MONKey

Relazione del progetto per l'insegnamento di Algoritmi e strutture di dati

Gaia Clerici (#971338), Stefano Volpe (#969766)

Università di Bologna
1 settembre 2021

Figura 1: una scimmia (foto di Bob Brewer)

Fa' la brava scimmietta.
L'uomo con il cappello giallo

Indice

1 Specifiche

1.1 Il gioco: *m,n,k-game*

Il gioco *m,n,k-game* è deterministico, a turni, a due giocatori, a somma zero e con informazione perfetta. In una partita, i due agenti si alternano nel marcare una cella vuota in una griglia di dimensione $m \times n$ con un simbolo del proprio colore. Se un giocatore allinea in orizzontale, verticale o diagonale almeno k simboli, questi vince la partita e il suo avversario la perde. Se non rimangono più celle vuote sulla griglia, la partita finisce in pareggio.

1.2 Il torneo: la classifica dei giocatori

Ogni volta che, all'interno del torneo, un giocatore conclude una partita, egli guadagna:

- 3 punti in caso di una vittoria come secondo giocatore, ma non a tavolino;
- 2 punti in caso di una vittoria come primo giocatore o a tavolino;
- 1 punto in caso di un pareggio;
- 0 punti in caso di una sconfitta.

Le regole del torneo considerano una vittoria "a tavolino" quando l'avversario non restituisce una mossa entro il tempo limite (approssimativo) di 10 secondi, o comunque sceglie un mossa illegale (già occupata o esterna alla griglia). Non è dato conoscere l'identità dell'avversario. Per ognuna delle configurazioni in tabella 1, ciascun agente gioca esattamente quattro partite contro ogni altro partecipante, di cui due come primo giocatore e due come secondo giocatore.

1.3 L'obiettivo: il giocatore

Lo scopo del progetto è lo sviluppo di un'intelligenza artificiale in Java per un giocatore di *m,n,k-game*. Se qualità della risposta e costo temporale sono prioritari, lo stesso non vale per il costo in memoria, anche se devono comunque essere evitati sprechi. Infine, a parità dei fattori di cui sopra, la precedenza spetta alle strategie concettualmente più semplici.

1.4 L'interfaccia: `mnkgame.MNKPlayer`

L'interfaccia che l'intelligenza artificiale deve implementare è contenuta nel pacchetto `mnkgame`. Oltre a un metodo per la selezione della mossa desiderata, ne viene concesso un altro invocato in fase di inizializzazione prima di ogni partita.

M	N	K
3	3	3
4	3	3
4	4	3
4	4	4
5	4	4
5	5	4
5	5	5
6	4	4
6	5	4
6	6	4
6	6	5
6	6	6
7	4	4
7	5	4
7	6	4
7	7	4
7	5	5
7	6	5
7	7	5
7	7	6
7	7	7
8	8	4
10	10	5
50	50	10
70	70	10

Tabella 1: configurazioni previste dal torneo.

2 Analisi del problema

2.1 Il valore di gioco teorico

Molte istanze del problema in questione si presentano come giochi a sé stanti, talvolta arrichiti con limitazioni imposte dalle regole: ne sono esempi Tris, Go-Moku e Renju. Per alcune configurazioni, il valore di gioco teorico è già stato dimostrato. Van den Herik, Uiterwijk e van Rijswijck [13] hanno raccolto tali risultati dalla letteratura precedente in tabella 2.

Tramite l'argomento del "furto di strategia", usato per la prima volta da John Nash nel 1949 [2], si dimostra che nessuno dei valori di gioco teorico non ancora aggiunto alla tabella è una vittoria per il secondo giocatore.

mnk-game (k=1,2)	Vittoria per il primo giocatore
333-game (Tris)	Pareggio
mn3-game ($m \geq 4, n \geq 3$)	Vittoria per il primo giocatore
m44-game ($m \leq 8$)	Pareggio
mn4-game ($m \leq 5, n \leq 5$)	Pareggio
mn4-game ($m \geq 6, n \geq 5$)	Vittoria per il primo giocatore
mn5-game ($m \leq 6, n \leq 6$)	Pareggio
15,15,5-game (Go-Moku)	Vittoria per il primo giocatore
mnk-game ($k \geq 8$)	Pareggio

Tabella 2: valori di gioco di *m,n,k-game*

2.2 L'albero di gioco

Van den Herik, Uiterwijk e van Rijswijck [13] classificano *m,n,k-game* come di "categoria 3", ovvero con un'alta complessità dello spazio degli stati e una bassa complessità dell'albero di gioco. Per poter quantificare meglio il numero di stati effettivamente appartenenti al nostro albero di gioco, faremo uso del numero s di celle della griglia come indice della dimensione dell'istanza del problema. Esso vale:

$$s = m \times n \tag{1}$$

Cominciamo osservando che il numero di figli f di un qualsiasi nodo non terminale a profondità p coincide con il numero di celle rimaste vuote, e cioè:

$$f(p) = s - p \tag{2}$$

Se assumessimo che tutte le foglie dell'albero si trovassero a profondità s, potremmo facilmente calcolare il numero di nodi n a una data profondità p:

$$n(p) = \frac{s!}{f(p)!} \tag{3}$$

Quindi, sempre sotto questa ipotesi, il numero di nodi dell'albero sarebbe dato da:

$$\overline{n_{tot}} = \sum_{p=0}^{s} n(p) = \sum_{p=0}^{s} \frac{s!}{f(p)!} = s! \sum_{p=0}^{s} \frac{1}{p!} = \Theta(s!) \tag{4}$$

Una stima per difetto dell'effettivo numero di nodi è fornita dall'albero di gioco le cui partite terminano tutte nel minor numero di mosse possibili. Essendo ragionevole assumere

$$m, n > 1 \wedge k \leq m, n \tag{5}$$

si ha $2k - 1 < s$ e quindi:

$$n_{tot} = \Omega((2k-1)!) \tag{6}$$

Una stima per eccesso dell'effettivo numero di nodi è fornita dall'albero di gioco le cui partite terminano tutte nel maggior numero di mosse possibili:

$$n_{tot} = O(s!) \tag{7}$$

2.3 Gli allineamenti

La seguente formula permette di calcolare il numero a di allineamenti orizzontali (a_{hor}), verticali (a_{ver}) e diagonali (a_{dia}) di lunghezza k in una griglia di dimensione $m \times n$:

$$\begin{aligned} a &= a_{hor} + a_{ver} + a_{dia} = \\ &= m(n-k+1) + n(m-k+1) + 2(n+m-1)(\min(m,n)-k+1) = \\ &= \Theta(s) \end{aligned} \tag{8}$$

3 Strumenti

Durante lo sviluppo del progetto, sono stati impiegati i seguenti strumenti:

- Vim e Visual Studio Code come *editor* di codice;
- `make` per la compilazione automatizzata;
- OpenJDK Runtime Environment 11.0.12 e Java™ SE Runtime Environment 15 come piattaforme Java;
- Git per il controllo di versione;
- GitHub Actions per l'integrazione e lo sviluppo continui;
- AlmaStart per la ricerca integrata nella base di dati del sistema bibliotecario di ateneo;
- LaTeX 2_ε per la preparazione della relazione di progetto.

Per disporre di ulteriore supporto computazionale, si è infine fatto uso da remoto delle macchine gestite dal Dipartimento.

4 Scelte progettuali

Oltre a `MoNKey.java`, che implementa `mnkgame.MNKPlayer`, e `Tester.java`, classe di collaudo del progetto, `monkey` disponde di tre sottopacchetti:

- `monkey.util` fornisce strutture dati e algoritmi di uso generale ma non presenti nella libreria standard Java;

- `monkey.ai` definisce un'intelligenza artificale per un generico gioco a turni a due giocatori i cui stati siano istanze di un'unica classe implementante `monkey.ai.State`;
- `monkey.mnk` espone la rappresentazione di *m,n,k-game* implementando `monkey.ai.State`.

4.1 `monkey.util`

Questo pacchetto sopperisce alle mancanze della libreria standard con i metodi per il calcolo del minimo e del massimo fra due elementi, nonché con una mappa implementata come tabella ad indirizzamento diretto [12]. Da notare che, fatta eccezione per l'inizializzazione della suddetta mappa (che ha un costo temporale lineare nella propria capacità), tutti i metodi di questo pacchetto hanno costo costante.

4.2 `monkey.ai`

L'intelligenza artificiale progettata propone due diversi algoritmi per la scelta della mossa: il primo effettua una ricerca all'interno dell'albero di gioco, mentre il secondo esegue un più veloce ma meno accurato confronto fra le mosse al momento disponibili. Quest'ultimo è pensato per le configurazioni troppo impegnative per il primo algoritmo. Spetta a `MoNKey.java` decidere a quale metodo fare affidamento basandosi su s.

4.2.1 Ricerca nell'albero di gioco

Nonostante Herik, Uiterwijk e Rijswijck [13] raccomandino per i giochi di "categoria 3" l'uso di metodi basati sulla conoscenza, abbiamo optato per una scelta più tradizionalista: una sintesi di più varianti della potatura alfa-beta, che si colloca invece nei metodi a forza bruta.

In primo luogo, viste le imposizioni sul tempo di esecuzione, il nostro algoritmo ha necessariamente bisogno di un limite di profondità d. L'interfaccia delle due funzioni per una "classica" potatura alfa-beta [6] risulta quindi essere:

- MAX-VALUE(s, α, β, d): applica la ricerca al nodo che rappresenta lo stato s assumendo che esso sia di massimo;
- MIN-VALUE(s, α, β, d): applica la ricerca al nodo che rappresenta lo stato s assumendo che esso sia di minimo.

In entrambi i casi, α e β fungono da parametri della potatura, mentre d è il limite di profondità imposto. La mutuale ricorsione di queste due procedure è responsabile di un'esplorazione a profondità limitata dell'albero; tale ricerca supporta il dietrofront [7] (*backtracking*) tramite le funzioni RESULT(s, a) e REVERT(s). La loro implementazione non viene riportata in quanto piuttosto canonica, eccezion fatta per ripetuti controlli sullo scadere del tempo di gioco e

per l'uso di una tabella delle trasposizioni [3]. Tale tavola *hash* è a due livelli e fa uso dello schema di rimpiazzamento TwoBig1: Breuker, Uiterwijk e van den Herik [4] [5] ne hanno dimostrato la superiorità rispetto a Deep, New, Old, Big1, BigAll e TwoDeep.

La variante della potatura alfa-beta scelta, che fa uso di Max-Value e Min-Value, è la ricerca del nodo migliore (*best node search*, o *BNS*) di Rutko [10], che porta a prestazioni migliori rispetto a tutte le varianti precedenti, vale a dire PVS, Negascout, NegaC*, SSS*, Dual*, MTD(f). Il nostro algoritmo presenta varie differenze rispetto alla versione originale:

- il chiamante specifica il limite di profondità d desiderato;
- il calcolo degli inizializzatori per α e β viene delegato nelle linee 1 e 2 allo stato corrente s, che può così effettuare considerazioni specifiche per il gioco in questione;
- proprio come per Max-Value e Min-Value, gli unici sottoalberi ispezionati sono quelli considerati "rilevanti" da s (vedi 4.3.3);
- se più sottoalberi superano una prova, *best-node* ne memorizza sempre il primo. Questo permette di approfittare del fatto che l'interfaccia `monkey.ai.State` promette un'iterazione sulle azioni rilevanti in ordine decrescente di aspettative;
- l'algoritmo originale, ispirandosi a Negamax, sostituiva alla chiamata a Min-Value della linea 9 una a Max-Value con argomenti e valore resituito negati di segno. La nostra accezione di "giochi a somma zero" designa però i giochi a somma costante, per i quali tale espediente non è utilizzabile nel caso generale: evitarlo ci permette di fare uso di funzioni di valutazione degli stati asimmetriche rispetto ai due giocatori;
- la nostra esplorazione dell'albero, come già detto, supporta il dietrofront (linea 13).

```
BEST-NODE-SEARCH(s, d)
 1  α = INITIAL-ALPHA(s)
 2  β = INITIAL-BETA(s)
 3  subtree-count = COUNT-RELEVANT-CHILDREN(s)
 4  repeat
 5      best-node = NIL
 6      test = NEXT-GUESS(α, β, subtree-count)
 7      better-count = 0
 8      for ogni azione rilevante a di s
 9          if MIN-VALUE(RESULT(s, a), test − 1, test, d) ≥ test)
10              better-count = better-count + 1
11              if best-node == NIL
12                  best-node = a
13          REVERT(s)
14      if better-count == 0
15          β = test
16      elseif better-count > 1
17          subtree-count = better-count
18          α = test
19  until (β − α < 2 or better-count == 1) and better-count ≠ 0
20  return best-node
```

```
NEXT-GUESS(α, β, subtree-count)
1  return α + (β−α)(subtree-count−1) / subtree-count
```

Il limite di profondità d è stabililto da una ricerca ad approfondimento iterativo. Rispetto alla versione proposta da Russel e Norvig [8], la nostra differisce sotto alcuni aspetti e ne espande altri che erano stati originariamente solo accennati dagli autori:

- l'intervallo dei possibili limiti di profondità ha un estremo superiore finito la cui scelta può dipendere da considerazioni specifiche su un gioco e uno stato particolari (linea 2);
- qualora una chiamata a BEST-NODE-SEARCH segnali lo scadere del tempo, viene restituita la mossa calcolata con la più profonda ricerca completata;
- la linea 1 memorizza una copia dello stato di gioco di partenza. In questo modo, in caso di terminazione urgente dovuta allo scadere del tempo di gioco, è possibile ripristinare alla linea 8 tale copia con costo temporale costante. Nessuna pila di chiamate BEST-NODE-SEARCH, MAX-VALUE e MIN-VALUE dovrà così affannarsi a invertire l'effetto di più mosse di gioco una per una.

Lo pseudocodice che segue impiega un semplice sistema di gestione delle eccezioni alla linea 5. Questo ha permesso di codificare BEST-NODE-SEARCH facendo affidamento al principio dell'"acchiappa e rilascia", e ignorando così la

gestione delle scadenze temporali, che viene delegata a ipotetiche implementazioni di MAX-VALUE e MIN-VALUE. Ovviamente, è comunque possibile (e fruttuoso a livello di prestazioni) fare affidamento a valori restituiti dalle varie funzioni che segnalino questo tipo di occorrenze.

```
ITERATIVE-DEEPENING-SEARCH(s)
 1  backup-state = COPY(s)
 2  max-limit = OVERESTIMATED-HEIGHT(s)
 3  res = NIL
 4  for d = 0 to max-limit
 5      try
 6          res = BEST-NODE-SEARCH(d)
 7      catch TIMEOUT-EXCEPTION
 8          s = backup-state
 9          if res ≠ NIL
10              return res
11          // fallback action
12          return FIRST-RELEVANT-ACTION(s)
13  return res
```

In memoria sarà necessario solamente conservare due stati di gioco alla volta, la tabella delle trasposizioni, la pila implicita delle chiamate ($O(d)$) e altre variabili di costo in memoria costante.

4.2.2 Confronto fra mosse

Questa secondo modo di individuare una mossa promettente ne estrae una in modo pseudorandomico fra quelle che, applicate allo stato attuale, massimizzano la funzione di valutazione. Anche questo algoritmo supporta il dietrofront e si limita ad analizzare le azioni considerate "rilevanti".

```
IMMEDIATE-SEARCH(s)
 1  best-moves = ∅
 2  max-eval = NIL
 3  for ogni azione rilevante a di s
 4      current-eval = EVAL(RESULT(s, a))
 5      REVERT(s)
 6      if max-eval == NIL
 7          max-eval = current-eval
 8          best-moves = best-moves ∪ {a}
 9      elseif current-eval ≥ max-eval
10          if current-eval > max-eval
11              max-eval = current-eval
12              best-moves = ∅
13          best-moves = best-moves ∪ {a}
14  return RANDOM(best-moves)
```

Banalmente, il numero di nodi esaminati da IMMEDIATE-SEARCH coincide con le azioni rilevanti per lo stato corrente; nel caso peggiore, in cui la funzione di valutazione assegna lo stesso valore a ciascuna di esse, *best-moves* le memorizzerà tutte.

4.3 `monkey.mnk`

La rappresentazione dello stato di un *m,n,k-game* fa ampio uso dell'approccio incrementale: la maggior parte delle informazioni derivate riguardanti la situazione attuale vengono salvate nella rappresentazione stessa. In questo modo, non devono essere calcolate ogni volta da capo né quando vengono richieste più volte, né quando lo stato viene minimizzato.

4.3.1 Stato iniziale

In fase di costruzione, vengono istanziati (i costi riportati tra parentesi sono sia temporali che spaziali):

- una matrice rappresentante la griglia di gioco ($\Theta(s)$);
- un elenco di tutte le posizioni di gioco in ordine di valore decrescente ($\Theta(s)$);
- tabelle ad indirizzamento diretto che tengano traccia delle condizioni di ogni possibile allineamento di lunghezza k, $k-1$ o $k-2$ ($\Theta(s)$);
- una matrice di codici pseudorandomici per l'*hashing* ($\Theta(s)$);
- altre variabili in quantità costante ($\Theta(1)$).

Sia il tempo che la memoria richiesti appartengono quindi alla classe di costo $\Theta(s)$. Si noti che neanche un'implementazione minimale della sola griglia di gioco potrebbe rimanere in una classe di costo inferiore.

4.3.2 Inizializzatori per α e β

Basandosi sulla tabella 2, è possibile calcolare inizializzatori per α e β in tempo e spazio costanti.

4.3.3 Azioni

Ciascuno stato di gioco fornisce un'iterazione sulle proprie mosse legali e rilevanti di costo complessivo $\Theta(s)$ nel caso pessimo e $\Theta(1)$ nel caso ottimo. L'ordinamento è già stato determinato staticamente in fase di costruzione: è quello della "chiocciola", vale a dire a spirale a partire dal centro. In questo modo, viene data più rilevanza alle posizioni centrali, generalmente considerate più forti. In maniera analoga alla ricerca per schemi [13], ci si limita alla enumerazione delle mosse appartenenti a un certo "schema di minacce" ψ considerato l'area attualmente degna di nota della griglia. In questa implementazione, viene

congetturato che essa coincida con l'insieme delle celle vuote adiacenti anche in diagonale a una piena o, qualora non ve ne fossero, al singoletto contenente solo la posizione centrale.

4.3.4 Modello di transizione

Ogni volta che un simbolo viene aggiunto/rimosso dalla griglia, le seguenti operazioni vengono effettuate:

- aggiornamento della cella coinvolta ($\Theta(1)$);
- aggiornamento dei contatori delle minacce influenzate ($\Theta(k)$);
- aggiornamento dei contatori di celle piene adiacenti ($\Theta(1)$);
- eventuale aggiornamento dell'esito della partita ($\Theta(1)$);
- aggiornamento della cronologia di gioco ($\Theta(1)$);
- aggiornamento del codice *hash*.

Complessivamente, il costo in tempo è di $\Theta(k)$, mentre quello in memoria è costante.

4.3.5 Funzione di valutazione

Al fine di evitare approcci rudimentali come le euristiche di Shevchenko e Chua Hock Chuan, `monkey.mnk.Board` fa uso della seguente versione semplificata della funzione di valutazione di Abdoulaye-Houndji-Ezin-Aglin [1]:

$$\begin{aligned} f = {} & 100p_{k-2,2} + 80p_{k-1,1} + 250p_{k-1,2} + 1000000p_k \\ & - 1300q_{k-2,2} - 2000q_{k-1,1} + 5020q_{k-1,2} + 1000000q_k \end{aligned} \tag{9}$$

nella quale $p_{i,1}$ è il numero di minacce semiaperte di dimensione i del giocatore; $p_{i,2}$ è il numero di minacce aperte di dimensione i del giocatore; p_i è il numero di minacce senza buchi di dimensione i del giocatore; $q_{i,1}$ è il numero di minacce semiaperte di dimensione i dell'avversario; $q_{i,2}$ è il numero di minacce aperte di dimensione i dell'avversario; q_i è il numero di minacce senza buchi di dimensione i dell'avversario. Grazie all'approccio incrementale, tale calcolo viene sempre effettuato in tempo e spazio costanti.

4.3.6 Funzione di *hashing*

Il metodo di *hashing* proposto da Zobrist [14] è semplice, in tempo costante e accompagnato da una letturatura ben più corposa delle sue controparti come BHC. Usando un seme costante per la generazione dei disgiunti, è possibile verificarne la validità staticamente. Per esempio, è utile accertarsi che tale seme generi disgiunti distinti due a due.

Per aumentare il numero di ricerche con successo all'interno della tabella delle trasposizioni, a trasposizioni simmetriche (fino a otto su una griglia quadrata, o fino a quattro in caso contrario) viene assegnato lo stesso codice *hash*, e sempre in tempo costante. Questo permette di evitare di esplorare più volte sottoalberi di fatto equivalenti. Gli effettivi vantaggi di questo accorgimento sono stati dimostrati sperimentalmente da Schiffel [11].

5 Conclusioni

Anche se MONKey è stato in grado di giocare in maniera più che competitiva contro un giocatore umano, quantificarne le prestazioni complessive non si è rivelato possibile. La mancanza di un'infrastruttura comune agli studenti del corso per il collaudo dei propri progetti sotto forma di torneo informale ha inciso negativamente sulle nostre capacità di valutazione obiettiva del lavoro svolto. Dal punto di vista dell'analisi del costo computazionale, l'informazione mancante è un'accurata stima asintotica del numero $n_{BNS}(d)$ dei nodi ispezionati da Best-Node-Search con limite di profondità d. Esso, come nel caso delle altre varianti della potatura alfa-beta, viene di norma misurato solo sperimentalmente. Al momento tali ricerche sono state effettuate solo per la versione originale dell'algoritmo, e solo per alberi con fattore di ramificazione omogeneo. Se in futuro si dovesse proseguire con le ricerche in questa direzione, si potrebbe derivare il numero di nodi raggiunti da Iterative-Deepening-Search al variare di s:

$$n_{IDS}(s) = \sum_{i=0}^{s} n_{BNS}(i) \tag{10}$$

Questo porterebbe quindi al calcolo del costo temporale della procedura Iterative-Deepening-Search:

$$T(s) = n_{IDS}(s)\Theta(k) = \Theta(n_{IDS}(s)k) \tag{11}$$

6 Ricerche future

6.1 Alcune strategie ancora inutilizzate

Fra le tecniche di ricerca non adottate in questo progetto ma che potrebbero migliorarne i risultati ricordiamo:

- il metodo Monte Carlo;
- la ricerca di quiescienza;
- l'estensione singola;
- analisi retrograda [9];

- la ricerca per schemi;
- la ricerca per prova numerica;
- la ricerca dello spazio delle minacce;
- la λ-ricerca [13].

6.2 Nel contesto del torneo

Il progetto ha trattato esclusivamente strategie circoscritte alla singola partita: espedienti sul lungo periodo, orientati ad un piazzamento il più alto possibile in classifica, sono stati ignorati. Possibili spunti per ricerche future includono:

- "fare lo gnorri": davanti ad una sconfitta come primo giocatore reputata quasi inevitabile, resituire una mossa illegale o stallare fino allo scadere del tempo permette di far guadagnare all'avversario un punto in meno (si veda 1.2);
- "congiurare": consiste in una cospirazione orchestrata da una maggioranza sufficientemente grande di intelligenze artificiali partecipanti al torneo. Tramite un uso oculato del "fare lo gnorri", è sempre possibile assicurare a uno dei propri membri il primo posto in classifica. Deve però essere possibile riuscire a identificare quest'ultimo quando vi si gioca contro, per esempio istruendolo in precedenza affinché giochi un'apertura inconsueta.

6.3 Approcci differenti

È utile inoltre osservare come, se presentate in un insegnamento diverso, ad esempio un corso di calcolo numerico o approfondimento profondo, le stesse specifiche avrebbero potuto stimolare a soluzioni completamente differenti.

Riferimenti bibliografici

[1] Abdel-Hafiz Abdoulaye et al. «Generic heuristic for the mnk-games». In: ott. 2018, pp. 265–275.

[2] Elwyn R. Berlekamp, Richard K. Guy e John H. Conway. *Winning ways for your mathematical plays. Volume 2.* [CRC Press], 2018. ISBN: 978-04-294-8732-3. URL: http://ezproxy.unibo.it/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=cat05251a&AN=at.UBO7164935&lang=it&site=eds-live&scope=site.

[3] D. M. Breuker, J. W. H. M. Uiterwijk e H. J. Van Den Herik. «Information in Transposition Tables». In: *ADVANCES IN COMPUTER CHESS*. 1970, pp. 199–211. URL: http://ezproxy.unibo.it/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=edsbl&AN=CN020689428&lang=it&site=eds-live&scope=site.

[4] Dennis Breuker, Jos Uiterwijk e H. Herik. «Replacement Schemes for Transposition Tables». In: *ICCA Journal* 17 (feb. 1970). DOI: 10.3233/ICG-1994-17402.

[5] Breuker Dennis M., Uiterwijk Jos W. H. M. e Herik H. Jaap van den. «Replacement Schemes and Two-Level Tables.» In: *J. Int. Comput. Games Assoc* 19.3 (1996), pp. 175–180. URL: http://ezproxy.unibo.it/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=edsdbl&AN=edsdbl.journals.icga.BreukerUH96&lang=it&site=eds-live&scope=site.

[6] S.J. Russell, P. Norvig e F. Amigoni. «Intelligenza artificiale. Un approccio moderno». In: Intelligenza artificiale. Un approccio moderno v. 1. Pearson Italia, 2010, pp. 200–202. ISBN: 9788871922287. URL: http://ezproxy.unibo.it/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=cat05251a&AN=at.UBO0290346&lang=it&site=eds-live&scope=site.

[7] S.J. Russell, P. Norvig e F. Amigoni. «Intelligenza artificiale. Un approccio moderno». In: Intelligenza artificiale. Un approccio moderno v. 1. Pearson Italia, 2010, p. 108. ISBN: 9788871922287. URL: http://ezproxy.unibo.it/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=cat05251a&AN=at.UBO0290346&lang=it&site=eds-live&scope=site.

[8] S.J. Russell, P. Norvig e F. Amigoni. «Intelligenza artificiale. Un approccio moderno». In: Intelligenza artificiale. Un approccio moderno v. 1. Pearson Italia, 2010, pp. 109–111. ISBN: 9788871922287. URL: http://ezproxy.unibo.it/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=cat05251a&AN=at.UBO0290346&lang=it&site=eds-live&scope=site.

[9] S.J. Russell, P. Norvig e F. Amigoni. «Intelligenza artificiale. Un approccio moderno». In: Intelligenza artificiale. Un approccio moderno v. 1. Pearson Italia, 2010, pp. 207–211. ISBN: 9788871922287. URL: http://ezproxy.unibo.it/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=cat05251a&AN=at.UBO0290346&lang=it&site=eds-live&scope=site.

[10] Dmitrijs Rutko. «Fuzzified algorithm for game tree search with statistical and analytical evaluation.» In: (2011), pp. 90–111. URL: http://ezproxy.unibo.it/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=edseul&AN=edseul.2000091630846&lang=it&site=eds-live&scope=site.

[11] Stephan Schiffel. «Symmetry Detection in General Game Playing.» In: vol. 2. Gen. 2010.

[12] Cormen Thomas H. «Introduzione agli algoritmi e strutture dati.» In: McGraw-Hill Italy, 2010, pp. 209–210. ISBN: 978-88-386-9771-5. URL: http://ezproxy.unibo.it/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=cat05251a&AN=at.UBO7083448&lang=it&site=eds-live&scope=site.

[13] H.Jaap van den Herik, Jos W.H.M. Uiterwijk e Jack van Rijswijck. «Games solved: Now and in the future». In: *Artificial Intelligence* 134.1 (2002), pp. 277–311. ISSN: 0004-3702. DOI: https://doi.org/10.1016/S0004-3702(01)00152-7. URL: https://www.sciencedirect.com/science/article/pii/S0004370201001527.

[14] A.L. Zobrist. «A New Hashing Method With Application for Game Playing.» In: (2012). URL: http://ezproxy.unibo.it/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=edsoai&AN=edsoai.ocn799905211&lang=it&site=eds-live&scope=site.