PROGETTO ASD2021

1) INTESTAZIONE

-Nome progetto: ComputerPlayer.
-Nome giocatore: Cosimo++

Progetto realizzato singolarmente: Cosimo Coppolaro 999368

2) PROBLEMA

A-Individuare vantaggi/svantaggi di Minimax e AlphaBeta Pruning per scegliere quale adottare nel progetto. Capire inoltre come applicarli al problema in questione.

B-Capire se il Game Tree è necessario generarlo “fisicamente” tramite apposita funzione e successivamente visitarlo con gli algoritmi citati oppure già l’algoritmo in se è in grado di visitare “l’albero” senza bisogno di crearlo.

Questo dubbio era però dovuto ad una mancata (totale) comprensione di tali algoritmi. E’ stata necessaria una iniziale implementazione di Minimax che visita un albero precedentemente creato tramite apposita funzione per capire che quella non era la strada più efficiente in termini di costo computazionale per risolvere il problema.
Inoltre il primo giocatore creato era inefficiente.
A quel punto ho capito che era sufficiente l’algoritmo AlphaBeta Pruning adeguatamente applicato (in aggiunta ad una buona euristica per i nodi non foglia) per ottenere una soluzione accettabile/ottima.

C-Capire come implementare la funzione euristica per analizzare i nodi non-foglia.

L’idea è sempre stata quella di favorire configurazioni che mi portassero alla creazione (per il mio giocatore) di “forchette” all’interno della griglia di gioco. Ed evitassero quelle dell’avversario.
Dopo varie valutazioni e prove capii che le “forchette” si formano automaticamente, basta una corretta implementazione dell’algoritmo minimax/alphabeta pruning.
Si ripresentava quindi il problema di creare una buona euristica per velocizzare tale processo.
Inizialmente l’idea era quella di valutare ogni configurazione riga per riga. Dopo poche prove ho però capito che tale soluzione era troppo dispendiosa a livello di tempo e soprattutto inutile.
La soluzione adottata è stata quindi la seguente: valutare le quattro righe (verticale, orizzontale, diagonale ed antidiagonale) che passano per l'ultima cella marcata. Sommare poi le 4 valutazioni.
La valutazione di ogni riga avviene tramite una funzione, “evaluateLine()”, che restituisce uno score positivo o negativo a seconda se la configurazione analizzata è favorevole a noi o all’avversario.

D-Valutazione dei nodi foglia vittoria/sconfitta influenzata dalla profondità della mossa.

Per le configurazioni di gioco finali, vittoria e sconfitta, non ci si limita ad assegnare valori fissi (rispettivamente 45 e -45), ma si fa variare lo score in base alla profondità della mossa vincente/perdente.
Vittoria: lo score diminuisce con l'aumentare dei livelli. Si favoriscono le vittorie più vicine.
Sconfitta: lo score diminuisce in valore assoluto con l'aumentare dei livelli, si assegna cioè peso maggiore alle configurazioni più lontane in modo da ritardare la sconfitta e sperare nell’errore avversario in caso di configurazioni esclusivamente perdenti.

E-Capire come valutare una riga.

La valutazione viene effettuata chiamando la funzione “maxSubLineOccupiedCells()” che calcola il sottovettore (della riga in input) di lunghezza massima contenente indistintamente celle marcate da player (parametro in input) e celle libere. Restituisce 2 parametri: il numero di elementi del sottovettore citato e il numero di celle libere in esso contenute. La funzione precedente viene chiamata 2 volte per linea, cioè una per giocatore.
Si valuta se la sottoriga presenta numero di celle >= al numero di celle necessarie da marcare consecutivamente per vincere (K). In caso affermativo si somma tale score, altrimenti passiamo alla valutazione della (eventuale) sottoriga dell'avversario, ed in tal caso lo score viene sottratto.
Lo score è dato dal numero di elementi della sottoriga meno quello di celle libere in esso contenute.

F-Costatazione del fatto che non è possibile generare l’intero GameTree ma bisogna limitarci a ricerche mirate oppure limitare il numero di livelli di profondità dell’albero.

Nella mia implementazione ho optato per la ricerca limitata su un numero variabile (ma prefissato) di livelli di profondità. Il numero di tali livelli su cui si applica l’algoritmo dipende dalla grandezza della griglia, e più precisamente dal numero di celle (ancora) vuote presenti ad ogni turno.
Il numero di livelli varia da 1 per n° celle vuote > 85, a 6 per n° celle vuote >= 0 e <= 16.
Tali parametri sono stati decisi attraverso test pratici.

3) SCELTE PROGETTUALI

A_Implementazione metodo selectCell()

MNKCell selectCell(FC, MC)
{
-si inizializza il timer che conta i millisecondi a partire da quello corrente
-si recupera l’ultima mossa effettuata per aggiornare la board locale
-se c’è un unica mossa possibile si ritorna immediatamente
-se la prima mossa spetta al mio giocatore può essere effettuata randomicamente
-ciclo FOR su tutte le celle libere della board locale
{
-se il tempo sta per superare 10 sec si ritorna una mossa random
-si marca una cella
-IF-ELSE a cascata per chiamare la funzione
ALPHABETA(board, true, n°livelli, -INF, +INF) su numero di livelli
inversamente proporzionale al numero di celle libere sulla board locale
-si valuta lo score dell’ultima mossa con quello della migliore precedente e se è > si
seleziona quella corrente come mossa migliore (massimizzazione)
-si deselezione l’ultima mossa effettuata
}
-si marca la cella risultante come migliore sulla board locale e si ritorna come parametro
}

```
double alphabeta(board, myNode, depth, alpha, beta)
{
        -Se si è raggiunta profondità 0 o un nodo foglia o il tempo sta per superare 10 sec allora si
         chiama la funzione EVALUATE(board, depth)

        -Altrimenti Se (myNode) allora{
                        eval = +INF
                        FOR su tutte le celle libere
                                -si marca la cella libera
                                eval2 = alphabeta(board, myNode, depth-1, alpha, beta)
                                eval = Min(eval, eval2)
                                beta = Min(eval, beta)
                                -si deseleziona l'ultima mossa fatta
                                -se beta <= alpha allora esci dal ciclo

        }altrimenti "!myNode"
        {
                …
        }
        -ritorna eval
}

double evaluate(board, depth)
{
        -Se myWin allora
                eval = 45
                eval -= livelli di profondità
        -Se yourWin allora
                eval = -45
                eval += livelli di profondità
        -Se DRAW allora eval = 0
        -Altrimenti eval = evalOpenConfig(board)

        -return eval
}

double evalOpenConfig(board)
{
        double score
        …
        -Se cols >= k allora  score += EVALUATELINE(rowLine)
        …
        -Se rows >= k allora  score += EVALUATELINE(colLine)
        …
        -Se diagLine.size() >= k allora  score += EVALUATELINE(diagLine)
        …
        -Se antiDiagLine.size() >= k allora  score += EVALUATELINE(antiDiaglLine)

        return score
}
```

```
double evaluateLine(vector MNKCellState)
{
        double score

        {n° subline's cells, n° celle libere} myLine[2] = maxSubLineOccupiedCells(vector, myCell)
        -Se myLine[0] >= K allora   score += myLine[0] – myLine[1]

        {n° cells, n° celle libere} yourLine[2] = maxSubLineOccupiedCells(vector, yourCell)
        -Se yourLine[0] >= K allora   score -= myLine[0] – myLine[1]

        return score
}

int[] maxSubLineOccupiedCells(array MNKCellState, player)
{
        Funzione che adotta la programmazione dinamica per individuare il sottovettore dell'array
        in input di lunghezza massima contenente celle marcate da player e celle free.
        Ritorna il numero di elementi di tale sottovettore e quello degli elementi liberi.
}
```

B_Strutture dati e algoritmi noti utilizzati

Algoritmi:
-alphabeta pruning
-algoritmo basato sulla PROGRAMMAZIONE DINAMICA per calcolare il sottovettore di lunghezza massima nel caso della funzione "maxSubLineOccupiedCells()"

Strutture Dati:
-ArrayList class di Java

Strategie originali:
-funzione euristica che valuta una configurazione valutando le 4 linee che passano per l'ultima cella marcata
-valutazione di una riga basata sulla funzione che calcola il sottovettore di lunghezza massima
-chiamata alla funzione alphabeta() su numero variabile di livelli a seconda del numero di celle libere presenti sulla board. Con il diminuire delle celle libere alphabeta() può agire su profondità via via sempre maggiori (fino ad arrivare a 6). Ciò garantisce un buon rapporto efficienza computazionale/ottimalità scelta della mossa.

C_Analisi complessità computazionale

-"maxSubLineOccupiedCells(MNKCellState[] vec, MNKCellState player)" ha costo lineare O(t) con 't' numero di elementi di vec. L'algoritmo è basato su programmazione dinamica e ogni elemento dell'array è visitato una sola volta attraverso un ciclo for.

-"evaluateLine(MNKCellState[] l)" esegue due chiamate alla funzione maxSubLineOccupiedCells() + operazioni di costo costante. Sia 'h' il n° di elementi dell'array in input, allora il costo della funzione è $T(h) = 2*O(h) + O(1) = O(h)$.

-”evalOpenConfig(MNKBoard b_)” all’interno del suo corpo presenta:

_2 FOR annidati, uno eseguito N volte e l’altro M volte (con NxM dimesione della board). Hanno costo O(N*M).

_1 FOR eseguito H volte con H numero di celle marcate nella board. Costo O(H).

_Presenta inoltre 4 condizioni IF con chiamate alla funzione ‘evaluateLine()’. Le 4 chiamate a funzione hanno i seguenti costi: O(N), O(M), 2*O(sqrt(N^2 * M^2)).

_Abbiamo altri cicli while che hanno costi irrisori rispetto alle altre operazioni presenti nella funzione, cioè molto minori di O(sqrt(N^2 * M^2)). Mi riferisco ad operazioni eseguite sulle diagonali della board.

Il costo totale della funzione è quindi O(N*M), con NxM dimensione della board.

-”evaluate(MNKBoard b, int d)” all’interno del corpo presenta 4 IF condizionali.

_Se (b.gameState == myWin || b.gameState == yourWin || b.gameState == DRAW) allora abbiamo solo operazioni di costo costante, quindi T(N*M) = O(1).

_Altrimenti si ha ‘b.gameState == OPEN’ e quindi si effettua la chiamata alla funzione ‘evalOpenConfig()’. In questo caso quindi il costo è T(N*M) = O(N*M).

-”alphabeta(board, .., depth, ..)”, sia ‘m’ n° medio di mosse e ‘d’ massima profondità di ricerca.

L’algoritmo alphabeta pruning ha costo:

_caso pessimo: O(m^d) _caso ottimo: O(sqrt(m^d))

Nel nostro caso la profondità d varia da 1 a 6.

Continuiamo l’analisi del nostro algoritmo considerando solo il caso pessimo.

Bisogna aggiungere il costo della funzione ‘evaluate()’, che come detto è O(M*N).

Il costo della funzione sarà quindi:

T(M, N, d, m) = O(m^d) + O(M*N) = O(max(m^d, M*N))

con m=[1, FC.size()] e d=[1,6].

Cosimo Coppolaro 999368