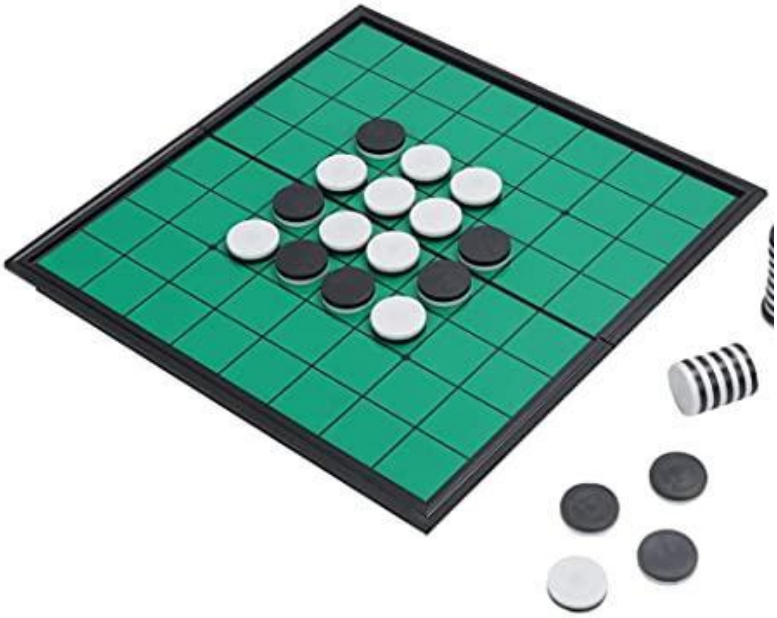
Paral·lelització de IA (Othello)

Curso 2022-2023



Assignatura: PACO

Marc Pérez Guerrero - 48076178Z
Adrian Garcia Campillo - 47114120T

Index

Explicació Heurística

Introducció

La heurística està formada per un seguit de funcions que calculen la puntuació d'un moment determinat del tauler de joc, traient la diferencia de puntuació entre el rival i la nostre, obtenint com a resultat un valor més gran o més petit segons el que ens interessi arribar a aquella situació de tauler.

Explicació

L'heurística en aquest cas té en compte els següents aspectes:

- Nombre de fitxes de cada jugador
- Valor d'una fitxa a una determinada casella (per exemple les cantonades són rellevants)
- Quantitat de moviments possibles de cada jugador (com més moviments més probable tenir un de molt bo)

Com es calcula

Funció Heurística “*heuristica(GameStatus estat)***”**

Aquesta funció s'utilitza per a calcular la puntuació heurística d'un tauler de *Othello en particular. La puntuació heurística és un valor que s'utilitza per a avaluar l'estat actual del joc i determinar quina és la millor opció a prendre en aquest moment.

La funció comença comprovant si el joc ha acabat o no. Si ha acabat, retorna un valor molt alt o molt baix depenent de si el jugador actual ha guanyat o perdut. Si el joc no ha acabat, la funció continua avaluant el tauler.

Per a avaluar el tauler, la funció compta el nombre de fitxes del jugador i de l'oponent. Aquests valors s'emmagatzemen en les variables "*puntuaciónJugador" i "*puntuaciónOponente". Finalment, la funció retorna la diferència entre aquests valors, la qual cosa ens dona una idea de com està avançant el joc. Si el jugador té més fitxes, significa que està guanyant. D'altra banda, si l'oponent té més fitxes, significa que està guanyant.

Funció Contar Fitxes “*contarFichas(GameStatus estat, int fil)***”**

Aquesta funció s'utilitza per a comptar el nombre de fitxes que té cada jugador en un tauler de *Othello. La funció utilitza una estructura de dades anomenada "*Move" per a emmagatzemar el nombre de fitxes del jugador i de l'oponent.

La funció comença recorrent cada fila del tauler i, per a cada fila, recorre cada columna del tauler. Si troba una fitxa del jugador, augmenta la puntuació del jugador en una quantitat determinada. Si troba una fitxa de l'oponent, augmenta la puntuació de l'oponent en una quantitat determinada. Aquestes quantitats es determinen a partir d'una matriu de valors anomenada "valors".

Quan la funció ha acabat de recórrer totes les files i columnes del tauler, retorna el nombre total de fitxes de cada jugador a través de l'estructura de dades "*Move".
Tot i que l’estructura de dades "*Move" no està pensada per això, l’hem reutilitzat.

Paral·lelisme

Introducció

El paral·lelisme és la capacitat d'executar múltiples tasques o instruccions al mateix temps. Això es pot aconseguir mitjançant l'ús de múltiples CPUs o processadors, o utilitzant una sola CPU amb múltiples nuclis o fils d'execució.

El paral·lelisme es pot utilitzar per millorar el rendiment d'una aplicació dividint la feina en petites tasques que es poden executar de forma independent. Això es pot fer de forma manual, escrivint codi específicament per aprofitar el paral·lelisme, o utilitzant una biblioteca o una interfície de programació com OpenMP per automatitzar aquest procés.

En general, el paral·lelisme és una tècnica clau per millorar el rendiment de les aplicacions i fer-les més eficients en entorns de computació amb múltiples CPUs o nuclis.

Instruccions OpenMP

1. **#pragma omp parallel:** Aquesta directiva indica a OpenMP que ha de crear un grup de fils i executar la secció de codi de forma paral·lela.

2. **#pragma omp for:** Aquesta directiva indica a OpenMP que ha de paral·lelitzar un bucle for.

3. **#pragma omp sections:** Aquesta directiva permet dividir una aplicació en diferents seccions que es poden executar de forma paral·lela.

4. **#pragma omp single:** Aquesta directiva indica a OpenMP que ha de fer que un sol fil execute una secció de codi.

5. **#pragma omp task**: Aquesta directiva permet crear tasques que es poden executar de forma paral·lela.

6. **#pragma omp critical:** Aquesta directiva indica a OpenMP que ha de protegir una secció de codi perquè només es pugui accedir a ella per un sol fil a la vegada.

7. **#pragma omp barrier:** Aquesta directiva indica a OpenMP que ha de fer que tots els fils esperin fins que tots hagin arribat a aquest punt abans de continuar.

8. **#pragma omp flush:** Aquesta directiva indica a OpenMP que ha de sincronitzar la memòria compartida entre els fils.

Explicació estratègies

Hi ha diferents estratègies per paral·lelitzar una aplicació:

9. Paral·lelisme d'instrucció: Aquesta estratègia consisteix a dividir les instruccions d'una aplicació en petites tasques que es poden executar de forma independent. Això es pot fer amb l'ús de múltiples CPUs o nuclis.

10. Paral·lelisme de dades: Aquesta estratègia consisteix a dividir les dades d'una aplicació en petits conjunts que es poden processar de forma independent. Això es pot fer amb l'ús de múltiples CPUs o nuclis.

11. Paral·lelisme d'ordinador: Aquesta estratègia consisteix a dividir un problema gran en petits problemes que es poden resoldre de forma independent i després combinar els resultats. Això es pot fer amb l'ús de múltiples CPUs o nuclis.

12. Paral·lelisme d'entrada/sortida: Aquesta estratègia consisteix a paral·lelitzar les operacions d'entrada/sortida d'una aplicació per millorar el rendiment. Això es pot fer amb l'ús de múltiples CPUs o nuclis.

13. Paral·lelisme de tasques: Aquesta estratègia consisteix a dividir una aplicació en diferents tasques que es poden executar de forma independent. Això es pot fer amb l'ús de múltiples CPUs o nuclis.

14. Paral·lelisme de pipeline: Aquesta estratègia consisteix a dividir una aplicació en diferents etapes o "estats" que es poden executar de forma paral·lela. Això es pot fer amb l'ús de múltiples CPUs o nuclis.

15. Paral·lelisme de dominació: Aquesta estratègia consisteix a paral·lelitzar les instruccions d'una aplicació segons les dependències de dades entre elles. Això es pot fer amb l'ús de múltiples CPUs o nuclis.

16. Paral·lelisme de xarxa: Aquesta estratègia consisteix a paral·lelitzar una aplicació utilitzant diferents nodes d'un sistema de xarxa. Això es pot fer amb l'ús de múltiples CPUs o nuclis.

17. Paral·lelisme d'acceleració: Aquesta estratègia consisteix a utilitzar l'acceleració per hardware, com ara targetes gràfiques o acceleradors de càlcul, per millorar el rendiment d'una aplicació.

18. Paral·lelisme distribuït: Aquesta estratègia consisteix a paral·lelitzar una aplicació utilitzant múltiples màquines o nodes d'un sistema distribuït. Això es pot fer amb l'ús de múltiples CPUs o nuclis.

19. Paral·lelisme divisió i conqueri: Aquesta tècnica s'utilitza per paral·lelitzar algorismes que es basen en la resolució de problemes recursius. Consisteix a dividir un problema gran en subproblemes més petits que es poden resoldre de forma independent i després combinar els resultats per obtenir la solució final.

20. Paral·lelisme de memòria: Aquesta tècnica s'utilitza per millorar el rendiment dels sistemes de memòria d'un ordinador. Consisteix a utilitzar múltiples canals de memòria o bancs de memòria perquè els diferents processadors o fils de l'ordinador puguin accedir a la memòria de forma paral·lela.

Estratègies Utilitzades

Hem creat una intel·ligència artificial capaç de jugar al joc d'estratègia *Othello, des de 0. Per a millorar la seva eficiència, hem implementat dues tècniques de paral·lelització: paral·lelització de memòria i paral·lelisme de divisió i conquesta.

Mitjançant la comanda ***omp_set_num_threads(4)*** hem fixat els threads a 4, ja que després de fer proves hem arribat a la conclusió que era el que donava resultats òptims, tot i que no utilitza tots els threads.

Funció *"JugadorIA::Move JugadorIA::move(Tablero t)"*

La funció "move" és utilitzada per la IA per decidir quin moviment fer en el joc. La funció rep un objecte "Tablero" com a argument i retorna un objecte Move. L'objecte "Tablero" conté informació sobre l'estat actual del joc de tauler i l'objecte "Move" descriu el moviment que es pot fer en aquest joc.

La funció utilitza una llista de moviments possibles, obtinguda amb la funció "getMoves", i itera sobre ells per trobar el millor moviment a fer.
Utilitza la directiva **#pragma omp parallel for** per indicar que el bucle "for" s'ha de paral·lelitzar amb OpenMP. Això vol dir que OpenMP crearà múltiples fils d'execució i assignarà cada iteració del bucle a un fil diferent. D'aquesta manera, es pot processar cada moviment de forma paral·lela, reduint el temps total de càlcul.

Per evitar errors i canvis abans de procesar l'informació de retorn, els moviments es guarden juntament amb la seva puntuació a un vector, aixo permet que conforme van acabant els diferents *threads* es vagi guardant el resultat. Posteriorment agafarem el de major puntuació.

Funció *"int JugadorIA::ValorRetorn(Tablero t)"*

La funció "ValorRetorn" és una funció que utilitza una taula hash per emmagatzemar informació sobre els estats del joc de tauler i evitar calcular de nou el valor d'estats que ja s'han visitat. Aquesta tècnica es coneix com a "memorització" i és útil per accelerar l'execució de l'algorisme.

La funció rep un objecte "Tablero" com a argument i retorna un valor enter. La funció comença calculant un índex per la taula hash amb la funció "calcular_hash" i l'emmagatzema a la variable index. A continuació, comprova si ja hi ha un valor

emmagatzemat a la taula per aquest índex amb la variable “AlmacenHash”. Si no hi ha cap valor, crida la funció “heuristica” per calcular el valor de l'estat actual i ho emmagatzema amb l'índex corresponent.

Si ja hi ha un valor emmagatzemat a la taula hash, la funció el retorna.

La funció utilitza la directiva **#pragma omp critical** per protegir l'accés a la taula hash. Això vol dir que només un fil a la vegada pot accedir a la taula hash i modificar-la. Això és important perquè la taula hash és una estructura de dades compartida entre els diferents fils i es pot produir un conflicte si diferents fils intenten modificar-la alhora.

Finalment, la funció retorna el valor emmagatzemat a la taula hash per l'índex corresponent.

Funció *“calcular_hash(Tablero t, int i, int j)”*

La funció calcular_hash té com a objectiu calcular un hash per a un objecte Tablero donat. Aquest hash és un número sencer de 64 bits que es pot utilitzar per identificar de forma única una configuració de tauler.

La funció recorre cada posició del tauler i utilitza una matriu de hash per calcular el valor hash per a cada posició. Després, fa una operació XOR amb el valor hash actual i el valor hash de la posició actual per calcular el valor hash final.

La matriu de hash s'utilitza per associar cada valor de tauler amb un valor hash únic. Això és útil per poder comparar configuracions de tauler de forma ràpida i senzilla, ja que els valors hash són molt més petits que els valors de tauler complets.

L'ús de la funció XOR en el càlcul del hash és una tècnica comuna per combinar diferents valors en un únic valor hash. Això és útil perquè fa que el valor hash depengui de tots els valors de tauler, de manera que qualsevol canvi en una posició del tauler tindrà un impacte en el valor hash final.

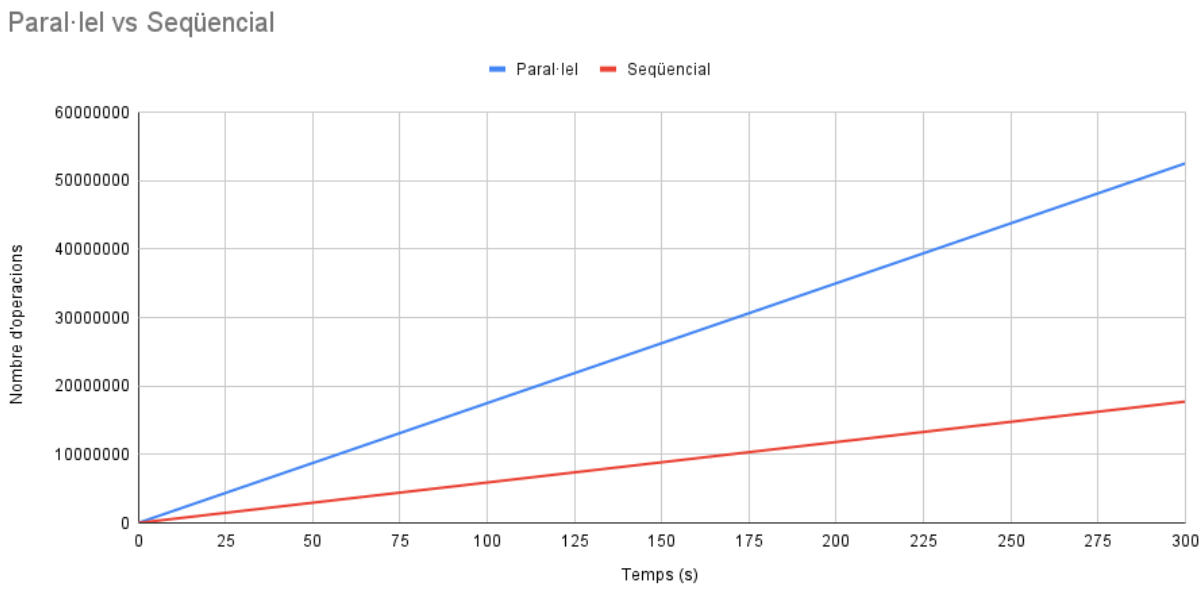
Aquesta funció utilitza el paral·lelisme de memòria. D’aquesta manera millorem el rendiment del programa, fent que diferents processadors no tinguin que calcular l’heurística repetidament. Així aconseguim que el que calculi un procesador es comparteixi.

Mostres/ Exemples

Dades

	Nombre d'operacions	
Temps (s)	Paral·lel (4 processadors)	Seqüencial
0	0	0
1	175156,4	59112,06436
25	4378909,999	1477801,609
50	8757819,999	2955603,218
75	13136730	4433404,827
100	17515640	5911206,436
125	21894550	7389008,045
150	26273460	8866809,653
175	30652370	10344611,26
200	35031280	11822412,87
225	39410189,99	13300214,48
250	43789099,99	14778016,09
275	48168009,99	16255817,7
300	52546919,99	17733619,31

Gràfic



Comentari de les dades

El gràfic mostra el temps que es triga a realitzar un cert nombre d'operacions tant en un procés paral·lel com en un seqüencial. En l'eix horitzontal (X) es troba el nombre d'operacions, mentre que en l'eix vertical (Y) es troba el temps necessari per a completar aquestes operacions.

En el cas del procés paral·lel, podem veure que a mesura que augmenta el nombre d'operacions, també augmenta el temps necessari per a completar-les, però en general el temps és menor que en el cas del procés seqüencial. Això es deu al fet que en un procés seqüencial, cada operació es du a terme una després de l'altra, sense poder aprofitar la potència de processament de múltiples nuclis o processadors.

Per exemple podem veure que en 1 segons el codi paral·lelitzat és capaç de fer un 296% més d'operacions que el seqüencial, i això es repeteix al llarg de les altres dades. Podem afegir que a causa de dependències el codi paral·lel ha d'esperar a alguns processos com per exemple escriure/ consultar la taula de hash o bé esperar a tenir tots els resultats dels moviments inicials possibles, per poder escollir el millor sense superposar cap. Això desencadena una inevitable pèrdua de rendiment que cal assumir, tot i això, el codi és molt més ràpid.

Per tant, quan es necessiten realitzar un gran nombre d'operacions, el temps total necessari pot ser significativament major en un procés seqüencial que en un paral·lel.