Progetto Algoritmi a.s. 2021/2022

Team

Xuanqiang Huang : 0001030271
Giovanni Spadaccini : 0001021270

Riassunto

Abbiamo utilizzato di un algoritmo di Minimax euristico con alpha-beta pruning per la risoluzione del gioco mnk, una forma generalizzata del tris. L'algoritmo utilizza l'euristica di valutazione per scoprire l'ordine di esplorazione di un numero limitato di nodi, le esplora fino a un livello di profondità prefissato, dopo il quale ritorna un valore euristico o un valore finale, nel caso in cui la board sia terminale. Dai test fatti in locale, l'algoritmo sembra avere capacità simili o superiori a quelle di un umane per le tavole accessibili ai limiti umani (ossia da 10 in giù).

Introduzione

Il gioco proposto è una versione generalizzata del gioco (nought and crosses) conosciuto più generalmente in occidente come tris, o gomoku in Giappone: un gioco a due giocatori su una tavola simile a quanto in immagine, a turni, in cui bisogna allineare un certo numero di pedine del prioprio giocatore al fine di vincere.

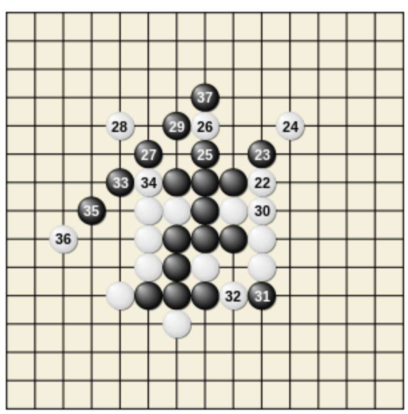

Figure 1: Esempio di tavola di gioco di gomoku in cui il giocatore nero ha vinto

Seguendo la caratterizzazione di un ambiente di gioco di Russel e Norvig[1] possiamo definire il gioco come un ambiente deterministico, multigiocatore, con informazioni complete, sequenziale, statico, conosciuto e discreto. Questa descrizione ci ha permesso di avere una prima idea di quali algoritmi potessero essere utilizzati per la risoluzione del gioco, in quanto in letteratura questo problema, e problemi simili, sono stati risolti con tecniche che ormai possiamo considerare *classiche*.

Sotto questa logica abbiamo scelto l'implementazione di un minimax euristico.

L'algoritmo ad alto livello

Il nostro algoritmo euristico minimax fa una stima iniziale di quanto può esplorare e in seguito utilizza un ordinamento delle mosse secondo una euristica per decidere l'ordine di eslorazione delle mosse ed esplora un numero limitato di celle fino a una profondità limitata.

Argomenti trattati

- *Markcell e unmarkCell*
  - Versione più veloce e cache-friendly per segnare le celle come marcate e per annullare quanto marcato
- *Euristica*
  - Spiegazione, calcolo e utilizzo dell'euristica per il nostro minimax
- *Ordinamento mosse*
  - Sull'algoritmo utilizzato per ordinare le mosse a seconda del valore restituito dall'euristica
- *Timer test*
  - Sull'algoritmo usato per avere una stima della quantità di nodi esplorabili
  - Sui metodi di allocazione di un numero di mosse preciso alle celle scelte per l'esplorazione
  - Sulla scelta dei valori di ramificazione e profondità

Markcell e unmarkCell

Problema

Trovare le celle libere e le celle occupate, dopo varie esecuzioni di markCell e unmarkCell.

Algoritmi considerati

1. riallocarci ogni volta un array con le freeCell meno quella appena fatta (nel caso di markCell), o di crearci un array con tutte le freeCell con anche l'ultima mossa fatta (nel caso di unmarkCell), costo $O(n)$

2. utilizzo di una linked list per tenerci i valori. Questo approccio nonostante avesse le operazioni di insert e remove in tempo constante performava peggio di riallocarsi un array ogni volta (che ha costo $O(n)$) (pensiamo che questo sia dovuto alle ottimizzazioni cache delgli array)

3. utilzzo di hash table e set

Algoritmo utilizzato

Abbiamo ideato un sistema che è in grado di fare `markCell`, `unmarkCell` in tempo costante in caso ottimo, pessimo e medio, senza considerare i checks aggiuntivi per verificare lo stato del gioco e l'aggiornamento dell'euristica.

Al fine di raggiungere questa velocità, l'insieme delle mosse eseguite è tenuto alla fine di un array che contiene tutte le mosse , in modo simile a quanto fa una heap, studiata durante il corso, al momento di rimozione. Queste mosse sono tenute come se fossero uno stack, e possono essere ripristinate con semplici assegnamenti. Questa implementazione migliora rispetto alla board del codice iniziale nel caso pessimo, in quanto non deve più necessitare di un hashtable, il cui caso pessimo è $O(n)$ con n la grandezza della table.

Al fine di raggiungere questa velocità, l'insieme delle mosse eseguite è tenuto alla fine di un array che contiene tutte le mosse, questo array contiene inizialmente tutte le celle libere. Mano a mano che le celle vengono utilizzate fa lo swap con l'ultima cella dell'array e si ricorda la posizione in cui era, così facendo quando viene chiamata l'unmarkCell riesce a riposizionarsi nella sua vecchia posizione.

Nota: tutte le celle contengono un **index** che indica la posizione dell'array in cui è contenuta la mossa, altrimenti, se questa non è ancora stata rimossa indica la posizione di ritorno.

Algorithm 1: markCell senza checks sulla board

Result: Cella richiesta della tavola è marcata
Input : int freeCellsCount: numero di celle libere, è compreso fra 1 e allCells.length
Input : int index: l'index della cella da marcare, compresa fra 0 e freeCellsCount
Input : Cell[] allCells: array tutte le celle, in cui le prime freeCellsCount sono considerate libere
Output: void: viene modificata allCells

```
allCells[freeCellsCount - 1].index = allCells[index].index;
swap(allCells[freeCellsCount - 1], allCells[index]);
// marca la cella come occupata dal giocatore
mark(allCells[freeCellsCount - 1]);
freeCellsCount = freeCellsCount - 1;
```

Algorithm 2: unmarkCell senza checks sulla board

Result: Viene rimosso l'ultima mossa della tavola
Input : int freeCellsCount: numero di celle libere, è compreso fra 0 e allCells.length - 1
Input : Cell[] allCells: array tutte le celle, in cui le prime freeCellsCount sono considerate libere

```
// marca la cella come libera
markFree(allCells[freeCellsCount - 1]);
swap(allCells[allCells[freeCellsCount].index], allCells[freeCellsCount]);
allCells[freeCellsCount].index = freeCellsCount;
freeCellsCount = freeCellsCount + 1;
```

L'Euristica

In questo progetto sono fondamentali le euristiche utilizzate al fine del successo del Player. Esiste una unica euristica che viene utilizzata, in modi diversi, sia per la valutazione della board sia per la scelta delle mosse.

MICS - Minimum Incomplete Cell Set

L'euristica del MICS racchiude in sé 3 informazioni principali:

1. Quanto è favorevole una cella secondo aperture e pedine mie e nemiche.
2. In modo analogo calcolo stesso valore per il nemico.
3. La somma dei due valori precedenti mi dà una stima di criticità della singola cella (senza la presenza di doppi-giochi e fine-giochi)

Con questo valore numerico è possibile ordinare le mosse secondo un ordine di priorità.

Questa euristica è una versione modificata dell'euristica proposta da Nathaniel Hayes and Teig Loge [3]

Calcolo del MICS con le sliding window

Rilevamento dei doppio-giochi e fine-giochi

Con il sistema a sliding window possiamo anche rilevare con molta facilità alcune celle particolari *critiche* ossia situazioni di doppi giochi oppure giochi ad una mossa dalla fine.

> Definiamo **fine-giochi** le celle per cui esiste almeno una sliding window a cui manca 1 mossa per vincere

È chiaro come queste celle siano molto importanti sia per noi, al fine della vittoria, sia per il nemico, al fine di bloccarli.

> Definiamo **doppi-giochi banali** le celle per cui esistono due o più sliding window per cui mancano 2 mosse per vincere

Se abbiamo una tale configurazione, si può notare come muovere su quella cella riduce le mosse per vincere di 1 in entrambe le sliding window. Abbiamo allora due sliding window in cui manca una mossa per vincere, per cui il nemico può bloccare al massimo una, garantendoci la vittoria sull'altra.

Riguardo i doppi-giochi non banali, ossia doppi giochi in cui abbiamo bisogno di 3 o più mosse per vincere ci basiamo sulla capacità del minimax di scovarli, non siamo riusciti a trovare un modo per codificare questo caso tramite le sliding-window.

Punteggi per configurazioni di doppio-gioco e fine-gioco

Sono assegnati alcuni punteggi speciali alle celle di doppio-gioco o fine-gioco.

Queste configurazioni non sono espressamente visibili al MICS, per cui abbiamo assegnato dei valori fissi a *gradini*, ossia in qualunque modo si calcoli il MICS, il valore euristico di questo non può superare il valore assegnato da una cella di doppio gioco, e quest'ultima non può superare il valore di una cella di fine-gioco.

Quindi in ordine di importanza abbiamo:

1. Cella di fine-gioco
2. Cella di doppio-gioco banale
3. Cella valutata dall'euristica del MICS + punteggi allineamento e vicinanza.

Punteggi per allineamento e vicinanza

Con prove empiriche abbiamo notato che l'euristica del MICS non è in grado di valutare correttamente alcune situazioni di allineamento, per cui abbiamo aggiunto dei moltiplicatori di punteggi per l'allineamento di celle e per favorire le mosse vicine ad alcune celle già allineate.

Si possono osservare i valori di questi moltiplicatori `MY_CELL_MULT` e `ADIACENT_MULT` rispettivamente nel file di `DirectionValue` e `HeuristicCell`.

Questi valori si sono rilevati fondamentali per il gioco intelligente del player.

Ordinamento delle mosse

Problema

Su un array di `n` celle libere, dobbiamo ordinare le prime `l` che hanno il valore più alto

Approcci presi in considerazione

Ad ogni momento dalla board dobbiamo riuscire a vedere le migliori `l` celle. Per farlo abbiamo guardato vari metodi.

1. Il primo sorting normale che andava in $O(n \log n)$ dove n sono le celle libere
2. Quick select, che non andava bene perchè separa i `l` elementi più grandi ma non sono oridnati, quindi ci sarebbe costato $O(n + l \log l)$. E tempo d'esecuzione peggiore però sarebbe stato $O(n^2 + l \log l)$.

Algoritmo Utilizzato

Il metodo che abbiamo utilizzato sfrutta una leggera variazione dell'algoritmo di heap-select: si scorre l'array delle celle libere tenendosi una heap di massimo `l` elementi di questa, e infine svuota la heap e la mette in un array che contiene le prime `l` celle sortate. Costo computazionale $O(n \log l)$

Questo miglioramento sui primi test è riustito a far esplorare la board 5 o 6 volte più mosse a parità di tempo in input.

Un esempio di utilizzo di questo algoritmo lo si può trovare in `updateCellDataStruct` della Board. In questo caso utilizziamo la variabile `branchingFactor` per tenere il valore di `l` molto basso.

Algorithm 3: Algoritmo di ordinamento delle mosse

Input : int l, numero elementi della heap
Input : array: array di elementi comparabili di lunghezza n t.c. n > l
Output: un array con le l elementi maggiori di array

```
queue = new MinPriorityQueue();
for element in array do
    if queue.size() < l then
        add element to queue;
    else if get first element of queue < element then
        delete top of queue;
        add element to queue;
    end
end
// dopo il ciclo abbiamo esattamente l elementi nella queue
out = new Array of l elements;
int i = l - 1; while queue.size() > 0 do
    out[i] = get first element of queue;
    delete top of queue;
    i = i - 1;
end
return out;
```

Timer Test

Problema

Al fine di utilizzare tutto il tempo computazionale a disposizione era presente la necessità di trovare un metodo che permettesse di esplorare i nodi interessanti sfruttando al meglio il tempo a disposizione.

Approcci provati

1. timer su ogni nodo del minimax, questo approccio non funzionava perché c'era il rischio che al primo livello venisse esplorato un singolo nodo, dato che si andava in profondità subito.
2. Esplorazione di una parte di albero di ampiezza e profondità prefissata: questo approccio portava il rischio del tuning dei parametri di profondità e ampiezza, che potevano cambiare a seconda del calcolatore.

Soluzione utilizzata

Alla fine abbiamo utilizzato idee da entrambi i metodi, creando una simulazione del processo di decisione che esplorasse quanti nodi più possibili e dasse una stima di quanti era possibile visitare, mantenendo sempre delle costanti di profondità e ampiezza prefissati per le varie tipologie di board.

Al fine di avere una stima di quanti nodi di ricerca una macchina remota con un limite di tempo prefissato abbiamo utilizziamo la classe `TimingPlayer`,

Dopo l'esecuzione di questa, avremo un numero di nodi esplorati nel limite di tempo dato, e utilizzeremo questo numero trovato durante l'init per decidere quanti nodi può esplorare il player vero.

Spartizione e utilizzo del numero di mosse

Problema

Come distribuire il numero di mosse, nell'esplorazine del minimax.

Soluzione utilizzata

Abbiamo visto che in seguito all'ordering del MICS le prime celle sono quelle più importanti da esplorare, quindi vogliamo distribuire più mosse alla prima cella, in modo che possa avere una esplorazione più approfondita.

In `findBestMove` vediamo come sono utilizzati il numero di celle trovate in questo modo. Nel caso in cui la prima cella non utilizzi tutte le celle date, queste saranno affidate alle celle di esplorazione successive. La cella di esplorazione successiva può, quindi, esplorare un numero di celle uguale a `numero nodi non utilizzati precedenti + addendo di nuove celle da esplorare`. Così per le prime `branchingFactior * 4` con i valori euristici più grandi.

Scelta del fattore di ramificazione e profondità

Problema

I fattori di ramificazione e profondità hanno un inpatto diretto sul tempo di esecuzione dell'algoritmo, e sulla qualità della mossa scelta Diventava quindi molto importante trovare i valori corretti da assegnare per ogni board.

Soluzione utilizzata

Eravamo coscienti della possibilità di utilizzare metodi di apprendimento automatico al fine di trovare in questi valori.

Tuttavia non eravamo a conoscenza dei metodi di applicazione nel nostro ambiente, né se i valori potevano dipendere dal calcolatore su cui veniva eseguito il programma.

Abbiamo quindi deciso di utilizzare alcuni valori fissati, che abbiamo trovato in base a delle prove empiriche.

TODOS

Alcune cose importanti che si dovrebbero fare?

- [x] pseudocodice di markCell
- [x] pseudocodice di unmarkCell
- [x] spiegazione delle FreeCell
- [ ] Spiegazione dell'euristica
  - [ ] 1. Spiegazione dell'ordering delle mosse -> cenno a Late Move reduction (o citazione del paper di MICS)
  - [ ] 2. Spiegazine dell'eval della board
  - [ ] 3. Note: sul pruning utile grazie a questo ordering
- [x] algoritmo di sorting delle celle
- [ ] spiegazione calcolo dell'euristica
- [x] spiegazione del timer test (numero dei nodi cercati)
- [x] spiegazione dei valori euristici per l'esplorazione in depth e in weight
- [ ] Minimax

Algorithm 1

Just a sample algorithmn

Algorithm 4: While loop with If/Else condition

Result: Write here the result
Input : Write here the input
Output: Write here the output

```
while While condition do
    instructions;
    if condition then
        instructions1;
        instructions2;
    else
        instructions3;
    end
end
```

Approcci fallimentari

1. Simulazione di Montecarlo (MCTS), dato il grande successo di AlphaGo
    1. Guardava celle che avevano poco valore per la vittoria
    2. Guardava tutti gli stati ad ogni livello, il che pesava molto anche sulla memoria (poichè si teneva tutto il game tree)
    3. Il limite di tempo era troppo basso per avere un numero di simulazioni sufficenti
    4. La capacità dell'hardware incideva molto sui risultati.
2. Puro algoritmo heuristico (anche conosciuto come Greedy best first Search)
    1. Non riusciva ad andare in profondità, dato che selezionava ogni volta la cella con maggiori probabilità di vittoria al primo livello, questo non gli permetteva di pianificare le proprie mosse.
3. alpha beta pruning puro, per i test grandi impiegava troppo tempo d'esecuzione, non riuscendo a fare l'eval di neanche una mossa
4. rule-based strategy[2], basato su 5 steps che riporto qui testualmente: Rule 1 If the player has a winning move, take it. Rule 2 If the opponent has a winning move, block it. Rule 3 If the player can create a fork (two winning ways) after this move, take it. Rule 4 Do not let the opponent create a fork after the player's move. Rule 5 Move in a way such as the player may win the most number of possible ways.
    1. Queste regole sono state molto importanti come guida del nostro progetto, nonostante non siano applicate in modo esplicito, hanno guidato il valore fisso per le celle di doppio-gioco e fine-gioco.
5. Iterative Deeping

Miglioramenti possibili

1. Utilizzare un sistema ad apprendimento automatico per decidere il `BRANCHING_FACTOR` e la `DEPTH_LIMIT` che ora sono di valori fissati, secondo l'esperienza umana.

2. Utilizzare più threads per l'esplorazione parallela dell'albero di ricerca (non possibile per limiti imposti).

Conclusione

Abbiamo osservato come un classico algoritmo Minimax con alpha-beta pruning possa giocare in modo simile, o superiore rispetto all'essere umano per le board di grandezza adeguata per l'umano (≤ 10 per M e N), data una euristica che gli permetta di potare ampi rami di albero.

Appendice

Algorithm 5: Minimax Euristico riadattato da Norvig e Russel [1]

```
Function MiniMax-Search(game, state)
    // Cerca fra le mosse la migliore possibile
    // move è l'azione da intraprendere
    value, move = Max-Value(game, state);
    return move;

Function Max-Value(game, state)
    if game.terminalTest(state) then return game.utility(state), null;
    value, move = -∞;
    for each action in game.actions(state) limitato da una branching factor
      do
        v, m = Min-Value(game, game.result(state, action));
        if v > value then
            value, move = v, action;
        end
    end
    return value, move;

Function Min-Value(game, state)
    if game.terminalTest(state) then return game.utility(state), null;
    value, move = ∞;
    for each action in game.actions(state) limitato da una branching factor
      do
        v, m = Max-Value(game, game.result(state, action));
        if v < value then
            value, move = v, action;
        end
    end
    return value, move;
```

References

1. Russell, Stuart J., e Peter Norvig. Artificial Intelligence: A Modern Approach. Fourth edition, Global edition, Pearson, 2022.

2. Development of Tic-Tac-Toe Game Using Heuristic Search IOP Publishing, 2nd Joint Conference on Green Engineering Technology & Applied Computing 2020, Zain AM, Chai CW, Goh CC, Lim BJ, Low CJ, Tan SJ

3. Developing a Memory Efficient Algorithm for Playing m, n, k Games, Nathaniel Hayes and Teig Loge, 2016.