LombEnis: ottimizzazione strategica per connectx

Matteo Lombardi: 0001071217
Enis Brajevic: 0001070214

1 Introduzione al problema computazionale

Il progetto in oggetto si concentra sull'implementazione di un algoritmo per il gioco **ConnectX**, una variante del noto gioco Connect 4. Questo progetto affronta una sfida computazionale intrigante, che consiste nell'elaborare una strategia ottimale al fine di effettuare la mossa migliore in un dato contesto di gioco, all'interno di un tempo prestabilito.

La variabilità delle dimensioni della griglia e del numero di pedine richieste per la vittoria rende ConnectX un problema che tocca diversi argomenti nell'ambito degli algoritmi e delle strutture di dati. Il nostro obiettivo principale in questo progetto è stato sviluppare un algoritmo capace di valutare rapidamente la situazione di gioco attuale e selezionare la mossa ottimale per massimizzare le probabilità di vittoria o minimizzare possibile sconfitta, a seconda del contesto.

In questa relazione esamineremo approfonditamente il problema computazionale di ConnectX. Discuteremo le sfide specifiche associate al gioco, gli approcci utilizzati per affrontarle e i risultati ottenuti tramite l'implementazione dell'algoritmo. Sarà presentata un'analisi delle **strategie algoritmiche** impiegate per affrontare questo problema, comprese le strutture dati utilizzate per rappresentare i processi decisionali dell'algoritmo e per migliorarne la velocità di esecuzione. Infine valuteremo in maniera approssimativa il **costo computazionale** dell'algoritmo allo scopo di comprenderne l'efficienza.

2 Scelte progettuali adottate

In questa sezione, esamineremo le **scelte progettuali** adottate per la creazione di un algoritmo preciso ed efficiente. Si tenga presente che il codice sottostante a queste scelte è stato accuratamente organizzato e commentato per garantirne la leggibilità e la comprensibilità, qui forniremo una visione più ad alto livello delle scelte chiave, le quali sono state suddivise in tre fasi distinte per una migliore comprensione del processo di sviluppo.

2.1 Scelta dell'algoritmo di decisione

La prima fase del progetto ha comportato la selezione dell'**algoritmo di decisione** da adottare.

Sin dall'inizio, abbiamo optato per l'utilizzo dell'algoritmo **minimax**, riconosciuto come adatto per giochi di strategia in cui l'obiettivo è minimizzare le perdite e massimizzare le possibilità di vittoria.

Tuttavia, data la variabilità delle dimensioni della griglia di gioco (MxN) e del numero di pedine necessarie per vincere (X), diventa computazionalmente impossibile valutare tutte le possibili combinazioni di gioco in un tempo finito, soprattutto per quanto riguarda valori di M N X molto grandi. Per affrontare questa sfida abbiamo dunque introdotto l'**iterative deepening**, un approccio che chiama l'algoritmo minimax con una crescente profondità di ricerca, adattandosi al tempo disponibile. La scelta della mossa migliore si basa sull'ultimo albero decisionale completamente esplorato prima dello scadere del tempo.

2.2 Valutazione della situazione di gioco

La seconda fase si è concentrata sulla **valutazione** delle situazioni di gioco non finali. Per farlo abbiamo sviluppato un algoritmo che, dato lo stato attuale della tabella di gioco, restituisce un valore attribuendo peso alle nostre sequenze e alle sequenze avversarie valutate in tutte le direzioni e a patto che siano aperte da almeno un lato. La lunghezza di una sequenza determina il peso attribuito, con pesi crescenti in base alla lunghezza (ad esempio, una sequenza da 4 pedine vale di più di due sequenze da 2 pedine).

Questa valutazione della singola situazione di gioco è fondamentale per la corretta operatività dell'algoritmo ed è determinante nello stabilire il livello di precisione delle scelte selezionate dall'algoritmo.

2.3 Ottimizzazione generale dell'algoritmo

La terza fase ha riguardato l'**ottimizzazione** generale dell'algoritmo per accelerare la selezione della mossa e consentire una maggiore profondità di ricerca nell'albero decisionale nello stesso intervallo di tempo.

Inizialmente, abbiamo implementato la tecnica dell'**alpha-beta pruning**, che ci ha permesso di tagliare gran parte dell'albero decisionale, riducendo notevolmente il tempo di calcolo. Successivamente, abbiamo introdotto una **transposition table** la quale si basa sul concetto di caching per evitare di valutare più volte situazioni di gioco identiche. Nella realizzazione di tale tabella è stata utilizzata una hashmap, la quale come sappiamo sfrutta il concetto di hashtable con liste concatenate che vengono convertite in alberi AVL quando tali liste assumono una dimensione elevata. Il generatore delle chiavi necessarie per ac-

cedere agli elementi della tabella utilizza le informazioni sulla disposizione delle pedine sulla griglia di gioco, insieme ai valori attuali di alpha e beta. In questo modo, ogni situazione di gioco differente viene associata a una chiave univoca. Per valore viene invece memorizzata la valutazione effettuata tramite alpha-beta pruning relativa al nodo nell'albero di decisione.

Tuttavia, la più significativa ottimizzazione è stata l'implementazione di una euristica leggera di **prevalutazione**, che ha migliorato ulteriormente l'efficienza dell'alpha-beta pruning permettendoci di raggiungere altezze molto elevate con l'iterative deepening. Questa prevalutazione è stata applicata in due momenti:

1. In primo luogo è stato inserito un ordinamento delle prime n chiamate all'alpha-beta (con n riferito al numero delle colonne disponibili su cui poter inserire la prossima pedina), il quale sfrutta come prevalutazione il precedente passo dell'iterative deepening. Per fare ciò viene salvato ad ogni iterazione il valore di ogni colonna all'interno di una coda con priorità, al fine di poterla sfruttare poi nel passo successivo, quando l'altezza dell'albero di decisione è stata incrementata.

2. In secondo luogo abbiamo inserito una prevalutazione all'interno dell'alpha-beta pruning stesso, valutando, prima di effettuare le chiamate ricorsive, la situazione attorno alla cella in cui si desidera posizionare la prossima pedina. Questa euristica considera le sequenze favorevoli che si vanno ad ampliare e quelle nemiche che si vanno a bloccare, fornendo dei pesi in base alla loro lunghezza.

3 Valutazione approssimativa del costo computazionale

Per ottenere una valutazione approssimativa del costo computazionale del nostro algoritmo, esamineremo attentamente ciascuna delle fasi coinvolte al funzionamento di questo.

Partiamo dall'analizzare il costo del metodo **iterative deepening**, il quale ci consente di esplorare l'albero di decisione progressivamente, incrementando l'altezza ad ogni iterazione. Il costo computazionale di tale metodo è prevalentemente influenzato dall'ultima esecuzione completa delle iterazioni che avvengono al suo interno, in cui si raggiunge un'altezza massima. Questo dato, come precedentemente menzionato, è estremamente difficile da stimare in quanto dipende da due variabili chiave: il tempo massimo durante il quale è possibile prendere una scelta e le prestazioni specifiche del computer su cui viene eseguito l'algoritmo.

Passiamo ora ad analizzare il costo della **valutazione** di una specifica situazione di gioco al fine di attribuirgli un valore. Questa euristica controlla ogni

cella della colonna di gioco al fine di controllare quante e quali sequenze di pedine sono presenti in essa, dunque tale valutazione assume un costo di $O(MN)$. Questo deve poi essere moltiplicato per ogni nodo dell'albero al quale viene svolta la valutazione, che senza ottimizzazioni particolari sappiamo essere corrispondente al numero di foglie dell'albero di decisione, ovvero $O(N^{MN})$. Abbiamo dunque trovato che il costo della valutazione dell'albero di gioco è equivalente a $O(MNN^{MN})$.

Analizziamo ora il costo dell'implementazione delle varie **ottimizzazioni** utilizzate, per poi solo al termine valutare i vantaggi che queste hanno portato e considerare la somma di questi nel costo computazionale totale.

Per quanto riguarda l'utilizzo della **transposition table** questo è stato realizzato con efficienza, avvalendosi di una tabella hash per effettuare operazioni di ricerca e inserimento, i quali hanno un costo $O(1)$ nel caso medio. Sappiamo inoltre che ciascuna delle operazioni di inserimento e rimozione nella tabella hash presenta anche la generazione di una chiave data una certa situazione di gioco. Il metodo che svolge questa operazione, ovvero generateKey, prende in considerazione tutte le celle della tabella di gioco al fine di costruire una chiave unica, dunque il costo dell'utilizzo di tale metodo, che domina ciascun inserimento o rimozione nella tabella hash, è $O(MN)$. Poiché ogni nodo dell'albero viene inserito o estratto dalla transposition table, il costo totale dell'uso di questa struttura dati è $O(MNN^{MN})$, concordando con il costo della valutazione dell'albero.

Valutiamo infine la **prevalutazione**, la quale è stata integrata per ordinare nel migliore dei modi la valutazione dei nodi, al fine di migliorare l'efficacia dell'alpha-beta pruning. Il costo dell'utilizzo di tale ottimizzazione comprende il costo dell'inserimento e rimozione dalla coda con priorità di ogni nodo dell'albero ad eccezione delle foglie, le quali vengono valutate direttamente e non hanno bisogno di una prevalutazione. Sappiamo dunque che tali operazioni hanno un costo di $O(NlogN)$, in quanto il numero di massimo di elementi che vengono inseriti nella coda con priorità corrispondono al numero di colonne su cui poter inserire la prossima pedina. Per ogni inserimento nella coda avviene inoltre una chiamata alla funzione preEvaluate, la quale svolge un'analisi delle sequenze adiacenti alla cella in cui si vuole inserire la prossima pedina, dunque il costo di tale metodo è $O(X)$, dove X rappresenta il numero di pedine necessarie per vincere la partita, dunque la lunghezza massima di una sequenza di pedine, la quale è stimabile essere $O(\sqrt{M^2 + N^2})$. Otteniamo dunque che il costo computazionale dell'utilizzo della prevalutazione risulta essere $O(NlogN\sqrt{M^2 + N^2}N^{MN-1}) = O(logN\sqrt{M^2 + N^2}N^{MN})$

Nel **caso pessimo**, tenendo conto di tutte le valutazione che abbiamo svolto, il costo computazionale complessivo del nostro algoritmo è stimato in $O(MNN^{MN} + MNN^{MN} + logN\sqrt{M^2 + N^2}N^{MN}) = O((MN + logN\sqrt{M^2 + N^2})N^{MN})$, ovvero la somma dei costi di valutazione con minimax, transposition table e pre valutazione.

Nel **caso ottimo**, invece, le ottimizzazioni implementate conducono a un miglioramento significativo, consentendo di saltare la valutazione di gran parte dell'albero di decisione. Per quanto riguarda il potenziamento ottenuto dall'implementazione della transposition table, quest'ultimo è molto difficile da stimare, ma ciò che invece può essere valutato è il miglioramento ottimale portato dalla prevalutazione. Quest'ultima infatti, nel caso in cui consenta di ordinare i nodi dell'albero in modo da valutare per primo sempre il sottoalbero migliore, porta, come visto a lezione, uno speed up quadratico sul numero dei nodi dell'albero di decisione, il quale dunque diventa $O(\sqrt{M^{MN}})$. Possiamo dunque concludere che il costo computazionale complessivo del nostro algoritmo nel caso ottimo è stimato essere $O((MN + logN\sqrt{M^2 + N^2})\sqrt{N^{MN}})$.