Intelligenza Artificiale
Agente per il gioco tic-tac-toe

Nicola Carlesso (1237782) Niccolò Vettorello

a.a. 2019/2020

Contents

Abstract

In tale documento mostreremo come sono stati sviluppati i due agenti per giocare a tic-tac-toe, rispettivamente coi linguaggi *Haskell v.3.8.8* da Nicola Carlesso e in *Python v.3.8* da Niccolò Vettorello. Infine evidenzieremo le principali caratteristiche tra i due.

1 Introduzione

Tic-tac-toe è un gioco deterministico ad informazione completa, infatti nel gioco non è presente alcun elemento di casualità ed ogni giocatore è a conoscenza di tutto ciò che avviene nella nella griglia di gioco.

Un agente, per poter giocare a tic-tac-toe, deve utilizzare l'algoritmo `Minimax` $\alpha - \beta$ `pruning`

1.1 Algoritmo Minimax $\alpha - \beta$ pruning

Tale algoritmo utilizza come struttura dati un albero n-ario dove in ogni nodo è presente uno stato del gioco, nel contesto del gioco tic-tac-toesono tutte le possibili griglie creabili. La radice di quest'albero è, rimanendo nell'esempio di tic-tac-toe, la griglia vuota. Ogni livello di profondità dell'albero contiene tutte le possibili configurazioni dopo che il giocatore A o B hanno effettuato una mossa, come mostrato in Figura 1.

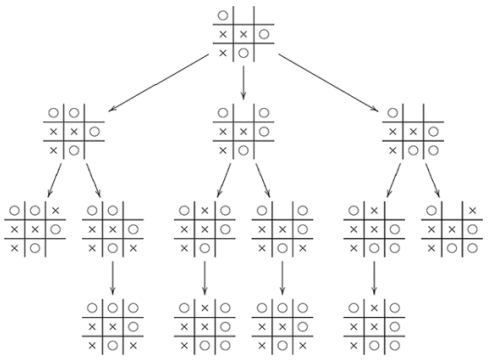


Figure 1: Esempio di albero nell'algorimo `Minimax`.

Supponendo che il computer sia il giocatore A, l'algoritmo `Minimax`, esplorando sempre pù in profondità l'albero, assegnerà ad ogni livello in cui sono presenti le mosse di A con `MAX`, ed i livelli dell'albero con le mosse di B con l'etichetta `MIN`. Questo perché vuole calcolare il massimo vantaggio quando muove A ed il minimo svantaggio quando muove B.

Alle configurazioni finali (le foglie dell'albero) viene poi assegnato un valore di utilità:

- **+1** in caso di vittoria di A;

- **0** in caso di pareggio;
- **-1** in caso di sconfitta di A.

L'algoritmo dunque, lavorando ricorsivamente, per ogni nodo di un dato livello di profondità dell'albero, in base all'etichetta che questo possiede, verrà assegnato un valore di utilità pari al minimo (per nodi etichettati `MIN`) o il massimo (`MAX`) dei valori di utilità dei nodi figli, come mostrato in Figura 2 (il valore di utilità del giocatore "O" ha un valore di utilità -1, mentre quello per il giocatore "x" +1), dove il nodo radice possiede l'etichetta `MIN`.

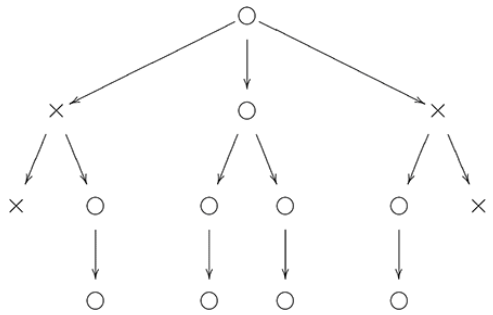


Figure 2: Etichette dei nodi nell'albero mostrato in Figura 1.

L'algoritmo `Minimax` risulta essere *completo* nel caso in cui il gioco arriva sempre ad un termine (avitando situazioni di stallo), perché analizza tutti gli scenari possibili, e dunque risulta anche essere *ottimo*.
Considerando b come il fattore di branching dell'albero ed m il numero massimo di mosse per concludere una partita, l'algoritmo `Minimax` ha complessità in tempo $\mathcal{O}(b^m)$ e complessità in spazio di $\mathcal{O}(bm)$.

Non è il caso ditic-tac-toe, ma in alcuni giochi l'algoritmo `Minimax` può portare alla creazione di alberi molto grandi, chiedendo un tempo di esecuzione non accettabile.

Per superare questo inconveniente si applica inizialmente la tecnica dell'$\alpha-\beta$ `pruning` che consiste nel potare i "rami" dell'albero che portano a mosse non migliori di quelle già trovate. Nell'esempio a Figura 2, l'algoritmo, che sta calcolando i valori di utilità dei nodi figli della radice, dopo aver calcolato il valore di utilità del nodo figlio più a destra (X) e di quello al centro (O), eviterà poi di esplorare il sotto-albero del nodo figli di destra, dato che possiede un valore peggiore dei rami precedentemente analizzati. Questa tecnica può portare in caso di *ordine perfetto* dei nodi un complessità in tempo di $\mathcal{O}(b^{\frac{m}{2}})$.

Un altro metodo per ridurre la complessità dell'algoritmo è quello di impostare un profondità massima dell'albero, impedendogli però così di non raggiungere sempre gli stadi finali del gioco, facendo perdere all'algoritmo le proprietà della *completezza* e *correttezza*.

2 Agente sviluppato in Haskell

2.1 Breve introduzione ad Haskell

Per comprendere il funzionamento dell'agente, si basti sapere del linguaggio *Haskell*[1] è un linguaggio *funzionale puro* ed utilizza la *lazy evaluation*. Un linguaggio funzionale prevede la creazione solo di funzioni, perciò non vengono utilizzate variabili o oggetti di alcun tipo. La lazy evaluation prevede invece di valutare i parametri di una funzione non subito, ma solo quando questi vengono utilizzati nel corpo della funzione, come mostrato nella derivazione 1 di `inc n = n + 1`, a differenza della più usata *eager evaluation* che valuta prima tutti i parametri di una funzione prima di eseguirla, come mostrato nella derivazione 2.

$$inc(2 * 3) \rightarrow (2 * 3) + 1 \rightarrow 6 + 1 \rightarrow 7 \tag{1}$$

$$inc(2 * 3) \rightarrow inc\ 6 \rightarrow 6 + 1 \rightarrow 7 \tag{2}$$

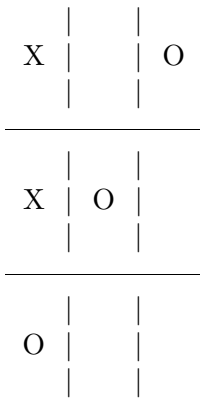
2.2 Sviluppo dell'agente

L'agente è stato sviluppato seguendo le indicazioni del libro di testo del corso *Functional Languages*.

L'agente è suddiviso in 7 moduli presenti in 7 file differenti:

- **Type.hs** : contiene tutte le definizioni delle tipi utilizzati dall'agente, le più importanti sono
  - `Player` : dato che indica il giocatore, può dunque assumere il valore X, O oppure B (blank) per indicare una casella della griglia vuota. Per il buon funzionamento dell'algoritmo `Minimax`, dato che il computer gioca sempre come X, ai giocatori è stato dato l'ordine $O < B < X$;
  - `Grid` : una matrice $3x3$ di *Player*;
  - `Tree` : struttura necessaria per l'algoritmo `Minimax`, dove ogni nodo possiede un valore ed una lista di nodi figli; le foglie sono dunque in nodi che possiedo una lista vuota di nodi figli;
- **GridUtilities.hs** : contiene una serie di funzioni utili per gestire la griglia di gioco, come la creazione di una griglia vuota;
- **DisplayGrid.hs** : contiene le funzioni necessarie per stampare nel terminale la griglia che durante lo svolgimento del gioco cambia;

[1] https://en.wikipedia.org/wiki/Haskell_(programming_language)



- **PromptUtilities.hs** : modulo con funzioni, che in ausilio col modulo *DisplayGrid*, contiene gestiscono l'interazione attraverso il prompt del giocatore col computer;
- **MoveUtilities.hs** : modulo con le funzioni che modificano restituiscono la griglia risultante dopo una determinata mossa, e la funzione `moves` che data una griglia ed un *Player*, restituisce tutte le possibili mosse del giocatore indicato. Tale funzione è fondamentale per costruire l'albero usato dall'algoritmo `Minimax`;
- **Minimax.hs** : modulo centrale per il funzionamento dell'agente. Tale modulo contiene le funzioni per eseguire l'algoritmo `Minimax` con $\alpha - \beta$ pruning in due versioni differenti
  - **miniman** : versione dell'algoritmo che risulta essere una semplice traduzione dello pseudo-codice mostrato in Figura 3 e dunque restituisce per ogni nodo dell'albero il valore di utilità del suo sotto-albero;
  - **minimaxPruning** : versione modificata della funzione *minimax* che prevede l'utilizzo della tecnica dell'$\alpha - \beta$ pruning;

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

Figure 3: Pseudo-codice dell'algoritmo minimax.

entrambe queste funzioni vengono chiamate dalla funzione `bestmove` che ricevendo una lista di nodi col proprio valore di utilità e il *Player* che deve muovere, scorre la lista finché non trova un valore di utilità pari al valore del *Player* dato. La funzione `bestmove` si occupa anche di limitare la profondità dell'albero grazie alla funzione *prune*, la quale data una profondità ed un albero, "taglia" quest'ultimo alla profondità indicata;

- **TicTacToe.hs** : modulo principale che possiede la funzione `main` per avviare l'agente e la funzione `tictactuo` per giocare con un'altra persona.

Per poter giocare a tic-tac-toecon l'agente o con un altro giocatore le mosse devono essere date attraverso un intero da 0 a 9 nel seguente modo:

0 | 1 | 2
---
3 | 4 | 5
---
6 | 7 | 8

2.3 Utilizzo della Lazy evaluation

In primo luogo la *lazy evaluation* risulta utile nel momento in cui si effettua il taglio in profondità dell'albero, infatti, grazie alla lazy evaluation, non viene tagliato un albero già più profondo della profondità indicata, ma quest'ultimo viene creato fino a che la profondità indicata nella funzione `prune` non viene raggiunta.

Un altro aspetto che interessante che porta la lazy evaluation si può vedere nella funzione `bestmove`, che come detto prima analizza in modo iterativo i sotto-alberi di un nodo contenuti in una lista, fermando l'analisi nel momento in cui trova il valore desiderato. La funzione, dato che è lazy, non calcola tutti i sotto-alberi per poi analizzarli iterativamente, bensì calcola il valore del primo sotto-albero e se questo non restituisce il valore desiderato calcola quello successivo, altrimenti la funzione si ferma. Questo è esattamente il procedimento che prevede la tecnica $\alpha - \beta$ pruning, che grazie alla lazy evaluation si ottiene in automatico. Infatti, le prestazioni dell'agente utilizzando le funzioni `minimax` e `minimaxPruning` sono uguali.

	`minimax`	`minimaxPruning`
Tempo di esecuzione 1m̂ossa (s)	6.823	6.927
Tempo di esecuzione 2m̂ossa (s)	0.118	0.135
Tempo di esecuzione 3m̂ossa (s)	0.011	0.011
Memoria utilizzata (GB)	1.36	1.39