoveTurn \in O(4n)$$

- **Random**: Tret de les crides a *getLegals* i *makeMoveTurn*, el cost d'aquest algorisme és constant. Per tant, el seu cost ve definit per la suma del cost d'aquestes:

$$randAlgorithm \in O(h*w*(h+w+d) + (h+w+2d))$$

En taulells quadrats:

$$randAlgorithm \in O(3n^3+4n)$$

$$randAlgorithm \in O(3n^3)$$

- **Greedy**: Tret de les crides a *getLegals*, *clone* (el seu cost és recórrer el taulell: $clone \in O(h*w)$) i *makeMoveTurn*, el cost d'aquest algorisme és constant. Per tant, el seu cost ve definit per la suma del cost d'aquestes:

$$greedyAlgorithm \in O(h*w*(h+w+d) + (h*w) + (h+w+2d))$$

En taulells quadrats:

$$greedyAlgorithm \in O(3n^3+n^2+4n)$$

$$greedyAlgorithm \in O(3n^3)$$

- **Minimax**: Aquest cas és més interessant i complex. Consta de dues parts diferenciades, la funció externa, que recorre els moviments legals i escull el millor, i la interna, *minimaxValue*. *minimaxValue* crea recursivament un arbre d'escenaris d'una profunditat determinada per a cada moviment legal. Per defecte, *depth* val 3. Primer cal saber el nombre de casos recursius *ncases* dins de *minimaxValue*, que serà igual al nombre de vèrtexs de l'arbre. Donat *m*, sent el nombre de moviments legals que pot fer un jugador en un torn:

$$ncases = \sum_{i=0}^{depth} m^i$$

Per a cada cas recursiu, tenim una crida a *getLegals* i una altra a *makeMoveTurn*. Un cop hem arribat a una fulla de l'arbre, calculem l'heurística, que recorre el taulell per a saber la puntuació. Aquest càlcul és lineal amb el tamany del taulell ($score \in O(h*w)$). Això porta el cost de *minimaxValue* a:

$$minimaxValue \in O(ncases*(h*w*(h+w+d)+(h+d+w)) + m^{depth}*h*w)$$

En taulells quadrats:

$$minimaxValue \in O(ncases(3n^3+3n) + m^{depth}n^2)$$

En quant a *minimaxAlgorithm* en general, aquest crida *getLegals* i per a cada posició legal fa *makeMoveTurn* i *minimaxValue*. El seu cost és la suma del cost de *getLegals* amb el producte de *m* (moviments legals) i el cost de *makeMoveTurn* i *minimaxValue*:

$$minimaxAlgorithn \in O(2*h*w*(h+w+d)) + m*((h+w+2d) +$$

$$(ncases*(h*w*(h+w+d)+(h+d+w))+m^{depth}*h*w)$$

En taulells quadrats:

$$minimaxAlgorithn \in O(6n^3 + m*4n + ncases(3n^3+3n) + m^{depth}n^2)$$

Per últim, esmentar que en el cas de trobar una cantonada a la primera cerca, s'estalvia tot el cost relatiu a *minimaxValue*, reduint el cost al de *getLegals*:

$$minimaxAlgorithn \in O(3n^3)$$

- **Weighted Minimax**: Com que l'única diferència es troba en els valors que sumem a l'hora de calcular puntuacions, la seva complexitat és exactament igual al del minimax sense pesos.

- **generateRanking**: Donat *n* elements del registre de partides, com que el recorrem comprovant la informació de cada *game*, el cost és lineal:

$$generateRanking \in O(n)$$

- **sortRanking**: Donat *n* elements de la llista, ens trobem amb el cost habitual d'ordenar utilitzant *quicksort*:

$$sortRanking \in O(n*log(n))$$