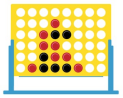
Relazione Progetto ASD2023

Alma Mater Studiorum - Università di Bologna



Erik Dervishi
numero di matricola : 0001069599
mail istituzionale : erik.dervishi@studio.unibo.it

Gian Luca Montefiori
numero di matricola : 0001070166
mail istituzionale : gian.montefiori@studio.unibo.it

Piero Mongelli
numero di matricola : 0001070630
mail istituzionale : piero.mongelli@studio.unibo.it

A.A. 2022-23

Indice

1 Introduzione

Il progetto si basa su una generalizzazione del gioco forza 4/connect 4. La differenza con il gioco originale consiste nel fatto che l'incontro si svolgerà in una matrice di gioco M×N con M righe, N colonne e X pedine consecutive (da qui Connect X) da collegare per poter vincere. Le pedine possono essere collegate tra loro in modo verticale, orizzontale e diagonale. Il turno di ogni giocatore consiste nell'inserire un gettone in una determinata colonna, con lo scopo di concatenarne piu' possibili.
L'obiettivo del progetto e' quello di creare un bot in grado di svolgere e vincere partite di ConnectX contro altri bot. Ad ogni turno, quindi, dovra' scegliere una colonna tra quelle disponibili nella quale inserire un gettone. L'unico vincolo che viene posto per quanto riguarda il turno e' il tempo di decisione, perche' la scelta deve essere fatta entro un dato TIMEOUT, oltre il quale il turno verra' considerato invalido.
Per compiere sempre la scelta migliore e vincere piu' partite possibili, si rende necessario prevedere le mosse dell'avversario e agire di conseguenza.

2 Scelte implementative

2.1 SelectColumn()

L'obiettivo di selectColumn() e' di scegliere una colonna tra quelle disponibili nella quale compiere una mossa.
Per scegliere la mossa migliore viene usata una strategia di ricerca chiamata Iterative Deepening, basata su un algoritmo di minimax con alpha/beta pruning.

2.1.1 Iterative Deepening

L'idea di base è quella di utilizzare al massimo il tempo disponibile per il turno, allo scopo di ponderare la scelta in modo che sia la migliore possibile.
Per applicare questa idea la scelta è ricaduta sull'iterative deepening. Questa tecnica esegue ripetutamente l'alpha/beta pruning, ma aumentando la depth ad ogni iterazione fino a quando non si raggiunge il numero di celle libere in quel momento (ovvero si prevede tutta la partita) o fino a quando non si raggiunge una vittoria. Ogni volta viene salvata la mossa migliore fino a quel momento, e quando il tempo sta per scadere viene fatto il return di quella trovata per ultima.
L'unico svantaggio di questa strategia è il fatto che a ogni iterazione si devono ricontrollare anche nodi gia' visti in passato.

2.1.2 Alpha/Beta Pruning

Per valutare e poter prevedere mosse che potrebbero succedere nella partita utilizziamo l'algoritmo Alpha/Beta Pruning (variante di Minimax altamente ottimizzata).
Le possibili mosse in un turno possono essere viste come nodi di un albero, dove la profondità dell'albero indica il numero del turno.
Con l'utilizzo di un minimax la quantità di calcolo sarebbe troppo onerosa, perchè dovrebbe esplorare troppi nodi per poter restituire una valutazione affidabile entro il tempo limite.
Per questo motivo si utilizza l'algoritmo di Alpha-Beta pruning, il quale riduce notevolmente il numero di nodi da visitare, ciò è possibile grazie a un meccanismo di potatura, che ignora sotto alberi che non influenzerebbero la nostra valutazione della mossa ottima.

La visita delle mosse possibili inizia dalla colonna centrale,grazie alla funzione middlesort, in modo tale da vedere prima le mosse che sono probabilisticamente le migliori, questa scelta è per avere una potatura dei nodi più ampia e velocizzare il ritrovamento della migliore posizione.
La valutazione della mossa corrente viene influenzata dalla sua profondità, più aumenta il numero di depth più la sua valutazione è bassa, così da assegnare una rilevanza maggiore a vittorie ottenute con una profondità minore oppure per ritardare il più possibile le sconfitte.

2.1.3 MiddleSort()

Questa funzione riordina un array di interi L (array che indica le diverse colonne libere dove si possono aggiungere le pedine) in base a una logica che posiziona gli elementi del centro all'inizio, seguiti dagli elementi che si allontanano progressivamente dal centro.

2.2 Euristica

L'euristica utilizzata si basa sul concetto di token. Un token e' una concatenazione di pedine che potrebbe portare a una vittoria. Per esempio, in connect 4, due pedine di fila verticali sono un token. Se pero' l'avversario dovesse posizionarci una pedina sopra, interrompendo la sequenza, quella concatenazione di token smetterebbe di avere importanza perche' verticalmente non potrebbe piu' portare a una vittoria. Il numero di token viene salvato in due array, myToken e oppToken. Gli array hanno lunghezza X-1 perche' non contano le pedine singole, ma solo i token a partire da lunghezza 2.
I token di N gettoni allineati vengono messi all'indice N-2 dell'array corrispondente al giocatore (mytoken=noi, opptoken=nemico). Gli array vengono resettati e riempiti per ogni mossa.
Ad ogni mossa viene studiata tutta la board di gioco e aggiornati i token.
La board di gioco viene esplorata grazie a 3 metodi:
verticalCount(), che conta tutti i token verticali; horizontalCount(), che conta tutti gli orizzontali; diagonalCount(), che conta i diagonali e gli antidiagonali.

Il valore delle rispettive mappe di gioco sono valutate col seguente criterio:

```
public Integer heuristic() {
       calculate_tokens();

       Integer heuristic = 0;
       for(int i = 0; i < K-1; i ++) {
           heuristic += myTokens[i]*myTokens[i]*myTokens[i] -
               oppTokens[i]*oppTokens[i]*oppTokens[i];
           myTokens[i]=0;
           oppTokens[i]=0;
       }
       return heuristic;
}
```

Qui sotto viene riportato il codice di AnitaMaxMin, per contare il numero di token verticale(esecuzioni analoghe per le altre 3 direzioni).

```
private void vertical_count(Cell) {
    int counter;
    CellState expected;
    for(int j = 0; j < N; j ++) {
        if(cell[M-1][j] == FREE) //colonna libera, non ha senso
            controllarla
            continue;
        int cell2controll = 0;
        for(int i = 0; i < M; i++) {
            if(cell[i][j] != FREE) {
                cell2controll = i;
                break;
            }
        }
        expected = cell[cell2controll][j];
        counter = 0;
        while(cell2controll <= M-1 && expected ==
            cell[cell2controll][j] && counter < K) {
            counter++;
            cell2controll++;
        }
        if(counter > 1){
          if(expected == our){
              myTokens[counter-2] ++;
          }
          else{
              oppTokens[counter-2] ++;
          }
        }
    }
}
```

3 Costi computazionali

3.1 Costo dell'euristica

Il valore dell'euristica e' calcolato dalla funzione Heuristic, che chiama a sua volta i vari metodi di count:

- **Count Vertical**: **O**(**MN**)
- **Count Horizontal**:

  $O((M-X)(N-X)X) = O((MN-MX-NX+K^2)X) = O(MNX-MX^2-NX^2+X^3) = \mathbf{O(MNX)}$

- **Count Diagonal**:

  $$O(M(N-X)X) = O((MN-MX)X) = O(MNX-MX^2) = \mathbf{O(MNX)}$$

Per calcolare la valutazione finale avviene un ciclo per scorrere gli array, il suo costo è **O(X)**.
Quindi la valutazione del costo delle euristica è **O(MNX)**

3.2 Alphabeta

Come visto a lezione il costo computazionale dell'alphabeta e del minimax nel caso peggiore non migliora poiché non avvengono potature. Il numero di nodi dipende dal numero di mosse possibili in un turno elevato al numero di turni. Il numero di mosse dipende direttamente dal numero di colonne (N), invece il numero dei turni è la profondità dell'albero (p), quindi il costo computazionale è $\mathbf{O(N^p)}$. Inoltre ad ogni chiamata viene fatto il **middlesort** che ha un costo direttamente proporzionale a **O(N)**,perciò il costo totale dell'**alphabeta** sarà $\mathbf{O(N^p)}$.
Invece nel caso in cui i nodi dovessero essere valutati il suo costo sarebbe di $\mathbf{O(MN^pX)}$

3.3 SelectColumn

Nel codice di selctcolumn è presente un while che itera per il numero di profondità massima che riesce ad esplorare nei 10 secondi. All'interno del ciclo è presente a sua volta la chiamata ad alphabeta, quindi il costo totale è $\mathbf{O(MN^pXp)}$

3.4 Tabella riassuntiva

Funzioni	**Costi**
Count vertical	**O(MN)**
Count horiziontal	**O(MNX)**
Count diagonal	**O(MNX)**
Heuristic	**O(MNX)**
Middlesort	**O(N)**
Alphabeta	**O(MN^pX)**
Selectcolumn	**O(MN^pXp)**

4 Idea di futura implementazione

4.1 Salvare mosse/mappe di gioco

Per migliorare ulteriormente la massima profondità esplorabile e la velocità nel calcolare mossa ottima si dovrebbe aggiungere una tabella hash, che permette di salvare la mappa corrente e successivamente anche il suo valore.
Per fare ciò uno dei metodi più conosciuti è quello della Zobrist map, una tecnica utilizzata nell'ambito della programmazione degli algoritmi di giochi da tavolo, come gli scacchi.
La funzione hash di Zobrist coinvolge l'utilizzo di una tavola hash e numeri casuali precalcolati, noti come chiavi di Zobrist. Ogni volta che lo stato del gioco cambia, si esegue un'operazione XOR tra il valore corrente della tavola hash e la chiave di Zobrist associata alla modifica dello stato.
Questa tecnica è utilizzata per generare un valore hash unico per ogni configurazione di gioco, consentendo una rapida verifica delle posizioni precedenti durante la ricerca di un algoritmo di gioco.
L'obiettivo principale è migliorare le prestazioni nell'individuazione di posizioni già esplorate durante la ricerca di partite. Per capire meglio il suo funzionamento consigliamo il seguente video.

5 Sitografia

L'idea dell'implementazione dell'euristica è spunto del seguente pdf.
Idea dell'iterative deepening: link.