ConnectX

Relazione del progetto per il corso Algoritmi e Strutture di Dati

Cono Cirone 0001029785
Tiziano Rigosi 0000989615

Università degli Studi di Bologna
A.A. 2022/2023

Contents

1 Introduzione

1.1 Descrizione del gioco

ConnectX rappresenta una generalizzazione del gioco Connect4. Contrariamente alla classica board 6 x 7, si sviluppa su una griglia di dimensioni variabili. L'obiettivo rimane invariato: creare una sequenza di pezzi contigui, orizzontalmente, verticalmente o in diagonale, lunga X caselle. Ovviamente il primo dei due giocatori che raggiungerà tale traguardo conquisterà la vittoria.

1.2 Scopo del progetto

L'obiettivo primario di questo progetto è sviluppare un player che dimostri una capacità ottimale nel giocare in tutte le possibili configurazioni. Ciò che rende complesso ConnectX è l'aumento esponenziale del numero di mosse da prendere in considerazione. Quindi, la sfida consiste nel provare a esplorare solo i nodi maggiormente interessanti in modo da riuscire, comunque, a trovare una mossa ottimale pur rispettando i vincoli di tempo imposti dalle specifiche del progetto.

2 Scelte Progettuali

2.1 Metodo `selectColumn()`

Il metodo `selectColumn()` rappresenta il cuore del nostro progetto poiché viene utilizzato per scegliere la colonna da selezionare in ogni turno e, di conseguenza, guida il comportamento del nostro player durante il gioco. Di seguito, esamineremo in dettaglio il funzionamento di questo metodo e le funzioni correlate.

1. `Selezione Iniziale - Metodo center()`: Se la mossa corrente rappresenta il primo turno del player, sfruttiamo il metodo `center()` per posizionare automaticamente la pedina al centro della board. Questa scelta iniziale è strategica poiché aumenta il numero di combinazioni vincenti possibili, contribuendo così ad aumentare la probabilità di vittoria.

2. `Prevenzione Mosse Vincenti Avversarie - Metodo LosingColumn()`: Questo passaggio mira a impedire all'avversario di conseguire una vittoria immediata. Prima di avviare una ricerca più approfondita, verifichiamo se vi sono mosse che potrebbero portare all'immediata vittoria dell'avversario. Il metodo `LosingColumn()` svolge questa verifica: se una mossa vincente dell'avversario è imminente, restituiamo la colonna corrispondente.

3. `Selezione Ottimale - Metodo iterativeDeepening()`: Se nessuno dei casi precedenti si verifica, procediamo con l'utilizzo del metodo `iterativeDeepening()`. Questo approccio sfrutta la ricerca a profondità incrementale per determinare la mossa migliore da effettuare considerando diverse profondità di ricerca. La scelta progettuale fondamentale, in questo caso, è

che, anche quando la mossa è stata determinata tramite `center()` o `LosingColumn()`, avviamo comunque una ricerca con iterative deepening per sfruttare al massimo il tempo disponibile e arricchire le tabelle di trasposizione.

2.2 Algoritmo di Ricerca

Per l'analisi approfondita delle possibili mosse e delle situazioni di gioco, abbiamo implementato l'algoritmo di ricerca Alpha-Beta Pruning, già discusso a lezione. Abbiamo scelto di adottare questa strategia anziché il classico algoritmo Minimax, poiché abbinato a un buon move ordering offre il vantaggio di eliminare interi rami dell'albero di ricerca. Ciò permette di concentrarsi esclusivamente sulle mosse più promettenti, approfondendo ulteriormente la valutazione solo per quelle che mostrano il potenziale di portare a risultati positivi.

Per massimizzare l'efficienza dell'algoritmo Alpha-Beta Pruning, abbiamo introdotto una serie di ottimizzazioni:

1. `History Heuristic:` Questa tecnica mira a migliorare l'ordine in cui le mosse vengono considerate, mettendo in cima alla lista quelle che si sono rivelate maggiormente promettenti in passato. Ciò riduce il numero di mosse analizzate e consente all'algoritmo di esplorare prima le opzioni più probabili di successo.

2. `Transposition Table:` Quando valutiamo una determinata configurazione, anziché ricalcolare interamente il suo valore, possiamo prima cercarla all'interno delle tabelle di trasposizione. Se troviamo un'entry corrispondente, possiamo evitare il ricalcolo completo.

3. `Funzione di valutazione.`

2.2.1 Iterative Deepening

L'algoritmo di ricerca Alphabeta, a sua volta, viene richiamato in `'iterativeDeepening()'`. L'iterative deepening è una strategia di ricerca che inizia con una profondità data e, progressivamente, aumenta questa profondità finché non raggiunge il limite di tempo previsto. Questo approccio permette al nostro player di esplorare il più possibile l'albero delle mosse considerando sempre più rami a ogni iterazione.

2.2.2 Transposition Table

Come precedentemente menzionato, abbiamo adottato l'utilizzo delle tabelle di trasposizione al fine di ottimizzare l'algoritmo di ricerca. Per implementare questa ottimizzazione, ci siamo avvalsi dell'utilizzo dello `Zobrist Hashing`, una tecnica che assegna un hash univoco a ogni elemento della transposition table. L'hashing di Zobrist assegna valori hash casuali agli elementi della tabella, e viene eseguito uno XOR tra la cella e il valore generato casualmente in precedenza ogni volta che una mossa viene effettuata o annullata su una determinata cella della board.

Ogni entry contenuta nella tabella di trasposizione presenta i seguenti attributi:

1. `Key:` rappresenta la chiave hash che identifica in modo univoco la configurazione del gioco corrente. La chiave viene calcolata utilizzando la tecnica dello Zobrist Hashing, che aggiorna il valore attraverso le funzioni `doMove()` e `undoMove()`.

2. `Profondità:` Indica la profondità a cui è stata calcolata l'entry. Questa informazione aiuta a determinare se il valore memorizzato è calcolato a una profondità sufficientemente elevata.

3. `Score:` Rappresenta il punteggio associato alla configurazione del gioco. Questo valore viene utilizzato come stima precisa solo quando si tratta di un valore esatto.

4. `Flag:` Indica il tipo di valore memorizzato, ad esempio, un valore esatto (`hashfEXACT`), un limite superiore (`hashfBETA`) o un limite inferiore (`hashfALPHA`).

Tutti questi attributi sono fondamentali quando si deve recuperare un valore dalla transposition table. Con la funzione `get()`, consideriamo solo i valori a profondità sufficiente e restituiamo un valore diverso in base al flag con cui sono stati salvati.

2.2.3 History Heuristic

L' `History Heuristic` rappresenta una tecnica di dynamic move ordering basata sull'idea che le mosse che storicamente hanno portato a risultati migliori abbiano maggiore priorità nelle esplorazioni successive, poiché si presume siano più promettenti. Nel nostro progetto, l'implementazione della History Heuristic ha dimostrato di essere fondamentale. Un buon move ordering, infatti, migliora notevolmente le prestazioni dell'algoritmo Alpha-Beta Pruning. Per quanto riguarda l'implementazione, abbiamo realizzato la History Table come un array in cui ogni colonna ha un valore associato, determinato dal quadrato del punteggio ottenuto durante le ricerche precedenti. Questa scelta amplifica l'importanza delle colonne che hanno dimostrato di essere vantaggiose nelle fasi precedenti. Nell'ordinamento dell'array all'interno della funzione `HistoryHeuristic()`, abbiamo utilizzato la funzione `Array.sort()` di Java, che si basa sull'algoritmo di ordinamento MergeSort visto a lezione.

2.2.4 Evaluation

Fin da un'osservazione superficiale del problema di valutazione della board emergono immediatamente due possibili soluzioni:

- Valutazione dell'intera board
- Valutazione della singola mossa, quindi delle sole celle adiacenti a quella che andremo a occupare con la nostra mossa.

Abbiamo scelto di valutare la singola mossa per garantire una complessità computazionale minore (lineare rispetto alla lunghezza della sequenza da ottenere), con l'obiettivo di trovare sempre la mossa che avvicina maggiormente il player alla vittoria (lasciando al metodo `LosingColumn()` il compito di impedire vittorie immediate degli avversari). La funzione implementata presenta tre caratteristiche principali:

- Il concetto di `margine`: fissata una colonna in cui eseguire la mossa, consideriamo, per ogni direzione, se ci sono abbastanza spazi vuoti o pezzi propri per raggiungere X pezzi consecutivi (interrompendo il conteggio in caso ci sia un pezzo avversario).
- `best` e `secondbest`: per ogni direzione, calcoliamo un certo valore basato sulle sequenze contigue e non dei pezzi propri. Lo score assegnato alla mossa valutata dipende dalla somma del valore più alto e la metà del secondo valore più alto. Dare maggiore peso al valore più alto e ignorare completamente i due valori più bassi permette di rendere meno omogenei gli score e di considerare le mosse per il loro potenziale di vittoria nei prossimi turni.
- precedenza alla `creazione di sequenze contigue:` per ogni direzione, per ogni tassello strettamente adiacente a quello appena posizionato, viene aumentato il valore del potenziale score di 1; il valore viene poi aumentato del rapporto tra i tasselli non contigui e il margine (sarà quindi sempre una cifra inferiore ad 1). Questo disincentivizza l'algoritmo dal creare tante sequenze frammentate, facilmente isolabili da un tassello avversario, piuttosto che una unica contigua.

3 Complessità Computazionale

Analizzando il metodo `selectColumn()`, emergono chiaramente le diverse complessità computazionali dei vari componenti coinvolti. Innanzitutto, possiamo osservare che il costo della funzione `iterativeDeepening()` costituisce il principale fattore determinante. La funzione `center()` ha un costo costante $O(1)$. La funzione `LosingColumn()`, invece, presenta un costo lineare $O(n)$, dove n rappresenta il numero di colonne presenti sulla board. Per una valutazione accurata del costo di `iterativeDeepening()`, diviene fondamentale analizzare la complessità di `alphabeta()` e `alphabetaCol()`, dato che influenzano notevolmente il costo complessivo. Nel peggiore dei casi, `alphabeta` può richiedere un costo di $O(b^d)$, mentre in situazioni ottimali si riduce a $O(\sqrt{b^d})$, dove b indica quante mosse possono essere effettuate in una data posizione di gioco e d la profondità di ricerca nell'albero di gioco. Nella funzione alphabetacol() viene richiamata alphabeta su ogni colonna, dunque, il costo diviene$O(n \cdot b^d)$, e, successivamente, all'interno di `iterativeDeepening()` questo costo viene moltiplicato per un fattore k, rappresentante il numero di iterazioni di ricerca. Pertanto, il costo finale diventa $O(K \cdot n \cdot b^d)$.

Relativamente agli altri metodi implementati, il costo di `HistoryHeuristic()` è fortemente influenzato dalla fase di ordinamento dell'array, risultando in una complessità di $O(n \log n)$. Inoltre, per quanto riguarda la tabella di trasposizione, sia la funzione `get()` che la `put()` presentano un costo costante $O(1)$, garantendo operazioni di lettura e scrittura efficienti.

4 Possibili Miglioramenti

- `Utilizzo di algoritmi di ricerca più sofisticati:` Un'area in cui il nostro progetto potrebbe trarre notevoli vantaggi è l'adozione di algoritmi di ricerca più avanzati. Oltre all'attuale algoritmo di ricerca Alpha-Beta con iterative deepening, potremmo esplorare algoritmi come il Monte Carlo Tree Search (MCTS) oppure AlphaZero, in quanto entrambi si sono rivelati molto efficaci in progetti di questo tipo.

- `Miglioramento del Move Ordering:` L'efficienza dell'algoritmo di ricerca è fortemente influenzata dall'ordine delle mosse esplorate. Attualmente, utilizziamo la tecnica di History Heuristic per ordinare le mosse, che si basa sull'idea di assegnare priorità alle mosse con le performance storiche migliori. Tuttavia, esistono altre tecniche di ordinamento che potrebbero risultare più efficaci in determinate situazioni, come ad esempio:

  - ProbCut: Questa tecnica combina la ricerca alpha-beta con una strategia di pruning basata sulle probabilità, esplorando le mosse in ordine decrescente di probabilità di successo per evitare di esplorare le mosse meno promettenti.
  - Principal Variation Ordering (PVO): Questo approccio si basa sulla considerazione che la mossa migliore scoperta fino a quel punto è spesso la migliore da esplorare per prima. Durante l'iterative deepening, puoi salvare e mantenere la "linea principale" delle mosse e usarla come guida per l'ordinamento delle mosse nelle iterazioni successive.

- `Valutazione delle Minacce tramite Union-Find:` Una possibile funzione di valutazione dell'intera board potrebbe tener conto efficacemente di tutte le sequenze contigue amiche e nemiche tramite la struttura dati Union-Find, ponendo particolare attenzione alle "minacce", ossia le sequenze di lunghezza tale da poter portare a una vittoria in una o due mosse. Un'implementazione di questo tipo richiederebbe l'implementazione di una funzione di eliminazione efficiente degli elementi, che manca nella struttura dati tipicamente studiata, e presenterebbe sfide nell'integrazione delle informazioni di Union-Find all'interno delle tabelle di trasposizione.

5 Pseudocodice

5.1 Alpha-Beta Pruning

Algorithm 1 Algoritmo Alpha-Beta Pruning

```
1: function ALPHABETA(board, depth, alpha, beta, maximizingPlayer)
2:     color ← 1 if firstPlayer else -1
3:     value ← get(depth, alpha, beta)
4:     if value ≠ valUNKNOWN or checktime() then
5:         return value
6:     if depth = 0 or board.gameState() ≠ CXGameState.OPEN then
7:         if board.gameState() ≠ CXGameState.OPEN then
8:             return +∞ if board.gameState() = myWin else -∞ if
   board.gameState() = yourWin else eval(board, board.getLastMove().j) ×
   color
9:     bestValue ← -∞ if maximizingPlayer else +∞
10:    for column in HistoryHeuristic(board) do
11:        doMove(board, column)
12:        value ← alphabeta(board, depth - 1, alpha, beta,
   not maximizingPlayer)
13:        HistoryTable[column] += value × value
14:        if checktime() then
15:            return timeout_v
16:        if maximizingPlayer then
17:            bestValue ← max(bestValue, value)
18:        else
19:            bestValue ← min(bestValue, value)
20:        if bestValue ≥ beta or bestValue ≤ alpha then
21:            put(depth, bestValue, hashfBETA if maximizingPlayer else hashfALPHA)
22:            return bestValue
23:        if maximizingPlayer then
24:            alpha ← max(alpha, bestValue)
25:        else
26:            beta ← min(beta, bestValue)
27:        undoMove(board)
28:    put(depth, bestValue, hashfBETA if maximizingPlayer else hashfALPHA)
29:    return bestValue
```

La complessità varia da $O(b^d)$ a $O(\sqrt{b^d})$.

5.2 Alpha-BetaCol

Algorithm 2 Algoritmo Alpha-BetaCol

```
1: function ALPHABETACOL(B, depth)
2:     Initialize bestValue and bestMove
3:     for each column i from HISTORYHEURISTIC(B) do
4:         Make move i on board B
5:         if B.gameState() = myWin then
6:             return i                                   ▷ Immediate win
7:         value ← ALPHABETA(B, depth, Integer.MIN_VALUE,
   Integer.MAX_VALUE, false)
8:         Undo move i on board B
9:         if value = timeout_v then
10:            break                          ▷ Stop searching if timeout
11:        if value > bestValue then
12:            Update bestValue and bestMove
       return bestMove                        ▷ Return the best move found
```

`Complessità:` $O(n \cdot b^d)$

5.3 Iterative Deepening

Algorithm 3 Iterative Deepening

```
1: function ITERATIVEDEEPENING(max_depth, B)
2:     Initialize current_depth_limit, previousBestMove, and bestMove
3:     while not CHECKTIME and current_depth_limit ≤ max_depth and
   B.gameState() = CXGameState.OPEN do
4:         Store previous best move as previousBestMove
5:         Find best move using ALPHABETACOL(B, current_depth_limit)
6:         Increment current_depth_limit
7:     if bestMove = −1 then
8:         if previousBestMove = −1 then
9:             Randomly select a safe column to play
10:            return safe column
11:        return previousBestMove          ▷ Return best move found so far
       return bestMove                      ▷ Return the best move found
```

`Complessità:` $O(K \cdot n \cdot b^d)$

5.4 History Heuristic

Algorithm 4 History Heuristic

```
1:  function HistoryHeuristic(B)
2:      width ← Number of available columns in B
3:      int[]rating ← Array of ratings of length width
4:      Integer[]moves ← Available columns in B
5:      for m ← 0 to width − 1 do
6:          rating[m] ← Rating in HistoryTable at position moves[m]
7:      Integer[]indexes ← New array of length width
8:      for i ← 0 to width − 1 do
9:          indexes[i] ← i
10:     Sort indexes in descending order based on rating
11:     int[]orderedMoves ← New array of length width
12:     for i ← 0 to width − 1 do
13:         orderedMoves[i] ← moves[indexes[i]]
14:     return orderedMoves
```

`Complessità:` $O(n \log n)$.

5.5 SelectColumn

Algorithm 5 SelectColumn

```
1:  function selectColumn(B)
2:      START ← Current Time
3:      col ← center(B)
4:      max_depth ← Maximum Search Depth
5:      if col ≠ −1 then
6:          value ← iterativeDeepening(max_depth, B)
7:          return col
8:      possible_lose ← LosingColumn(B)
9:      if possible_lose ≠ −1 then
10:         value ← iterativeDeepening(max_depth, B)
11:         return possible_lose
12:     bestMove ← iterativeDeepening(max_depth, B)
13:     return bestMove
```

`Complessità:`

- `center()`: $O(1)$
- `LosingColumn()`: $O(n)$
- `iterativeDeepening()`: $O(K \cdot n \cdot b^d)$

5.6 Evaluation

Algorithm 6 Evaluation

```
1:  function EVAL(board, col)
2:      best ← 0, secondbest ← 0, last_row ← getX(board, col), s ← board.cellState(last row, col)
3:      for DIR in {vertical, horizontal, diagonal, antidiagonal} do
4:          n ← 1, notconnected ← 0, margin ← 0
5:          if DIR = vertical then
6:              n ← n + number of strictly connected pieces from the top of col to the bottom
7:              margin ← board.M - last_row
8:          else
9:              for k ← 1 to board.X do
10:                 n ← n + number of strictly connected pieces to the left of col, according to DIR
11:                 notconnected ← notconnected + number of unconnected pieces to the left of col, according to DIR
12:                 margin ← margin + number of empty and unconnected pieces to the left of col, according to DIR
13:                 if cellState ≠ s and cellState ≠ CXCellState.FREE then break
14:             for k ← 1 to board.X do
15:                 n ← n + number of strictly connected pieces to the right of col, according to DIR
16:                 notconnected ← notconnected + number of unconnected pieces to the right of col, according to DIR
17:                 margin ← margin + number of empty and unconnected pieces to the right of col, according to DIR
18:                 if cellState ≠ s and cellState ≠ CXCellState.FREE then break
19:         if n + margin < board.X then break
20:         if n ≥ board.X then return Integer.MAX_VALUE
21:         if n + margin ≥ board.X then
22:             if n + notconnected/margin > secondbest then
23:                 best ← Math.max(best, n + notconnected/margin)
24:                 secondbest ← Math.min(best, n + notconnected/margin)
25:     return int(best + secondbest × 0.5)
```

`Complessità:` $O(X)$