UNIVERSITÀ DEGLI STUDI DI TORINO

DIPARTIMENTO DI MATEMATICA GIUSEPPE PEANO

SCUOLA DI SCIENZE DELLA NATURA

Corso di Laurea in Matematica

Tesi di Laurea Triennale

MCTS e videogiochi: un'applicazione per le Gare Pokémon Live

Relatore: Roberta Sirovich

Candidato: Matteo Silvestro

ANNO ACCADEMICO 2014/15

Sommario

L'algoritmo MCTS è stato utilizzato con successo per giochi come Go, simulando molte partite in modo casuale e creando un albero di gioco. Può essere ulteriormente migliorato utilizzando l'UCT, un algoritmo di selezione delle mosse, trovando un compromesso tra esplorazione di nuove strategie e sfruttamento delle migliori.

In questa tesi viene spiegato il principio di funzionamento dell'algoritmo. Inoltre, si discute come possa essere applicato con successo anche ai videogiochi e se ne mostra un utilizzo pratico.

Il gioco scelto come applicazione sono le Gare Pokémon Live, in cui quattro giocatori si sfidano in una esibizione che dura cinque turni. Ogni turno i partecipanti eseguono una mossa per impressionare il pubblico o per infastidire gli avversari. Essendo un gioco a turni ma con scelta contemporanea delle mosse e elementi casuali, è utile per mostrare l'efficienza dell'MCTS anche al di fuori di giochi a turni sequenziali deterministici come il Go.

Infine, se ne analizza l'efficienza, confrontando le prestazioni dell'algoritmo contro giocatori casuali, giocatori MCTS di livello diverso (in base al numero di simulazioni effettuate) e giocatori dell'intelligenza artificiale del gioco originale.

Ringraziamenti

Desidero ringraziare la mia relatrice, la professoressa Roberta Sirovich, per la sua disponibilità ed entusiasmo nel seguirmi in questo progetto fuori dagli schemi, oltre che per avermi trasmesso la passione per la probabilità e la statistica.

Ringrazio di cuore Constanza Amanda Pelissero per aver dedicato il suo tempo alla creazione della grafica del programma e avermi sostenuto e consigliato nello svolgimento della tesi.

Infine, un ringraziamento va anche alla mia famiglia, per il supporto che non è mai mancato, e a tutti i professori che mi hanno fatto amare la matematica.

Indice

1 Cenni sull'intelligenza artificiale applicata ai giochi 3
1.1 Teoria dei giochi . . . 3
1.1.1 Giochi in forma estesa . . . 3
1.1.2 Classificazione dei giochi . . . 4
1.2 AI per i giochi . . . 5
1.2.1 Cenni sul minimax . . . 5
1.2.2 Scacchi . . . 5
1.2.3 Go . . . 6
1.3 Metodi Monte Carlo . . . 6
1.3.1 Calcolo di integrali . . . 7
1.3.2 Caratteristiche . . . 7
1.4 Multi-armed bandit problem . . . 7
1.4.1 Rimorso . . . 8
1.4.2 Upper Confidence Bounds (UCB1) . . . 8

2 Monte Carlo Tree Search 9
2.1 MCTS . . . 9
2.1.1 L'algoritmo MCTS . . . 9
2.1.2 Strategie delle fasi . . . 10
2.2 Upper Confidence Bounds for Trees (UCT) . . . 11
2.2.1 L'algoritmo UCT . . . 11
2.2.2 Sfruttamento contro esplorazione . . . 11
2.2.3 Convergenza al minimax . . . 12
2.3 Caratteristiche . . . 12
2.3.1 Non euristico . . . 12
2.3.2 Sempre arrestabile . . . 13
2.3.3 Asimmetrico . . . 13

3 Applicazione dell'MCTS per le Gare Pokémon Live 15
3.1 Implementazione dell'MCTS . . . 16
3.1.1 Albero di gioco e nodi . . . 16
3.1.2 MCTS . . . 17
3.2 Le Gare Pokémon Live . . . 18

3.2.1 Valutazione preliminare . . . 18
3.2.2 Valutazione esibizione . . . 19
3.2.3 Caratteristiche interessanti . . . 21
3.3 Implementazione delle Gare Pokémon Live . . . 21
3.3.1 Funzionalità implementate . . . 21
3.3.2 Ordine di partenza . . . 22
3.3.3 Completa sequenzialità . . . 22
3.3.4 Struttura dell'implementazione . . . 23
3.3.5 Ulteriori scelte dell'implementazione . . . 23
3.3.6 Interfaccia grafica . . . 25

4 Analisi sulle prestazioni 29
4.1 Struttura delle analisi . . . 29
4.1.1 Scelta dei giocatori . . . 29
4.1.2 Organizzazione dei dati . . . 29
4.2 Simulazioni interne al programma . . . 31
4.2.1 Casuali . . . 31
4.2.2 Equilibrati con MCTS . . . 33
4.2.3 Sbilanciati con MCTS . . . 35
4.3 Simulazioni contro il 3DS . . . 37
4.3.1 Risultati . . . 38
4.3.2 Confronto con i risultati precedenti . . . 40
4.4 Prestazioni al variare delle iterazioni . . . 40
4.4.1 Analisi qualitativa . . . 41
4.4.2 Cosa si può migliorare . . . 44

A Lua 45
A.1 LÖVE . . . 46
A.2 Programma e codice sorgente . . . 46

Capitolo 1

Cenni sull'intelligenza artificiale applicata ai giochi

L'intelligenza artificiale (in inglese *artificial intelligence* o semplicemente AI) è la scienza e l'insieme delle tecniche che si occupano di creare macchine intelligenti, in particolare programmi intelligenti per computer.[1] Così John McCarthy [11] definisce questa complessa e interessante disciplina, di cui ha coniato il termine e di cui è stato uno dei fondatori.

A cavallo tra psicologia, matematica e informatica, ci sono stati molti dibattiti riguardo alla possibilità o meno di creare un'intelligenza artificiale *forte* o generale, che possa cioè svolgere qualsiasi compito intellettuale di cui un essere umano è capace. Tuttavia, nel nostro caso ci focalizzeremo su un tipo di intelligenza artificiale *debole* o applicata, ovvero creata per svolgere un unico compito, come può essere giocare a un gioco o rispondere a delle domande.

1.1 Teoria dei giochi

Un *gioco* può essere definito con un insieme di regole stabilite che permette l'interazione di uno o più giocatori per produrre un certo esito.

1.1.1 Giochi in forma estesa

Un *gioco in forma estesa* può essere descritto dalle seguenti componenti [3]:

- un diagramma ad albero con radice, detto albero di gioco;
- un insieme di giocatori $\mathcal{N} = \{1, \ldots, N\}$;
- una funzione giocatore che assegna a ogni nodo non terminale un'etichetta in $\{0, \ldots, N\}$ in cui

[1] "It is the science and engineering of making intelligent machines, especially intelligent computer programs." [11]

- i nodi etichettati $\{1, \ldots, N\}$ sono nodi decisionali dei giocatori;
- i nodi etichettati 0 sono *nodi natura*, ovvero nodi in cui è il caso ad agire (es. il lancio di un dado).

Detto X l'insieme di tutti i nodi e Z l'insieme dei nodi terminali indichiamo tale funzione come $P : X \setminus Z \to \{0\} \cup \mathcal{N}$;

- una distribuzione di probabilità sulle alternative del nodo natura;
- una partizione dei nodi decisionali del giocatore $i \in \mathcal{N}$ nei sottoinsiemi detti *stati di informazione*. Se ogni stato di informazione è atomico, ovvero contiene un solo nodo, si dice che il gioco è a *informazione perfetta*;
- una funzione che associa ad ogni alternativa in un nodo decisionale una etichetta detta *mossa* che dipende solo dallo stato di informazione;
- una funzione *payoff* o *ricompensa* definita sui nodi terminali $u_i : Z \to \mathbb{R}$.

Ad ogni nodo dell'albero di gioco associamo uno *stato* del gioco (per es. la disposizione dei pezzi sulla scacchiera negli scacchi). La *strategia* di ogni giocatore determina la probabilità di scegliere la mossa m in un dato stato s. Una combinazione di strategie dei giocatori forma un *equilibrio di Nash* se per nessun giocatore è vantaggioso cambiare unilateralmente strategia, ovvero cambiare strategia se si suppone che tutti gli altri non cambino la propria. Esiste sempre un equilibrio di Nash, ma spesso è molto complicato trovarlo.

Spesso nei giochi di cui ci occupiamo la ricompensa in un nodo terminale è $+1$, 0 o -1 in caso di vittoria, pareggio o sconfitta, rispettivamente. Questi sono i valori *teorici del gioco* di un nodo terminale.

1.1.2 Classificazione dei giochi

I giochi possono essere classificati in base alle seguenti proprietà:

A somma nulla Se le ricompense per tutti i giocatori hanno somma nulla (nel caso di 2 giocatori, se sono in competizione tra loro, ovvero se la vittoria di uno implica la sconfitta dell'altro).

Informazione perfetta o imperfetta Se lo stato del gioco è completamente o parzialmente osservabile per i giocatori.

Stocastico o deterministico Se interviene un fattore casuale o meno.

Sequenziale o simultaneo Se le mosse avvengono sequenzialmente o simultaneamente.

Discreto o continuo Se le mosse sono discrete o applicate in tempo reale.

I giochi a 2 giocatori che sono a somma nulla, a informazione perfetta, deterministico, discreti e sequenziali sono descritti come *giochi combinatori*. In questa definizione ricadono i giochi che analizzeremo più avanti, ovvero gli scacchi e il Go.

1.2 AI per i giochi

I giochi combinatori sono stati sin da subito un ottimo terreno di prova per gli algoritmi di AI, in quanto si svolgono in ambienti controllati definiti da semplici regole, ma che solitamente mostrano una profondità e complessità di gioco molto elevate e quindi creano sfide molto interessanti.

Uno dei modi più classici per affrontare lo sviluppo di una AI per questo tipo di giochi è l'algoritmo di *minimax*.

1.2.1 Cenni sul minimax

L'algoritmo di minimax [14, p. 163–171] è un algoritmo ricorsivo per scegliere la prossima mossa in un gioco a n giocatori (noi ci restringeremo per semplicità a un gioco a 2 giocatori). A ogni stato del gioco viene assegnato un valore tramite una *funzione di valutazione* che indica quanto sia desiderabile raggiungere tale stato. Il giocatore sceglie la mossa che minimizza il valore della migliore posizione raggiungibile dall'altro giocatore.

Riferendosi al giocatore A, un possibile metodo per implementare l'algoritmo è di assegnare $+1$ (o $+\infty$) a una vittoria per il giocatore A e -1 (o $-\infty$) a una vittoria del giocatore B. Per ogni altra mossa, il valore per il giocatore A è il minimo di tutti i valori risultanti dalle possibili risposte del giocatore B. Per questa ragione, A è chiamato *giocatore massimizzante* (cerca di massimizzare il suo guadagno) mentre B *giocatore minimizzante* (cerca di minimizzare il guadagno di A), da cui il nome *minimax*.

Spesso, come nel caso degli scacchi o del Go, lo spazio di gioco è troppo grande per essere esplorato tutto in un tempo ragionevole. Per tale motivo, invece di proseguire fino alla fine del gioco, si limita l'algoritmo a un numero prefissato di mosse. Quindi si applica una funzione di valutazione euristica allo stato finale raggiunto che restituisce quanto è buona la situazione, con una stima basata su diversi fattori.

Ci sono diversi metodi per ottimizzare questo algoritmo, uno dei più conosciuti e usati è la potatura α-β, di cui però non tratteremo.

1.2.2 Scacchi

Gli scacchi sono stati ritenuti sin da subito una delle prove più ardue per le AI, arrivando perfino a essere definito la drosofila delle AI [6]. Molto famoso e giocato, nonostante abbia delle regole relativamente semplici è molto complesso dal punto di vista strategico. È un gioco *convergente*, in quanto con il tempo il numero di pezzi in gioco diminuisce e, di conseguenza, anche il numero di mosse possibili.

Tuttavia, con l'evoluzione dei computer, IBM riuscì a costruire Deep Blue, un computer che sfruttava un algoritmo di minimax che guardava fino a 12 mosse in avanti e poi applicava una funzione di valutazione euristica, insieme ad ulteriori ottimizzazioni [8]. Nel 1996 Deep Blue riuscì a vincere un incontro contro il Campione del Mondo in carica, Garry Kasparov, perdendo però la partita 2-4. Nel 1997, dopo ulteriori aggiornamenti, riuscì a battere Kasparov in una intera partita sconfiggendolo 3.5-2.5.

1.2.3 Go

In realtà, uno dei più complicati problemi per l'AI è stato ed è tuttora il gioco del Go, originario della Cina.

Nel Go, due giocatori, il bianco e il nero, posano dei pezzi (chiamati pietre) sulle intersezioni vuote della griglia disegnata sulla scacchiera (chiamata *goban*), di dimensioni 19x19 (anche se a volte sono 13x13 o 9x9). Il gioco comincia con la griglia vuota e finisce quando essa è divisa in aree bianche e aree nere e nessuno dei due giocatori può più fare una mossa. Vince chi ha un'area maggiore. Ogni pietra ha un certo numero di libertà, pari al numero di intersezioni adiacenti (ortogonalmente) libere. Pietre poste una di fianco all'altra formano un gruppo, le cui libertà sono la somma delle libertà delle pietre che la compongono. Quando una pietra o un gruppo di pietre non ha più libertà viene rimossa dalla scacchiera. Non si possono muovere le pietre. Esistono regole aggiuntive, per evitare il suicidio e le ripetizioni infinite. In caso di disparità è possibile inserire un handicap, che consiste nel permettere a un giocatore di posare già un certo numero di pietre prima dell'inizio della partita.

Come si può notare, è un gioco con regole molto semplici, eppure è molto complesso. Inoltre è *divergente*, in quanto con il tempo il numero di pezzi sulla griglia aumenta.

La ragione principale dell'inefficienza di algoritmi come il minimax è che lo spazio degli stati di gioco è troppo grande (essendo un gioco divergente) e, soprattutto, è difficile creare una funzione di valutazione euristica efficiente. Il motivo è che non è facile stabilire la bontà della situazione guardando solo la posizione delle pietre sulla griglia: il Go ha una natura più visiva che è difficile implementare in un computer [13].

Tuttavia dei metodi alternativi, basati sugli algoritmi Monte Carlo e chiamati MCTS, si sono dimostrati molto efficienti. Infatti, dal 2006, anno di implementazione di tali algoritmi al Go, ci sono stati rapidi progressi: il programma *MoGo* riuscì a battere un giocatore umano professionista 5 dan in una griglia 19x19 con sole 7 pietre di handicap [7]. Tuttora le migliori AI per il go usano l'MCTS e la loro forza è quella di un buon giocatore amatoriale.

1.3 Metodi Monte Carlo

I metodi Monte Carlo sono un insieme di algoritmi che si basano sul campionamento casuale ripetuto per ottenere risultati numerici. In genere seguono uno schema particolare:

1. definire il dominio dei possibili input;

2. generare input in modo casuale da una distribuzione di probabilità (di solito uniforme) sul dominio;

3. eseguire dei calcoli deterministici sugli input;

4. raccogliere i risultati.

1.3.1 Calcolo di integrali

Un esempio emblematico riguarda l'uso dei metodi Monte Carlo per stimare il valore di π [9]. Si consideri un cerchio inscritto in un quadrato unitario. Notiamo che il rapporto tra l'area del quadrato e del cerchio è di $\pi/4$. Possiamo procedere nel modo seguente:

1. disegnare un quadrato e inscrivere un cerchio all'interno;
2. prendere dei punti con le coordinate scelte in modo casuale uniforme all'interno del quadrato;
3. contare il numero di punti che si trova all'interno del cerchio (X) e il numero di punti totali (N);
4. il rapporto tra i due numero è il rapporto tra le aree delle figure che, come osservato sopra, è di $\pi/4$: quindi $\pi \approx 4\frac{X}{N}$.

Questo sistema può essere esteso per l'utilizzo nel calcolo di integrali a molte dimensioni di figure molto complesse. Questi metodi, infatti, sono stati ideati proprio nell'ambito della fisica per calcolare integrali altrimenti irrisolvibili (o quasi).

1.3.2 Caratteristiche

Il punto di forza di questi algoritmi è la velocità, infatti richiedono al computer solo di calcolare numeri casuali e eseguire semplici operazioni matematiche tra numeri (come decidere se un punto si trovi o meno all'interno di un cerchio, nell'esempio sopra). Tuttavia, perché questi metodi siano efficienti, ci sono due requisiti fondamentali

1. i numeri casuali devono essere distribuiti in modo uniforme (o seguire qualsiasi altra distribuzione di probabilità necessaria), tant'è che proprio per questo motivo è stato stimolato lo sviluppo di generatori di numeri casuali (RNG) più efficienti;
2. il numero di campionamenti deve essere elevato, maggiore è il numero di campionamenti più preciso sarà il risultato.

1.4 Multi-armed bandit problem

In inglese questo problema è noto come *multi-armed bandit problem*, un gioco di parole basato sul fatto che in inglese le slot machines sono anche chiamate *one-armed bandit* perché operate originariamente da una leva su un lato della macchina e perché lasciavano le persone senza soldi. Infatti, nella formulazione di questo problema si immagina di essere davanti a una slot machine con molte leve da poter abbassare.

Un *K-armed bandit problem* [1] è definito dalle variabili casuali $X_{i,n}$ per $1 \leq i \leq K$ e $n \geq 1$, dove i è l'indice che identifica una leva della slot machine. Abbassando la leva i si ottengono le ricompense $X_{i,1}, X_{i,2}, \cdots$ che sono indipendenti e identicamente distribuite in accordo con una legge di probabilità sconosciuta con un'attesa incognita

μ_i. Anche le ricompense tra leve diverse mantengono l'indipendenza, ovvero $X_{i,s}$ e $X_{j,t}$ sono indipendenti (e in genere non identicamente distribuite) per ogni $1 \leq i < j \leq K$ e per ogni $s, t \geq 1$.

Una *strategia di allocazione* è un algoritmo che sceglie la prossima leva da abbassare basandosi sulla sequenza di giocate precedenti e sulle ricompense ottenute.

1.4.1 Rimorso

Sia $T_i(n)$ il numero di volte che la leva i viene giocata dalla strategia nelle prime n giocate. Allora il *rimorso* della strategia dopo n giocate è definito da:

$$\mu^* n - \sum_{j=1}^{K} \mu_j \mathbb{E}[T_j(n)] \quad \text{dove} \quad \mu^* := \max_{1 \leq i \leq K} \mu_i. \tag{1.1}$$

L'obiettivo è quello di minimizzare questo rimorso o, equivalentemente, di massimizzare la somma delle ricompense ottenute dopo n giocate. Infatti possiamo interpretare il rimorso come la differenza tra il massimo guadagno possibile (aver tirato n volte la leva migliore, per definizione quella che restituisce μ^* come ricompensa) e il guadagno effettivo.

Questo problema, molto più complesso di quanto possa sembrare, può essere affrontato utilizzando una strategia molto ingenua, che però non è molto efficace:

- inizialmente si gioca n volte ognuna delle leve;
- si gioca la leva che ha restituito, in media, la ricompensa più alta.

1.4.2 Upper Confidence Bounds (UCB1)

L' UCB1 (*Upper Confidence Bounds*) [1] è una strategia di allocazione molto brillante ed efficiente:

- inizialmente si gioca ogni leva una volta;
- si gioca la leva j che massimizza $\overline{X_j} + \sqrt{\frac{2 \log n}{n_j}}$ dove $\overline{X_j}$ è la ricompensa media ottenuta dalla leva j, n_j è il numero di volte che la leva j è stata giocata finora, e n è il numero totale di giocate.

Analizziamo la formula, che trova un equilibrio tra lo sfruttamento (*exploitation*) delle leve che sono più promettenti e l'esplorazione (*exploration*) delle leve giocate poco, potenzialmente vantaggiose:

- il termine $\overline{X_j}$ è semplicemente la ricompensa media della leva j (sfruttamento);
- il termine $\sqrt{\frac{2 \log n}{n_j}}$ dà un peso maggiore alle leve che sono state giocate poche volte: infatti, fissando n, al diminuire di n_j aumenta questo valore (esplorazione).

Capitolo 2

Monte Carlo Tree Search

Ora possiamo affrontare l'algoritmo denominato Monte Carlo Tree Search, o più semplicemente MCTS. L'idea di base è molto semplice e si basa su:

- teoria dei giochi, per definire l'albero di gioco;
- metodi di Monte Carlo, per effettuare simulazioni degli esiti di una partita.

Inoltre, come vedremo, l'algoritmo può essere notevolmente migliorato usando l'UCT, che è una modifica dell'UCB1. Questa ottimizzazione è molto importante: proprio l'introduzione dell'UCT permise all'MCTS di raggiungere un'altissima efficienza, ampliandone i suoi utilizzo. L'esempio più emblematico, in questo senso, sono le AI del gioco del Go. Come già detto, le migliori AI si basano sull'MCTS con UCT.

2.1 MCTS

Il Monte Carlo Tree Search (MCTS) è una tecnica di ricerca *best-first*[1] che usa simulazioni casuali [5]. L'MCTS può essere applicato a ogni gioco di lunghezza finita e si basa su simulazioni di partite in cui sia il giocatore che l'AI giocano mosse in modo casuale. Da una sola partita casuale (in cui tutti i giocatori giocano mosse casuali) si può dedurre poco, ma, simulando molte partite casuali, si può trarre una buona strategia.

2.1.1 L'algoritmo MCTS

L'algoritmo [2] crea e usa un albero di possibili stati futuri del gioco (come si vede in figura 2.1), usando il seguente procedimento:

Selezione Partendo dal nodo radice, viene usata una *strategia* per la selezione dei figli in modo ricorsivo per scendere nell'albero finché il nodo espandibile più importante è raggiunto. Un nodo è *espandibile* se rappresenta uno stato non terminale e ha dei figli non visitati.

[1] La ricerca *best-first* è un algoritmo di ricerca che esplora un grafo (es. un albero) espandendo i nodi più promettenti in base a una regola specifica.

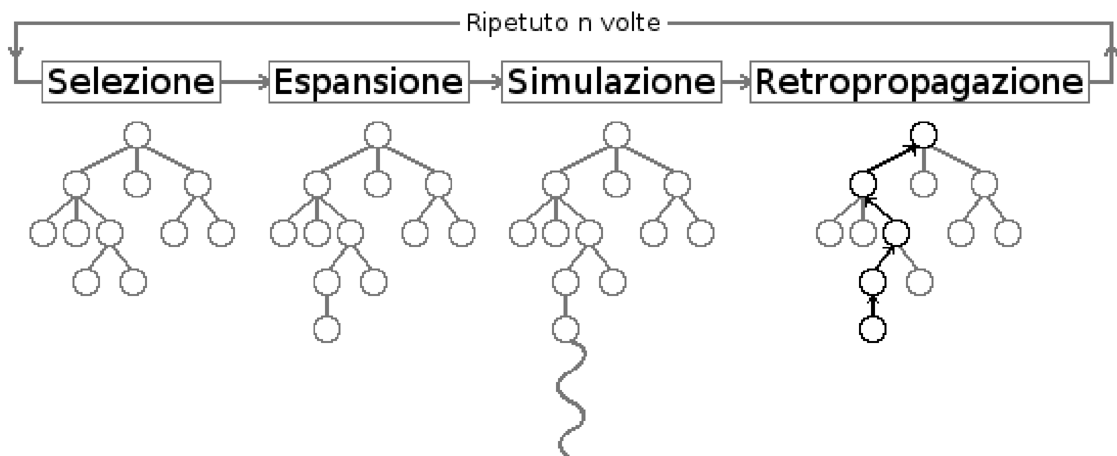


Figura 2.1: Rappresentazione grafica delle fasi dell'MCTS

Espansione Un nodo figlio è aggiunto all'albero per espanderlo, in accordo con le mosse disponibili.

Simulazione Si avvia una simulazione dal nuovo nodo in base a una *strategia di simulazione* finché non si raggiunge uno stato terminale e quindi un risultato (che spesso è vittoria, sconfitta o pareggio).

Retropropagazione Il risultato della simulazione viene usato per aggiornare ogni nodo dell'albero attraversato durante la partita. Vengono cambiate le sue statistiche in base al risultato ottenuto.

L'algoritmo sceglie, quindi, la mossa che corrisponde al nodo figlio (del nodo radice) migliore. La definizione di *migliore* dipende dall'implementazione.

2.1.2 Strategie delle fasi

In particolare, l'efficienza dell'algoritmo è fortemente influenzato dalle strategie usate nelle diverse fasi, ovvero:

Strategia per l'albero Seleziona o crea un nodo figlio tra i nodi già contenuti nell'albero (fasi di selezione e espansione). Più avanti tratteremo di una delle più efficienti strategie di questo tipo, ovvero l'UCT.

Strategia di simulazione Sceglie quali mosse eseguire nelle simulazioni casuali (fase di simulazione). La più semplice strategia di questo tipo consiste nel scegliere una mossa tra quelle disponibili in modo casuale uniforme. Può essere migliorata notevolmente aggiungendo conoscenza di dominio (*domain knowledge*), ovvero aiutandosi con strategie e trucchi già conosciuti specifici per il gioco, scartando in tal modo le mosse che si sanno già essere poco efficaci.

Solo la fase di retropropagazione non possiede una strategia specifica, perché aggiorna solo le statistiche dei nodi visitati per migliorare le prossime ricerche nell'albero. La statistica usata può variare di gioco in gioco, ma un modo molto immediato e di uso generico è di usare $Q(v)/N(v)$, dove $N(v)$ è la somma del numero delle visite e $Q(v)$ è la somma dei risultati delle partite (assegnando, per esempio, +1 a vittoria, +1/2 a pareggio e 0 a sconfitta). Questa statistica approssima il valore teorico del gioco (*game-theoretic value*)[2].

2.2 Upper Confidence Bounds for Trees (UCT)

Lo scopo dell'MCTS è approssimare il vero valore teorico delle mosse che possono essere eseguite dallo stato attuale. Questo viene effettuato, come visto, creando iterativamente un albero di gioco parziale. Il successo dell'MCTS, soprattutto per il Go, è dovuto in gran parte alla strategia di selezione. In particolare, Kocsis e Szepesvári [10] proposero di usare l'UCB1 come strategia per l'albero.

2.2.1 L'algoritmo UCT

Ogni volta che un nodo (raggiungibile tramite una mossa) viene scelto dall'albero esistente, la scelta può essere modellata come un *multi-armed bandit problem.* Viene scelto il nodo j che massimizza:

$$\text{UCT} = \overline{X_j} + 2C_p\sqrt{\frac{2\log n}{n_j}} \tag{2.1}$$

dove $\overline{X_j}$ è la ricompensa attesa approssimata dalle simulazioni Monte Carlo, n è il numero di volte che il nodo corrente (padre) è stato visitato, n_j è il numero di volte che il nodo figlio j è stato visitato e $C_p > 0$ è una costante. Se più nodi hanno lo stesso valore massimale, ne viene scelto uno in modo casuale uniforme. I valori di $X_{i,t}$ e quindi i valori di $\overline{X_j}$ si suppongono essere contenuti nell'intervallo $[0, 1]$ (ipotesi nelle dimostrazioni dell'UCB1 e dell'UCT). In generale, si pone che se $n_j = 0$ allora UCT vale $+\infty$, in modo che ai nodi figlio non ancora visitati venga assegnato il più grande valore possibile, per assicurarsi che tutti i figli di un nodo siano considerati almeno una volta prima che qualsiasi figlio venga espanso ulteriormente.

L'algoritmo termina dopo che è stato raggiunto un limite massimo, che può essere di tempo, di iterazioni o di memoria.

2.2.2 Sfruttamento contro esplorazione

Come già visto per l'UCB1, c'è un equilibrio tra il primo (sfruttamento) e il secondo (esplorazione) termine dell'equazione UCT. Quando un nodo viene visitato, il denomi-

[2] Per valore teorico del gioco di uno stato si intende il valore della ricompensa dell'albero di gioco per quel specifico stato. Si può ottenere, per esempio, creando l'albero di gioco completo tramite l'algoritmo di minimax.

natore del termine di esplorazione aumenta, quindi diminuisce il suo contributo complessivo. D'altra parte, se un altro nodo figlio del nodo padre è visitato, il numeratore aumenta e quindi il termine di esplorazione dei nodi fratello non visitati aumenta. Il termine di esplorazione assicura che ogni nodo figlio abbia una probabilità non nulla di essere selezionato, che è essenziale data la natura casuale delle simulazioni. Questo fornisce anche una proprietà intrinseca di *riavvio* dell'algoritmo, visto che anche i nodi figlio con bassa ricompensa hanno comunque la garanzia di essere scelti prima o poi, dopo un tempo sufficiente, e quindi sono esplorate diverse linee di gioco.

La costante C_p può essere modificata per diminuire o aumentare la quantità di esplorazione effettuata. Il valore $C_p = 1/\sqrt{2}$ è stato mostrato da Kocsis e Szepesvári [10] soddisfare la disuguaglianza di Hoeffding[3] con ricompense nell'intervallo $[0, 1]$. Con ricompense fuori da tale intervallo, potrebbero essere necessari un valore diverso per C_p e ulteriori miglioramenti.

Nell'applicare l'UCT, la scelta della mossa migliore tra i nodi figlio del nodo radice può avvenire in due modi diversi:

- si sceglie la mossa che porta al nodo con ricompensa più alta;
- si sceglie la mossa che porta al nodo più visitato, ricordandosi che, per come funziona l'UCT, il nodo più visitato è quello più promettente.

2.2.3 Convergenza al minimax

Il contributo chiave di Kocsis e Szepesvári [10] consiste nel mostrare che il limite sul rimorso dell'UCB1 vale anche per il caso di una distribuzione delle ricompense non stazionaria, e di dimostrare empiricamente i meccanismi dell'MCTS con l'UCT su molti domini. In particolare, mostrarono che la probabilità di fallimento alla radice dell'albero (ovvero la probabilità di scegliere una mossa sub-ottimale) converge a zero con un andamento polinomiale al tendere a infinito dei giochi simulati. Questa dimostrazione implica che, con abbastanza tempo (e memoria), l'UCT permette all'MCTS di convergere all'albero di minimax ed è, quindi, ottimale.

2.3 Caratteristiche

Ciò che rende l'MCTS una scelta popolare tra gli algoritmi di AI risiede in alcune sue caratteristiche uniche [2].

2.3.1 Non euristico

Uno dei più importanti benefici dell'MCTS è che non è necessaria conoscenza di dominio specifica[4], rendendolo facilmente applicabile a qualsiasi dominio che può essere modellato

[3] La disuguaglianza di Hoeffding fornisce un limite superiore sulla probabilità che la somma di variabili aleatorie devi dal suo valore atteso.

[4] Si ricorda che con *conoscenza di dominio* si intende l'insieme delle conoscenza legate a un certo ambito, in questo caso riguardo al gioco. Questo include strategie, aperture e chiusure, per esempio.

tramite un albero. Anche se l'algoritmo di minimax a massima profondità è ottimale nel senso teorico del gioco, la qualità di una partita per il minimax con profondità limitata dipende significativamente dalla funzione di valutazione euristica usata per valutare i nodi foglia. In giochi come gli scacchi, dove euristiche affidabili sono emerse dopo decenni di ricerca, il minimax funziona molto bene. In casi come il go, però, dove lo spazio degli stati di gioco ha un ordine di grandezza maggiore e sono più difficili da formula euristiche efficaci, le prestazioni del minimax diminuiscono significativamente.

Anche se l'MCTS può essere applicato in sua assenza, si possono ottenere miglioramenti importanti nelle prestazione usando conoscenza di dominio specifica. Tutti i programmi migliori di AI per go basati su MCTS usano attualmente informazioni specifiche sul gioco. Questa conoscenza non deve essere completa, a patto che sia in grado di influenzare la selezione delle mosse in modo favorevole.

Ci sono compromessi da considerare quando si usa la conoscenza di dominio: uno dei vantaggi della scelta delle mosse casuale uniforme è la velocità, permettendo di eseguire molte simulazioni in un dato tempo. La conoscenza di dominio specifica solitamente riduce drasticamente il numero di simulazioni possibili, ma può anche ridurre la varianza dei risultati delle simulazioni.

2.3.2 Sempre arrestabile

L'MCTS retropropaga il risultato di ogni partita immediatamente, facendo sì che tutti i valori siano sempre aggiornati a ogni iterazione dell'algoritmo. Questo gli permette di restituire un'azione dalla radice dell'albero in qualsiasi momento. Lasciare continuare l'algoritmo per ulteriori iterazioni spesso migliora il risultato.

Questo non è possibile tramite il minimax, il cui algoritmo deve concludersi per restituire il risultato.

2.3.3 Asimmetrico

La strategia di selezione dell'albero permette di favorire i nodi più promettenti (senza comunque lasciare che la probabilità di selezione degli altri nodi converga a zero), portando a un albero asimmetrico con il tempo. In altre parole, la creazione di un albero parziale è orientato verso le regioni più promettenti e quindi più importanti.

La forma dell'albero che emerge può essere usata per ottenere una migliore conoscenza sul gioco stesso. Per esempio, l'analisi della forma applicata all'albero generato tramite UCT può essere usata per distinguere partite giocabili e non giocabili.

Infatti, per implementare l'MCTS è necessario conoscere soltanto le regole del gioco e nient'altro (per cui l'assenza di conoscenza di dominio).

Capitolo 3

Applicazione dell'MCTS per le Gare Pokémon Live

I videogiochi rappresentano una sfida ancora più ardua per gli algoritmi di intelligenza artificiale. In particolare:

- sono più complessi da rappresentare e lo spazio degli stati è molto grande;
- hanno un'ampia variabilità e quindi può essere molto complesso trovare una funzione di valutazione euristica;
- molto spesso si basano su esiti casuali;
- possono essere in tempo reale (il che pone limiti a volte molto stretti riguardo al tempo di elaborazione permesso) e basati su decisioni simultanee.

Tutto questo può essere molto difficile da gestire per un algoritmo di AI. Tuttavia, l'MCTS permette di superare brillantemente questi problemi, tant'è che è stato utilizzato anche per videogiochi del calibro di *Total War: Rome II*[1] [4]. Il vantaggio principale risiede sempre nel fatto che, nella implementazione base, non necessita altro che la conoscenza e l'implementazione delle regole, quindi non richiede lavoro aggiuntivo rispetto al solo sviluppo del videogioco.

In questo capitolo si mostra l'implementazione di questo algoritmo e la sua applicazione al gioco scelto, le Gare Pokémon Live. Il linguaggio scelto è il Lua; per maggiori informazioni e per il codice sorgente è possibile consultare l'appendice.

[1] Videogioco di strategia a turni in cui si è chiamati a comandare una delle fazioni ai tempi dell'Impero Romano. È molto complesso, con azioni che vanno dal comandare l'esercito al gestire il malcontento nelle città.

3.1 Implementazione dell'MCTS

Perché l'algoritmo possa essere implementato, è stato necessario prima creare una classe che descriva l'albero di gioco, *node.lua*, e poi una classe che esegua le diverse fasi discusse nel capitolo precedente, *mcts.lua*.

3.1.1 Albero di gioco e nodi

Per implementare l'albero di gioco è sufficiente creare una classe `Node` che descriva un nodo dell'albero e collegare i nodi l'uno con l'altro (secondo la relazione genitore-figlio), presente nel file `node.lua`. Tra questi nodi ce ne sarà uno preferenziale che non avrà nodi genitori, ovvero la radice dell'albero di gioco, che rappresenterà l'intero albero, in quanto da esso sarà possibile risalire a qualsiasi altro nodo figlio discendendo l'albero.

Ogni nodo, ovvero ogni istanza della classe `Node`, rappresenterà uno stato del gioco e avrà come parametri:

`move` La mossa con la quale si arriva a questo nodo. È nulla solo nel caso si tratti del nodo radice.

`parent` Il nodo genitore, tramite il quale è possibile risalire l'albero fino alla radice. È nullo solo nel caso si tratti del nodo radice.

`children` I nodi figli, tramite i quali è possibile discendere fino a un nodo terminale.

`player` Il giocatore associato a questo nodo, ovvero il giocatore che ha eseguito la mossa con la quale si arriva a questo nodo.

`wins` Il numero di vittorie totali delle simulazioni che sono passate per questo nodo.

`visits` Il numero di volte che questo nodo è stato attraversato.

`untriedmoves` Quando viene inizializzato contiene l'insieme di tutte le possibili mosse che possono essere compiute dallo stato rappresentato da questo nodo. Con il tempo, vengono rimosse le mosse già provate: queste porteranno a un nuovo nodo che sarà aggiunto come nodo figlio di questo nodo.

Tra i metodi della classe `Node` è rilevante `Node:UCTSelectChild()`, che ordina i nodi figli in base al valore dato dalla formula UCT dal più grande al più piccolo e selezionando il più grande, ovvero quello più promettente come compromesso tra esplorazione e sfruttamento. Chiamiamo `self` il nodo corrente e `c` il nodo figlio analizzato, in Lua la formula UCT è resa da

```
function (c) return c.wins/c.visits +
↪  math.sqrt(2*math.log(self.visits)/c.visits) end
```

che ricalca l'equazione 2.1. Tale metodo è il fulcro dell'algoritmo UCT.

3.1.2 MCTS

L'algoritmo MCTS è implementato nel file `mcts.lua`. Consiste in un'unica funzione `MCTS(rootstate, itermax, rootplayer)` che riceve come input:

`rootstate` Stato di gioco iniziale, dal quale il giocatore deve fare la scelta della mossa.

`itermax` Numero di iterazioni da compiere, ovvero il numero di simulazioni totali da eseguire. Questo è il limite posto all'algoritmo e, variandolo, possiamo ottenere giocatori più forti, ma anche più lenti nell'eseguire mosse.

`rootplayer` Indice del giocatore che deve eseguire la mossa, ricordando che nelle Gare Pokémon Live i giocatori sono quattro.

Dopo aver inizializzato l'albero di gioco, associando il nodo radice allo stato di gioco iniziale fornito viene eseguito per `itermax` volte l'algoritmo MCTS. `node` rappresenta l'albero di gioco, mentre `state` rappresenta lo stato di gioco iniziale.

Nella fase di selezione, partendo dal nodo radice si sceglie, di volta in volta, un nodo figlio tramite la formula UCT. Il processo finisce quando si raggiunge un nodo che ha ancora delle mosse non esplorate o che è terminale.

```
while table.isempty(node.untriedmoves) and not
↪  table.isempty(node.children) do
        node = node:UCTSelectChild()
        state:doMove(node.move)
end
```

Nella fase di espansione, si sceglie in modo casuale uniforme la mossa da effettuare e si esegue tale mossa, aggiungendo il nuovo nodo all'albero.

```
if not table.isempty(node.untriedmoves) then
        local m = state:doRandomMove(node.untriedmoves)
        node = node:addChild(m, state)
end
```

Nella fase di simulazione, dallo stato in cui ci si trova si eseguono mosse in modo casuale uniforme fino alla fine del gioco.

```
while not state:isEnded() do
        local m = state:doRandomMove()
end
```

Nella fase di retropropagazione, si parte dal nodo aggiunto in fase di espansione e si risale fino al nodo radice. Per ogni nodo attraversato, si incrementa di 1 il numero di visite e si aggiorna il numero di vittorie in base all'esito della simulazione.

```
while node ~= nil do
        node:update(state:getResult(1))
        node = node.parent
end
```

Infine si sceglie, partendo dal nodo radice, la mossa che porta al nodo con maggior numero di visite e si restituisce tale mossa. Avendo applicato l'UCT, il nodo più promettente è anche quello che è stato visitato di più

3.2 Le Gare Pokémon Live

Le Gare Pokémon Live[2] (in inglese *Pokémon Contest Spectacular*) sono un tipo di competizione alternativo alle più famose lotte Pokémon. Si possono trovare solo in *Pokémon Zaffiro Alpha* e *Pokémon Rubino Omega* (2014), remake dei videogiochi di terza generazione *Pokémon Zaffiro* e *Pokémon Rubino* (2003).

In ogni Gara Pokémon Live quattro Coordinatori Pokémon si sfidano ognuno con un suo Pokémon. Ogni Gara è caratterizzata da una delle cinque virtù (Classe, Bellezza, Grazia, Acume, Grinta) e da uno dei cinque livelli (Normale, Super, Iper, Master) che denotano la difficoltà. Si svolgono due round:

Valutazione preliminare Ogni Pokémon fa sfoggio della propria virtù, ottenendo maggiore successo se il valore della sua virtù è elevato.

Valutazione esibizione I Pokémon eseguono delle mosse per impressionare il pubblico o per spaventare gli avversari.

3.2.1 Valutazione preliminare

Il pubblico assegna un punteggio a ognuno dei Pokémon in base al valore della sua virtù relativa alla Gara in corso. Infatti, ogni Pokémon possiede una statistica per ogni virtù, che può essere aumentata dando da mangiare al Pokémon delle *Pokémelle*. Il punteggio è proporzionale al valore della virtù relativa alla Gara: se, per esempio, si sta giocando in una Gara di Bellezza, il pubblico terrà conto della Bellezza dei partecipanti.

Questa valutazione può essere molto importante per gli esiti della Gara, ma non ha valore strategico. Infatti, è facile massimizzare la virtù relativa alla Gara in questione per avere il punteggio massimo. Pertanto, nella presente implementazione questa valutazione non è tenuta in considerazione.

[2]© 2015 Pokémon. © 1995–2015 Nintendo/Creatures Inc./GAME FREAK inc. Pokémon, i nomi dei personaggi Pokémon, e Nintendo 3DS sono marchi registrati Nintendo.

3.2.2 Valutazione esibizione

La valutazione esibizione, invece, è interessantissima a livello strategico, infatti la scelta delle mosse giuste al momento giusto è essenziale per vincere la Gara, quale che sia il risultato della valutazione preliminare.

La valutazione esibizione si compone di cinque turni, in ognuno dei quali i quattro Pokémon partecipanti devono eseguire una mossa, se ne hanno la possibilità. Le mosse vengono scelte contemporaneamente e, quando tutti e quattro i partecipanti hanno deciso quale usare, comincia un nuovo turno.

Vince il Pokémon che totalizza il maggior numero di cuori totale, a fine partita. Lo scopo di ogni Pokémon è ottenere più cuori possibili, impressionando il pubblico eseguendo delle mosse d'effetto, oppure impedire agli avversari di esibirsi o sottrarre loro cuori, spaventandoli.

L'ordine iniziale in cui i Pokémon si esibiscono dipende dai risultati della valutazione preliminare (maggiore è stato il punteggio, prima si esibirà). Dal secondo turno fino alla fine, l'ordine dipende dal successo dell'esibizione: alla fine di ogni turno, i Pokémon vengono ordinati in base al numero di cuori ottenuto per quel turno (migliore è stata l'esibizione, prima si esibirà) e, successivamente, i cuori sono azzerati per l'esibizione del turno successivo. In caso di parità, l'ordine viene scelto in modo casuale uniforme.

Ogni Pokémon partecipante può essere affetto da diversi stati.

Protetto Il Pokémon non può essere spaventato dagli avversari, quindi non può perdere cuori. L'effetto può durare solo una volta o per tutto un turno, in base al tipo di mossa che ha provocato tale stato.

Nervoso Il Pokémon è troppo nervoso per esibirsi, perciò per il turno corrente non può eseguire la sua mossa.

Impossibilitato Il Pokémon non può esibirsi, a causa di una mossa usata un turno precedente che richiedeva molte energie. Quindi, non può eseguire la mossa corrente. Questo effetto può durare solo il turno successivo all'uso della mossa stancante o fino alla fine della Gara, in base al tipo di mossa che ha provocato tale stato.

Se il Pokémon può esibirsi, utilizzerà la sua mossa. Ogni mossa possiede una virtù, un valore esibizione, un valore intralcio e un effetto.

Il pubblico valuterà l'esibizione del Pokémon e assegnerà un certo numero di cuori in base alla mossa eseguita. L'entusiasmo del pubblico è rappresentato dall'indicatore stelle, che può contenere da 0 a 5 stelle. L'entusiasmo viene modificato in base al rapporto tra la virtù della Gara e la virtù della mossa eseguita. Se la reazione del pubblico è di entusiasmo, vengono aggiunte stelle; se è di indifferenza, non cambia nulla; se è di scontento, vengono rimosse stelle. Di solito conviene eseguire mosse della stessa virtù della Gara, o al più provocare indifferenza.

	Entusiasmo	**Indifferenza**		**Scontento**	
Classe	Classe	Bellezza	Grinta	Acume	Grazia
Bellezza	Bellezza	Classe	Grazia	Acume	Grinta
Grazia	Grazia	Bellezza	Acume	Classe	Grinta
Acume	Acume	Grazia	Grinta	Bellezza	Classe
Grinta	Grinta	Acume	Classe	Bellezza	Grazia

Questi sono i passi principali che si seguono nell'esecuzione della mossa:

1. Si aggiunge il numero di cuori dato dal valore esibizione della mossa. Il numero di cuori può essere variabile, in base all'effetto della mossa. Per esempio, può dipendere dal numero di stelle.

2. Se previsto, si toglie il numero di cuori dato dal valore intralcio della mossa agli avversari. Il numero di cuori rimossi può essere variabile, in base all'effetto della mossa. Inoltre, i bersagli possono essere il Pokémon precedente o tutti i Pokémon precedenti, cosa che varia di mossa in mossa. Non possono essere colpiti i Pokémon protetti o quelli che, in un turno precedente, hanno eseguito una mossa che rende loro impossibile esibirsi fino alla fine della Gara.

3. Vengono eseguiti altri effetti, come l'aggiunta di protezione.

4. Si riceve il bonus per combinazione. Infatti, eseguire una specifica mossa dopo un'altra può dar luogo a una combinazione. Se così è, si ricevono 3 cuori aggiuntivi.

5. Se la mossa usata è la stessa del turno precedente, si perde un cuore e, inoltre, non viene applicato nessuno dei bonus successivi.

6. Se la virtù della mossa coincide con la virtù della Gara, si accende l'entusiasmo del pubblico: si aggiunge una stella e, di conseguenza, si riceve un cuore. In base al tipo di mossa, si possono ricevere anche due stelle (e quindi due cuori) oppure, se l'entusiasmo del pubblico è smorzato (a causa di una mossa usata da un avversario), non si riceve nulla.

7. Se invece la virtù della mossa genera scontento tra il pubblico, si rimuove una stella e quindi un cuore.

8. Se il pubblico è entusiasta e le stelle sono arrivate a 5, il Pokémon eseguirà un'Esibizione Live, ovvero un particolare spettacolo che lascia il pubblico a bocca aperta. Questo porterà un bonus di ben 5 cuori e azzererà il numero di stelle. È molto importante ottenere questo bonus, che può cambiare le sorti del gioco.

Sono presenti anche altre meccaniche, come mosse che rendono i Pokémon più sicuri di sé, che però non sono presenti nella presente implementazione (cfr. 3.3.1).

3.2.3 Caratteristiche interessanti

La scelta è ricaduta sulle Gare Pokémon Live, oltre che per preferenza personale in quanto amante della serie, perché possiedono delle caratteristiche molto interessanti.

Hanno delle meccaniche che ricordano le lotte Pokémon, queste ultime però sono molto più complesse e in esse intervengono molti più fattori, anche casuali. Le Gare Pokémon sono più semplici, il che le rende più facilmente spiegabili e implementabili.

Inoltre, le Gare Pokémon sono sì un videogioco, ma ritengono la struttura a turni tipica dei giochi da tavola. Nonostante questo, che semplifica un po' la creazione di un albero di gioco eliminando gli aggiustamenti necessari per un gioco in tempo reale, propone ulteriori sfide molto interessanti, che richiedono una modifica dell'algoritmo MCTS. In particolare, le Gare Pokémon sono un gioco a 4 giocatori con scelta delle mosse simultanea. In più, sono presenti fattori casuali, però in misura ridotta, il che lo rende un buon compromesso tra gioco deterministico e gioco stocastico, mostrando comunque le prestazioni dell'MCTS anche con l'intervento di fattori casuali.

3.3 Implementazione delle Gare Pokémon Live

Per l'implementazione delle Gare Pokémon Live sono state scelte alcune restrizioni, per facilitarne la programmazione e per questioni di tempo. Tuttavia, è possibile senza troppa difficoltà apportare delle aggiunte per renderle uguali in tutto e per tutto alla versione presente nei videogiochi.

Inoltre, non è stato molto facile capire l'effetto pratico delle diverse mosse, in quanto nel gioco è presente solo una breve descrizione che spesso non è molto esplicativa e non è presente molta documentazione a riguardo. Infatti, anche se le lotte Pokémon sono molto più complicate, sono presenti tantissime informazioni online sul loro funzionamento, a differenza delle Gare Pokémon Live. Per le lotte Pokémon si organizzano tornei e campionati, mentre le Gare Pokémon Live non hanno questo privilegio, essendo presenti solo in pochi giochi della serie e non avendo tutta la profondità strategica della loro sorella maggiore. Perciò è stato necessario effettuare molti test per capire l'effetto preciso di certe mosse.

3.3.1 Funzionalità implementate

Tutti i tipi di mosse implementati sono presenti nel documento reperibile al seguente indirizzo: `http://matteosilvestro.altervista.org/files/tesi/MosseGarePokemonLive.pdf`.

L'obiettivo che si voleva raggiungere con questa implementazione era di essere esattamente uguali al videogioco, per poter succesivamente eseguire simulazioni con l'originale per testare la forza dell'MCTS contro l'AI del gioco. Tuttavia, questo sarebbe stato molto complicato e lungo e si è deciso, pertanto, di lavorare su un sottoinsieme completo del gioco.

In particolare, il gioco è stato implementato in modo da rappresentare totalmente le Gare di Bellezza di livello Master. Per queste Gare i Pokémon scelti dall'AI del gioco sono

estratti da una rosa di 5 Pokémon: Tropica (un Gorebyss), Plumy (un Vileplume), Macy (un Machoke), Betta (un Wobbuffett) e Trod (un Electrode). Inoltre, è stato aggiunto un altro Pokémon per poter competere contro questi avversari, ovvero Speranza (un Jigglypuff). Quindi, sono stati implementati tutti e sei questi Pokémon e relative mosse, con relativi effetti. Nel 3DS, inoltre, è stata creata Speranza, in modo da poterla usare nel gioco e farle eseguire le mosse suggerite dall'MCTS.

Questo significa che, almeno per le Gare di Bellezza di livello Master, l'implementazione creata e il gioco funzionano esattamente allo stesso modo.

Ovviamente, l'implementazione è fatta per funzionare in qualsiasi tipo di Gara e, con qualche aggiunta, potrebbe essere esteso a qualsiasi gara di qualsiasi livello. La classe che gestisce le Gare Pokémon si chiama `Contest`.

3.3.2 Ordine di partenza

Si è già detto che non è stata implementata la Valutazione Preliminare, in quanto poco interessante. Tuttavia, questo pone il problema di scegliere l'ordine del primo turno.

La classe `Contest` è strutturata in modo da prendere in input i 4 partecipanti già ordinati secondo l'ordine del primo turno. Nel caso di simulazioni con il gioco originale, si lascia al gioco il compito di scegliere l'ordine in base al valore della virtù di ogni Pokémon e successivamente l'ordine verrà fornito nella presente implementazione.

3.3.3 Completa sequenzialità

Uno dei primi problemi nell'applicazione dell'MCTS alle Gare Pokémon Live è che è necessario creare un albero di gioco, ma, come accennato prima, non è banale in quanto, pur essendo a turni, la scelta delle mosse è simultanea (tutti e 4 i giocatori scelgono contemporaneamente e indipendentemente la mossa da eseguire).

Per risolvere questo problema, è stata creata una ulteriore classe, chiamata `ContestState`, che tratta le Gare come se fossero un gioco a turni completamente sequenziale. Per `Contest` è necessario fornire tutte e quattro le mosse scelte dai partecipanti per eseguire un turno, per esempio `Contest:doTurn{ Betta = 1, Speranza = 3, Tropica = 1, Trod = 4 }`. Invece, per `ContestState` basta dare una mossa per volta, per esempio `ContestState:doMove(2)`. Per prima cosa, assegna a ogni giocatore un indice da 1 a 4 che resterà immutato per tutto il gioco. Poi, ogni volta che gli viene fornita una mossa, essa viene assegnata al prossimo giocatore (secondo l'ordine prescelto) e la mette in coda. Quando la coda si riempie e raccoglie tutte e quattro le mosse, viene eseguito il turno.

Tramite questo semplice trucco, l'MCTS funziona senza modifiche, in quanto quello con cui lavora è un gioco a turni completamente sequenziale per cui è facile creare l'albero di gioco. È bene ricordare che l'albero ottenuto ha i nodi di quattro in quattro intercambiabili, in quanto l'ordine in cui i partecipanti scelgono le mosse non influisce sull'effettivo ordine in gioco (determinato dai cuori raccolti nel turno precedente).

3.3.4 Struttura dell'implementazione

L'implementazione delle Gare Pokémon Live ha richiesto la creazione di diverse classi che concorrono nella realizzazione delle esibizioni.

Nell'implementazione di base, tutti i nomi sono in inglese, per maggiore internazionalità. L'interfaccia, tuttavia, è in italiano, in quanto il gioco su cui si è basati per l'implementazione era impostato in italiano.

`movedb.lua` Contiene un database con tutti i tipi di mosse (ovvero i valori di esibizione e intralcio e l'effetto) e un database che associa al nome di una mossa la sua virtù e il suo tipo.

`move.lua` Classe che descrive le mosse, a cui è necessario fornire solo il nome per ottenere, tramite il database di `movedb.lua`, tutte le informazioni necessarie. Utilizzo: `m = Move:new("Protect")`.

`pokemon.lua` Classe che descrive i Pokémon partecipanti, fornendo il nome e le 4 mosse che possiede. Ognuno possiederà il numero di cuori del turno, il numero di cuori totale e lo stato. Utilizzo: `p = Pokemon:new("Speranza", { Move:new("Mimic"), Move:new("Round"), Move:new("Defense Curl"), Move:new("Hyper Voice") })`.

`contest.lua` Classe che descrive il funzionamento delle Gare Pokémon Live. Riceve in input `condition`, la virtù della Gara, `pokemons`, una tabella con i Pokémon partecipanti e `verbose`, che indica se salvare o meno i risultati (nella tabella `events`) perché poi possano essere visualizzati in seguito (in caso di simulazione, per cui ci interessa solo il risultato e non lo svolgimento). Utilizzo: `c = Contest:new(condition, pokemons, verbose)`.

`conteststate.lua` Classe intermedia che rende le Gare Pokémon Live un gioco completamente sequenziale, come descritto prima. Si utilizza allo stesso modo di `Contest`. Utilizzo: `c = ContestState:new(condition, pokemons, verbose)`.

`interface.lua` Permette di visualizzare graficamente (o anche testualmente) quello che è successo nella Gara Pokémon Live. Legge la tabella `events` e restituisce le informazioni sugli avvenimenti in lingua italiana, l'unica disponibile in questa implementazione. Se `c` è un `Contest` o `ContestState` che si vuole rappresentare testualmente: `c:printEvents()`.

3.3.5 Ulteriori scelte dell'implementazione

Dopo aver implementato le basi dell'algoritmo al gioco in questione, è stato necessario effettuare altri aggiustamenti per ottenere prestazioni migliori.

Tipo di risultato per la retropropagazione

All'inizio, come risultato per l'aggiornamento nella retropropagazione si ponenda 1 in caso di vittoria e 0 in caso di sconfitta. Tuttavia, dopo alcune simulazioni, si notava che questo sistema portava spesso a scegliere mosse poco adatte e perdere spesso occasioni che avrebbero, altrimenti, fruttato molti cuori (in particolare contro l'intelligenza artificiale del 3DS). Questa analisi si è basata, perlopiù, su esperienza personale. La ragione di questo comportamento è da ricercare nel fatto che, in effetti, indirizzare l'algoritmo unicamente verso la mossa che porta il maggior numero di vittorie non dà risultati molto precisi e stabili. Infatti, la vittoria o meno del giocatore non dipende unicamente da come ha giocato, ma anche (e soprattutto) dalle prestazioni degli altri giocatori. Quindi, paradossalmente, un giocatore potrebbe avere giocato molto bene (per esempio, aver guadagnato 30 cuori, che sono molti per il gioco) ma aver comunque perso in quanto un altro giocatore potrebbe aver fatto meglio (per esempio, 31 cuori). Viceversa, potrebbe vincere anche avendo giocato molto male.

Quindi, l'idea successiva è stata quella di usare come numero retropropagato il numero di cuori ottenuto a fine gara, normalizzato su base 80 (il massimo numero di cuori ottenibile, anche se, in pratica, è già molto improbabile raggiungere solo i 40 cuori) in quanto, per l'implementazione corrente dell'MCTS, è necessario che il numero retropropagato appartenga all'intervallo $[0, 1]$. Questo sistema ha migliorato notevolmente la qualità delle mosse scelte, ma erano ancora presenti molte occasioni sprecate. Infatti, in questo modo, l'algoritmo non teneva in conto che, solitamente, i giocatori tendono ad eseguire le migliori mosse a disposizione e, soprattutto, a non eseguire mosse suicide. Per esempio, una mossa come Esplosione, che permette sì di guadagnare ben 8 cuori ma poi non permette più di eseguire altre mosse fino alla fine, non verrà mai usata nei primi turni. Eppure, se Speranza seguiva un Pokémon con una mossa del genere, sceglieva quasi sempre Mimica (mossa che copia il numero di cuori guadagnati dal Pokémon precedente e ne aggiunge uno) persino il primo turno, invece di altre mosse più adatte.

Per questo, si è giunti all'idea definitiva che, come si vedrà nel capitolo successivo, è particolarmente forte contro l'intelligenza artificiale del 3DS. Il numero retropropagato è dato dalla somma del numero di cuori ottenuto a fine gara (dal giocatore corrente) e del numero di cuori ottenuto a fine gara dal giocatore peggiore. Questo, infatti, fa in modo che l'algoritmo non tenga solo conto del maggior numero di cuori ottenibile qualsiasi scelta, per quanto stupida sia, degli avversari, ma anche del fatto che gli altri giocatori sceglieranno mosse per massimizzare il loro guadagno.

Ordine iniziale

L'ordine iniziale dei giocatori è stato basato su quello del 3DS. Infatti, come detto precedentemente, l'algoritmo non si occupa della valutazione preliminare. Tuttavia, nel gioco è essa a determinare l'ordine e ognuno dei possibili avversari di Speranza possiede uno specifico valore della statistica Bellezza (virtù scelta per le simulazioni). Speranza stessa è stata fornita di un certo valore di questa statistica. Questo fa sì che l'ordine iniziale sia sempre basato sulla seguente sequenza: Betta, Speranza, Tropica,

Trod, Macy, Plumy. Speranza è presente in tutte le simulazioni, mentre i tre avversari sono scelti, in modo casuale uniforme, tra i cinque possibili. Le simulazioni all'interno del programma seguono lo stesso ordine che seguirebbe una Gara sul 3DS, per avere omogeneità tra esse e le simulazioni contro il 3DS. Si ricorda, infatti, che l'ordine può influire molto sugli esiti della Gara.

Eventi casuali

Perché l'implementazione dell'MCTS sia del tutto efficiente, è necessario, se succede nel gioco, gestire i punti in cui degli esiti casuali modificano il gioco. Nella corrente implementazione, può succedere se Macy usa Attrazione (mossa che può far innervosire tutti i Pokémon successivi con una probabilità del 20%) o se, alla fine di un turno, due o più Pokémon hanno guadagnato lo stesso numero di cuori (se così succede, la parità dei cuori viene risolta scegliendo l'ordine in modo causale uniforme). Tuttavia, per motivi di tempo e velocità di esecuzione, la gestione degli eventi causali non è stata implementata. Essi, però, non modificano mai radicalmente il gioco: per quanto l'ordine di gioco o il fatto di perdere o meno un turno di esibizione può cambiare di molto il numero di cuori guadagnati dai giocatori, il numero di turni totali da giocare rimane lo stesso e, di norma, le differenze di cuori non sono molto elevate. Questa tesi è stata avvalorata da alcune simulazioni all'interno del gioco tra l'algoritmo corrente e un algoritmo che risolveva i casi di parità di cuori a fine turno, le cui prestazioni non erano significativamente diverse.

Livello dei giocatori

Il livello dei giocatori (ovvero il numero di iterazioni dell'algoritmo MCTS) è assegnato a 100 per gli avversari e 10000 per il giocatore umano, ovvero Speranza. Gli avversari si muovono secondo la mossa scelta dall'algoritmo, mentre invece, per Speranza, la scelta dell'algoritmo fornirà un suggerimento per il giocatore umano, evidenziando la mossa corrispondente. Questo significa che, seguendo sempre i consigli, si vincerà la maggior parte delle volte.

3.3.6 Interfaccia grafica

L'interfaccia grafica riprende quella originale dei videogiochi per 3DS *Pokémon Zaffiro Alpha* e *Pokémon Rubino Omega*. È stata realizzata usando LÖVE, un framework per creare giochi in 2D utilizzando Lua. Il programma finale è reperibile tramite i collegamenti presenti in appendice.

Schermata di selezione delle mosse

In alto, è visualizzato il numero del turno che sta per iniziare e una percentuale che indica il progresso dell'algoritmo. Si deve ricordare che il programma deve calcolare, tramite l'MCTS, la mossa migliore per ognuno dei quattro partecipanti: nel caso degli avversari, per farli giocare; nel caso del Pokémon del giocatore umano (Speranza), la mossa da suggerire al giocatore.

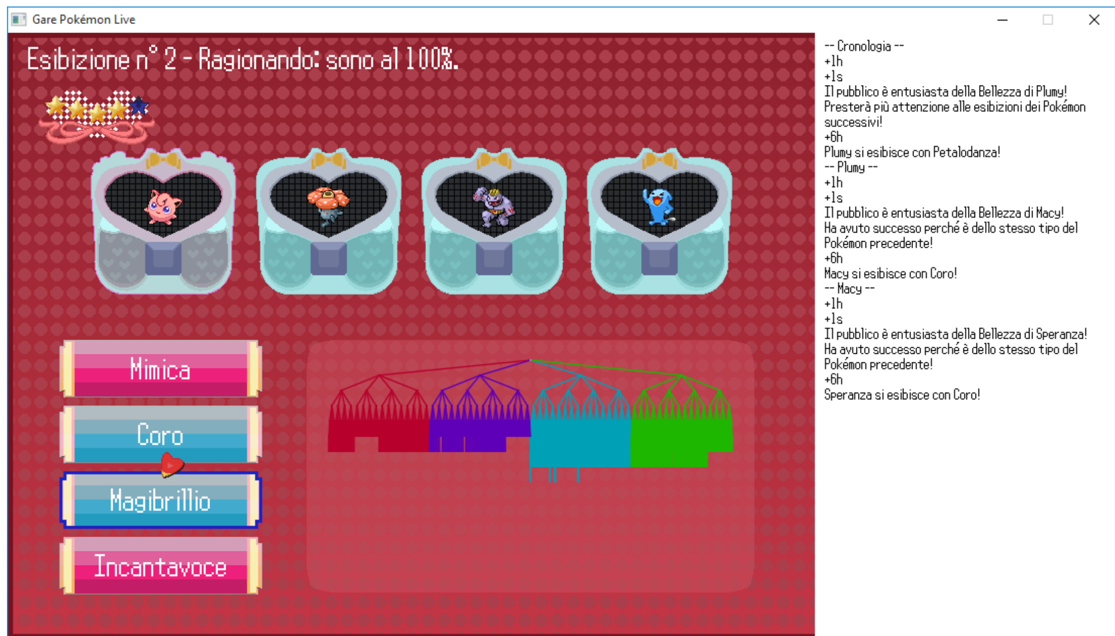


Figura 3.1: Schermata di selezione delle mosse

Sotto è presente l'indicatore stelle, che può contenere da 0 a 5 stelle. Ancora più giù, i quattro giocatori, ordinati secondo l'ordine di esibizione: ognuno possiede uno sprite, posti per i cuori (16, il massimo ottenibile in un singolo turno) e un quadrato grigio il cui posto verrà occupato dallo stato (nervoso, protetto o spaventato).

In basso a sinistra sono presenti le quattro mosse che possono essere scelte. Premendo il tasto Invio si esegue la mossa selezionata dalla freccia, che si muove con le frecce direzionali. A destra, invece, è presente una rappresentazione dell'albero di gioco creato dall'algoritmo. La pressione del tasto Tab, invece, mostrerà una schermata che descrive gli effetti della mossa selezionata. Dopo che l'algoritmo avrà finito di lavorare, verrà evidenziata in blu la mossa consigliata.

Sulla destra è mostrata la cronologia, un riepilogo di tutto quello che è successo nel turno precedente (a meno di essere al primo turno).

Schermata di svolgimento del turno

Una volta scelta la mossa e lasciato finire l'algoritmo, si svolge il turno. La schermata è pressoché identica, tuttavia sotto i partecipanti è semplicemente presente una vasta area che mostra del testo, che descrive quello che sta accadendo (che verrà aggiunto simultaneamente nella cronologia).

Schermata dei risultati

Finiti tutti e cinque i turni, vengono mostrati i risultati. Sono presenti quattro barre, ordinate secondo i cuori totali guadagnati. Le tacche rosse mostrano la percentuale di cuori rispetto al numero di cuori guadagnati dal vincitore, come nel gioco originale.

Capitolo 4

Analisi sulle prestazioni

Una volta implementato il programma si sono svolte delle analisi per verificare le prestazioni dell'algoritmo, sia all'interno del programma che contro il 3DS, l'unica intelligenza artificiale disponibile come metro di confronto.

4.1 Struttura delle analisi

Per procedere alle analisi è stato necessario seguire alcuni accorgimenti.

4.1.1 Scelta dei giocatori

In tutte le simulazioni, i quattro giocatori sono stati scelti nel seguente modo:

- un giocatore è rappresentato da Speranza;
- gli altre tre giocatori sono scelti, in modo casuale uniforme, tra i cinque possibili avversari, ovvero Macy, Trod, Tropica, Betta e Plumy.

Si è scelto di procedere in questo modo per rispecchiare le simulazioni fatte con il 3DS e avere omogeneità tra i dati: infatti, quando si gioca con il 3DS si sceglie un solo Pokémon da poter usare (in questo caso, sempre Speranza) e gli altri tre giocatori tra una rosa di giocatori preimpostati, nel caso delle Gare di Bellezza di livello Master composta dai cinque componenti citati prima.

4.1.2 Organizzazione dei dati

In tutti i casi, si ha a disposizione un dataset avente nelle colonne tutti e sei i giocatori e nelle righe il numero di cuori guadagnati nella partita. I giocatori non presenti in una partita hanno dei valori mancanti. Tuttavia, è stato ritenuto opportuno apportare una trasformazione a questi dati.

Speranza	**Tropica**	**Plumy**	**Macy**	**Betta**	**Trod**
13		23	16		19
33			19	26	18
34			23	25	21
32	19		16		20
24			24	19	21

Infatti, il numero di cuori guadagnato in una partita è sì un dato essenziale, ma dipende in gran parte dall'andamento della partita. Infatti, in una partita possono essere state usate spesso mosse che spaventano gli avversari, sottraendo loro cuori, che a loro volta non fanno guadagnare molto l'utilizzatore. Quindi, è più indicativo prendere in considerazione lo *scarto di cuori* rispetto al valor medio della partita, che indica, infatti, quanto il giocatore è stato migliore rispetto alla media della partita. Un valore alto indicherà che i cuori guadagnati hanno superato di molto la media della partita e, quindi, esso ha giocato molto bene.

Per ottenere questo risultato, per ogni riga a ogni valore è stata sottratta la media della riga.

Speranza	**Tropica**	**Plumy**	**Macy**	**Betta**	**Trod**
-4.75		5.25	-1.75		1.25
9			-5	2	-6
8.25			-2.75	-0.75	-4.75
10.25	-2.75		-5.75		-1.75
2			2	-3	-1

È presente inoltre un dataset strutturato come il precedente ma che possiede per ogni giocatore 1 in caso di vittoria e 0 in caso di sconfitta.

Speranza	**Tropica**	**Plumy**	**Macy**	**Betta**	**Trod**
0		1	0		0
1			0	0	0
1			0	0	0
1	0		0		0
1			0	0	0

Infine, è stato creato un unico grande dataframe con tre colonne:

pokemon Contiene il nome del Pokémon che ha ottenuto questi risultati.

hearts Lo scarto di cuori guadagnato rispetto alla media della partita.

wins Dipende dal risultato della partita, vale 1 in caso di vittoria e 0 in caso di sconfitta.

pokemon	hearts	wins
Speranza	-4.75	0
Speranza	9	1
Speranza	8.25	1
Speranza	10.25	1
Speranza	2	1
Tropica	-2.75	0
Plumy	5.25	1
Macy	-1.75	0
Macy	-5	0
Macy	-2.75	0
Macy	-5.75	0
Macy	2	0
Betta	2	0
Betta	-0.75	0
Betta	-3	0
Trod	1.25	0
Trod	-6	0
Trod	-4.75	0
Trod	-1.75	0
Trod	-1.00	0

Per le percentuali di vittoria, essendo che ogni giocatore gioca un numero diverso di partite, sono state calcolate svolgendo il rapporto tra il numero di vittorie sul totale di partite giocate.

Le analisi sono state eseguite con il software R, versione 3.2.2.

4.2 Simulazioni interne al programma

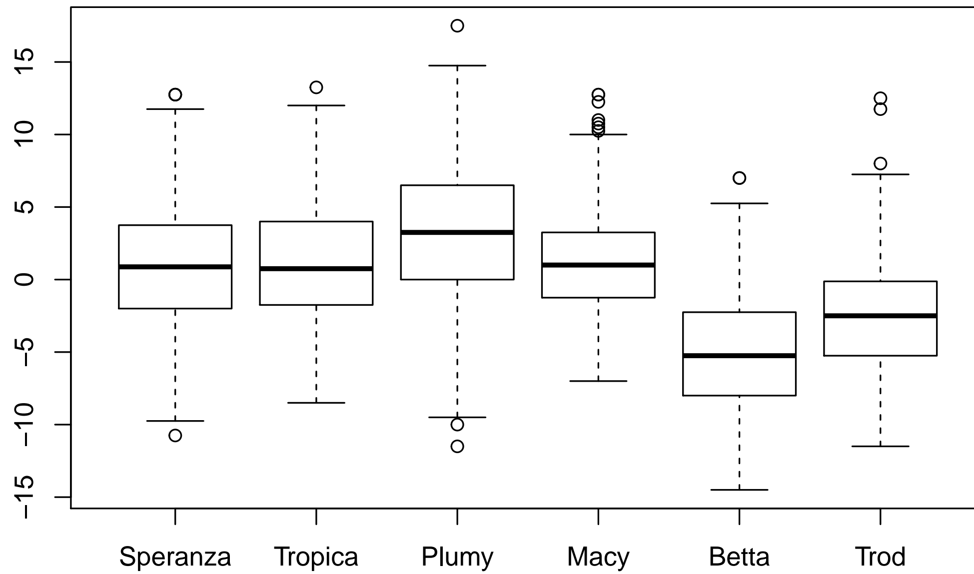
Per prima cosa, sono state testate le prestazioni dell'algoritmo contro se stesso, studiando diverse casistiche. Infatti, ogni giocatore ha le proprie particolarità: a causa delle diverse mosse che possiedono possono essere più o meno forti, e questo è un fattore molto importante da tenere in considerazione.

Tutti i campioni analizzati hanno taglia 500.

4.2.1 Casuali

In queste simulazioni tutti i giocatori scelgono le mosse in modo casuale uniforme, tra le quattro che possiedono. In questo modo si può saggiare la bontà delle mosse dei diversi giocatori.

Dopo aver appurato con un test di Shapiro che non tutti i campioni seguono una distribuzione normale, si è scelto di usare un test non parametrico, ovvero il test di Kruskal-Wallis, al posto dell'ANOVA a una via.




```
> kruskal.test(hearts ~ pokemon)

        Kruskal-Wallis rank sum test

data:  hearts by pokemon
Kruskal-Wallis chi-squared = 521.8, df = 5, p-value < 2.2e-16

> kruskalmc(hearts ~ pokemon)

Multiple comparison test after Kruskal-Wallis
p.value: 0.05
Comparisons
                    obs.dif critical.dif difference
Speranza-Tropica  30.528315     124.0492      FALSE
Speranza-Plumy   254.061399     123.6612       TRUE
Speranza-Macy     35.505841     122.0528      FALSE
Speranza-Betta   655.152876     122.9046       TRUE
Speranza-Trod    404.374357     126.5228       TRUE
Tropica-Plumy    223.533083     138.5184       TRUE
Tropica-Macy       4.977525     137.0845      FALSE
Tropica-Betta    685.681192     137.8434       TRUE
```

```
Tropica-Trod     434.902673     141.0790       TRUE
Plumy-Macy       218.555558     136.7335       TRUE
Plumy-Betta      909.214275     137.4944       TRUE
Plumy-Trod       658.435756     140.7380       TRUE
Macy-Betta       690.658717     136.0496       TRUE
Macy-Trod        439.880198     139.3269       TRUE
Betta-Trod       250.778519     140.0736       TRUE
```

Il test di Kruskal-Wallis permette di rifiutare l'ipotesi nulla di uguaglianza delle mediane.

Come si evidenzia dal test post-hoc realizzato, solo il gruppo Speranza-Tropica-Macy non ha differenze significative nella mediana, cosa già intuibile dal boxplot. In particolare, si può notare che Plumy sembra dotata già di un certo vantaggio, essendo significativamente diversa da tutti gli altri Pokémon e avendo una media migliore.

Il fatto che Betta e Trod abbiano una media più bassa non stupisce. Infatti, l'algoritmo tende a sfruttare al massimo mosse che generano tanti cuori, e in particolare Plumy, Speranza, Macy e Tropica ne sono dotate. Invece, Trod ha il problema che, tra le mosse a disposizione, ha Esplosione, che dà 8 cuori ma poi non permette più di esibirsi fino alla fine della gara, molto svantaggiosa se usata a inizio partita. Giocando casualmente, è possibile che accada questo. Inoltre, non possiede altre mosse che generano molti cuori: spesso non può guadagnarne più di 3. Betta è in una situazione ancora peggiore: infatti, oltre ad avere anch'essa una mossa come Esplosione, ovvero Destinobbligato, quest'ultima (insieme a Contrattacco) fa calare l'entusiasmo del pubblico, diminuendo di una stella l'indicatore stelle e sottraendo un cuore a Betta. Per di più, il successo di tutte le mosse di Betta sono strettamente legate alla tempistica: Salvaguardia è meglio usarla se si è primi nel turno, perché dà solo 2 cuori ma protegge per il resto del turno; Contrattacco e Specchiovelo sono più adatte se si è ultimi nel turno, perché ne danno 6 invece dei soliti 2. Quindi, scegliendo le mosse a caso è molto difficile fare le scelte giuste al momento giusto e, quindi, questi risultati erano prevedibili.

Le percentuali di vittoria sono le seguenti:

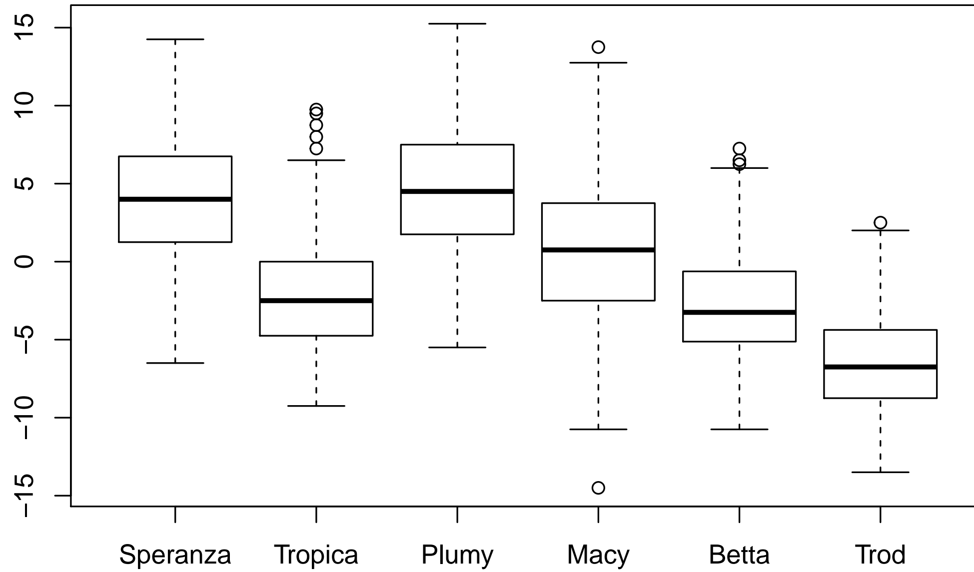
Speranza	**Tropica**	**Plumy**	**Macy**	**Betta**	**Trod**
24.40%	32.21%	49.17%	25.80%	7.82%	10.36%

4.2.2 Equilibrati con MCTS

In queste simulazioni, tutti i giocatori giocano utilizzando l'algoritmo MCTS con numero di iterazioni uguale, fissato a 100. In questo caso, non ci si aspetta che tutti i giocatori abbiano la stessa forza, in quanto la presenza di mosse diverse cambia drasticamente la bontà di un giocatore.

I test di Shapiro mostrano, anche in questo caso, che non tutti i campioni seguono una distribuzione normale. Quindi, si procede di nuovo con i medesimi test non parametrici.

```
> kruskal.test(hearts ~ pokemon)
```




```
        Kruskal-Wallis rank sum test

data:  hearts by pokemon
Kruskal-Wallis chi-squared = 1069, df = 5, p-value < 2.2e-16

> kruskalmc(hearts ~ pokemon)

Multiple comparison test after Kruskal-Wallis
p.value: 0.05
Comparisons
                    obs.dif critical.dif difference
Speranza-Tropica  672.05813     124.1799       TRUE
Speranza-Plumy     51.55971     124.1799      FALSE
Speranza-Macy     348.66209     128.9571       TRUE
Speranza-Betta    735.44921     121.0039       TRUE
Speranza-Trod    1106.17894     121.4639       TRUE
Tropica-Plumy     723.61785     139.0985       TRUE
Tropica-Macy      323.39604     143.3794       TRUE
Tropica-Betta      63.39107     136.2705      FALSE
Tropica-Trod      434.12081     136.6792       TRUE
Plumy-Macy        400.22180     143.3794       TRUE
```

```
Plumy-Betta       787.00892     136.2705       TRUE
Plumy-Trod       1157.73865     136.6792       TRUE
Macy-Betta        386.78712     140.6376       TRUE
Macy-Trod         757.51685     141.0336       TRUE
Betta-Trod        370.72973     133.8001       TRUE
```

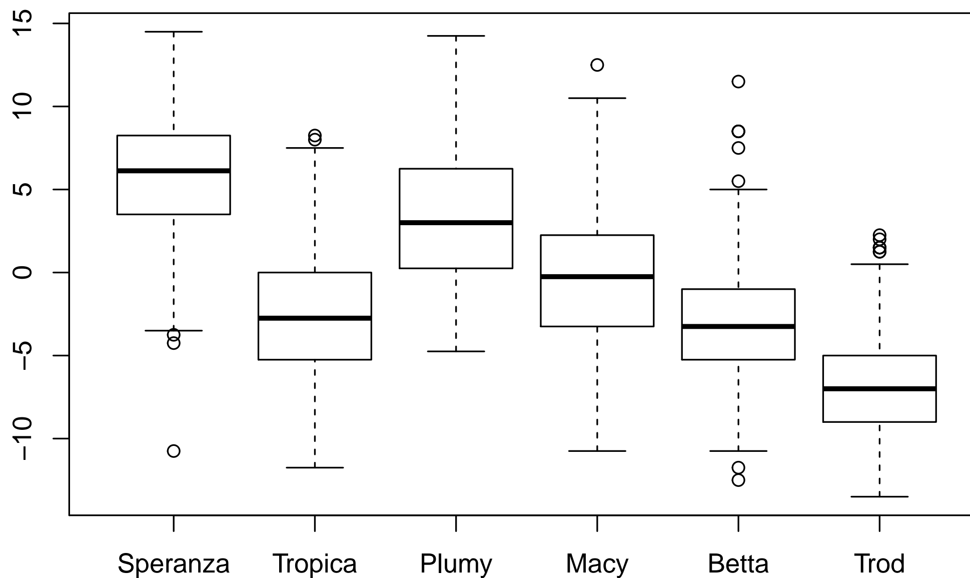
Come si nota dall'ultimo test, le uniche coppie che non presentano differenze significative sono Speranza-Plumy e Tropica-Betta. In particolare, rispetto al caso precedente, si nota come Speranza abbia ottenuto un notevole miglioramento, raggiungendo Plumy, mentre Betta ha avuto un buon miglioramento: grazie all'MCTS riesce a utilizzare meglio le sue mosse.

Le percentuali di vittoria sono le seguenti:

Speranza	**Tropica**	**Plumy**	**Macy**	**Betta**	**Trod**
44.60%	7.74%	52.19%	27.27%	7.74%	0.63%

4.2.3 Sbilanciati con MCTS

In queste simulazioni, Speranza gioca usando l'MCTS con 10000 iterazioni, mentre tutti gli altri con sole 100 iterazioni.



Di nuovo i test di Shapiro mostrano che non tutti i campioni sono normali, quindi si procede nuovamente con i test non parametrici.

```
> kruskal.test(hearts ~ pokemon)

        Kruskal-Wallis rank sum test

data:  hearts by pokemon
Kruskal-Wallis chi-squared = 1195.8, df = 5, p-value < 2.2e-16

> kruskalmc(hearts ~ pokemon)

Multiple comparison test after Kruskal-Wallis
p.value: 0.05
Comparisons
                    obs.dif critical.dif difference
Speranza-Tropica  846.25180     123.0290       TRUE
Speranza-Plumy    234.85428     125.8094       TRUE
Speranza-Macy     584.75710     124.0492       TRUE
Speranza-Betta    882.35237     123.9191       TRUE
Speranza-Trod    1267.68110     122.2929       TRUE
Tropica-Plumy     611.39752     139.5392       TRUE
Tropica-Macy      261.49470     137.9543       TRUE
Tropica-Betta      36.10056     137.8374      FALSE
Tropica-Trod      421.42930     136.3772       TRUE
Plumy-Macy        349.90282     140.4395       TRUE
Plumy-Betta       647.49809     140.3246       TRUE
Plumy-Trod       1032.82682     138.8906       TRUE
Macy-Betta        297.59527     138.7487       TRUE
Macy-Trod         682.92400     137.2983       TRUE
Betta-Trod        385.32873     137.1808       TRUE
```

Notiamo che l'unica coppia che non presenta differenze significative è la coppia Tropica-Betta. In particolare notiamo che Speranza, il cui algoritmo esegue ben 10000 iterazioni, riesce a distaccarsi da Plumy e superarla.

Questo è un buon risultato: significa che l'algoritmo riesce a sfruttare bene le mosse a disposizione. Infatti, Speranza possiede le seguenti quattro mosse:

Mimica Mossa di Grazia. Copia il numero di cuori del Pokémon che si è esibito precedentemente e ne aggiunge uno. Questa mossa è il fulcro della strategia di Speranza: infatti, usata al momento giusto, può copiare i cuori di una esibizione che ha fatto scatenare l'entusiasmo del pubblico (dando il bonus di 5 cuori) o di una mossa a grande effetto, come le prima citate Esplosione e Destinobbligato che danno 8 cuori di base.

Coro Mossa di Bellezza. Dà 2 cuori, ma se il Pokémon precedente ha eseguito una mossa della stessa virtù (in questo caso, Bellezza) ne dà ben 6. Anche questa mossa è importante che sia eseguita al momento giusto, prevedendo che il Pokémon precedente usi una mossa dello stesso tipo.

Magibrillio Mossa di Bellezza. Dà sempre 4 cuori. È l'unica mossa di Speranza che può essere usata in ogni momento garantendo un buon risultato, ma non deve essere usata a sproposito, in quanto, essendo di Bellezza, aumenta il numero di stelle e può quindi avvantaggiare un Pokémon che si esibisce successivamente.

Incantavoce Mossa di Grazia. Dà 2 cuori, ma se eseguita quando si è primi nel turno ne dà ben 6. Da usare solo se si è primi, altrimenti quasi ogni alternativa è più adatta.

Come si nota, quasi tutte le mosse richiedono un ben preciso tempismo e si può dire che l'algoritmo riesce a sfruttare al meglio questi vantaggi, anche se non si può dire la stessa cosa degli avversari. Questo è da ricercare nel fatto che l'algoritmo è stato pensato, in particolare, per avvantaggiare Pokémon che massimizzano il numero di cuori, come Speranza. Infatti, i Pokémon che ne traggono maggiore vantaggio sono proprio Speranza e Plumy, che hanno mosse che possono far guadagnare molti cuori, se usate nel momento giusto. Altri, come Tropica, che tendono più a ostacolare gli altri, non sono premiati. Ancor meno Trod: infatti, avendo mosse che danno non più di 3 cuori, eccetto Esplosione che però poi impedisce di fare altre esibizioni, non riesce ad essere sfruttato bene dall'algoritmo.

Le percentuali di vittoria sono le seguenti:

Speranza	**Tropica**	**Plumy**	**Macy**	**Betta**	**Trod**
67.00%	4.25%	29.12%	17.79%	4.68%	0.64%

Si nota anche nella percentuale di vittorie che Speranza ha ottenuto un notevole miglioramento, rubando una buona fetta di vittorie a Plumy. È anche molto emblematico Trod: la sua percentuale di vittorie si riduce drasticamente rispetto alle simulazioni casuali. La spiegazione è in linea con quanto detto prima: infatti, non avendo tante mosse che gli portano molti cuori, non riuscirà mai a guadagnarne molti, pur giocando benissimo, e gli altri giocatori più avvantaggiati riusciranno facilmente a superarlo e, quindi, a sconfiggerlo.

4.3 Simulazioni contro il 3DS

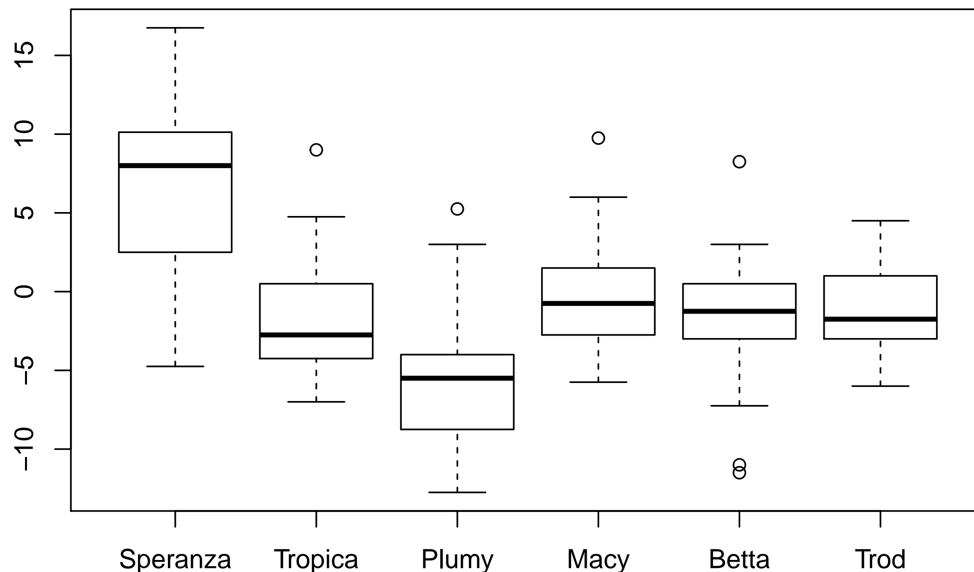
Il vero terreno di prova dell'algoritmo è però dato dalle simulazioni contro l'intelligenza artificiale del gioco originale, a cui ci si riferirà come "simulazioni contro il 3DS". La piattaforma su cui sono state eseguite le simulazioni è un 3DS XL, mentre il gioco usato è Pokémon Zaffiro Alpha.

Ogni simulazione consisteva nell'andare in una Arena delle Virtù del gioco, iniziare una Gara Pokémon di Bellezza di livello Master scegliendo come Pokémon Speranza, con

le mosse uguali a quelle implementate nel programma. Poi si attendeva la valutazione preliminare e, nel programma di simulazione scritto appositamente `dssim.lua`, si inserivano i quattro Pokémon scelti dal 3DS e il loro ordine. Di lì in poi, a ogni turno il programma restituisce la mossa da utilizzare, utilizzando l'MCTS con 10000 iterazioni. Dopo aver cominciato il turno nel 3DS, si inserivano nel programma di simulazione tutti i dati della partita, ovvero mosse eseguite dagli avversari e risultati degli eventuali eventi casuali. Una volta finito il turno, il programma di simulazione suggeriva la prossima e così via.

La taglia del campione è 55.

4.3.1 Risultati



Questa volta i test di Shapiro mostrano che, per tutti i campioni, possiamo accettare l'ipotesi nulla di distribuzione normale. Quindi procediamo con test parametrici, ovvero l'ANOVA a una via e il test post-hoc di Tukey.

```
> oneway.test(hearts ~ pokemon)

        One-way analysis of means (not assuming equal variances)
```

```
data:  hearts and pokemon
F = 37.612, num df = 5.000, denom df = 94.738, p-value < 2.2e-16

> TukeyHSD(aov(hearts ~ pokemon), ordered = T)

  Tukey multiple comparisons of means
    95% family-wise confidence level
    factor levels have been ordered

Fit: aov(formula = hearts ~ pokemon)

$pokemon
                       diff        lwr       upr     p adj
Tropica-Plumy     4.3965438  1.604861  7.188226 0.0001422
Betta-Plumy       4.5625616  1.720288  7.404835 0.0000977
Trod-Plumy        4.8188312  2.072377  7.565285 0.0000142
Macy-Plumy        5.7359073  3.066958  8.404856 0.0000000
Speranza-Plumy   12.8233766 10.375921 15.270832 0.0000000
Betta-Tropica     0.1660178 -2.758166  3.090202 0.9999837
Trod-Tropica      0.4222874 -2.408851  3.253426 0.9981352
Macy-Tropica      1.3393636 -1.416652  4.095379 0.7285136
Speranza-Tropica  8.4268328  5.884713 10.968952 0.0000000
Trod-Betta        0.2562696 -2.624767  3.137306 0.9998496
Macy-Betta        1.1733458 -1.633904  3.980595 0.8355626
Speranza-Betta    8.2608150  5.663239 10.858391 0.0000000
Macy-Trod         0.9170762 -1.793116  3.627268 0.9260143
Speranza-Trod     8.0045455  5.512179 10.496912 0.0000000
Speranza-Macy     7.0874693  4.680777  9.494162 0.0000000
```

Riguardo al test di Tukey, il test pone come ipotesi nulla che le medie tra una coppia siano uguali. Quindi, un p-value alto permette di accettare l'ipotesi nulla di uguaglianza delle medie. Pertanto, possiamo notare che il gruppo Betta-Tropica-Macy-Trod non presenta differenze significative e, in particolare, Speranza si staglia nettamente contro gli altri avversari.

È importante notare come il 3DS tenda a privilegiare delle strategie aggressive. Infatti, i Pokémon avversari tendono a usare spesso mosse che danneggiano gli avversari, piuttosto che mosse che potrebbero fargli guadagnare molti cuori. Un esempio emblematico di questo comportamento è Plumy.

Plumy possiede due mosse che sono entrambe utili da usare a fine turno: una è Fiortempesta, che fa guadagnare 2 cuori e ne toglie 2 a tutti i Pokémon che si sono esibiti precedentemente; l'altra è Petalodanza, che fa guadagnare ben 6 cuori ma, se l'utilizzatore viene colpito da una mossa che lo spaventa, perde il doppio dei cuori. Come si può ben capire, sono entrambe molto buone da usare in ultima o penultima posizione, la prima per spaventare più Pokémon possibili e la seconda per non correre il

rischio di essere spaventati e perdere molti cuori. Tuttavia, l'intelligenza artificiale del 3DS preferisce quasi sempre usare Fiortempesta, mentre l'algoritmo MCTS preferisce quasi sempre Petalodanza.

Tuttavia, si può notare come la strategia che ottimizza i cuori dell'MCTS riesce a funzionare molto bene e battere senza molti problemi tutti i possibili avversari schierati dal 3DS.

Le percentuali di vittoria sono le seguenti:

Speranza	**Tropica**	**Plumy**	**Macy**	**Betta**	**Trod**
70.91%	9.68%	5.71%	16.22%	6.90%	9.09%

Le percentuali di vittoria confermano le analisi. È interessante notare come le strategie del 3DS fanno giocare molto meglio Trod rispetto all'MCTS, mentre invece sfavoriscono molto Plumy.

4.3.2 Confronto con i risultati precedenti

Avendo tutte le simulazioni, è possibile confrontarle tra di loro per avere una visione di insieme. Il confronto tra simulazioni interne al programma e contro il 3DS sono sensate perché, oltre a rispettare le stesse composizioni delle squadre, sono fatte in modo simile (per esempio, c'è il campione *MCTS sbilanciato* in cui Speranza usa un MCTS con 10000 iterazioni, lo stesso che usa Speranza nel campione contro il 3DS).

In figura 4.1 vengono mostrati i quattro boxplot delle simulazioni affiancati.

Eseguendo i test, si è riusciti ad isolare alcuni gruppi che tra di loro mostrano differenze significative di mediana (in caso di test non parametrici) o media (in caso di test parametrici). Nella tabella 4.3.2 sono mostrati, per ogni simulazione, i cuori di scarto della media della partita e i vari raggruppamenti.

Inoltre, confrontando le percentuali di vittoria si può notare che Speranza è riuscita a ottenere un notevole incremento di prestazioni all'aumentare delle iterazioni. La tabella 4.3.2 mostra l'evolversi delle percentuali di campione in campione.

4.4 Prestazioni al variare delle iterazioni

Infine, come ulteriore analisi, si è scelto di confrontare le prestazioni dei singoli giocatori al variare delle iterazioni. Sono state eseguite 500 partite facendo giocare tutti i giocatori con MCTS a 1, 10, 100, 1000 e 10000 iterazioni, in modo equilibrato (quindi tutti hanno giocato a parità di iterazioni).

I dati sono stati organizzati come già fatto in precedenza, calcolando lo scarto di cuori dalla media della partita per ogni giocatore, ma è stata aggiunta una ulteriore colonna, *iter*, che contiene il numero di iterazioni che il giocatore ha utilizzato per decidere le mosse da eseguire. In questo modo, si ha un unico grande dataframe con tutti i campioni da analizzare.

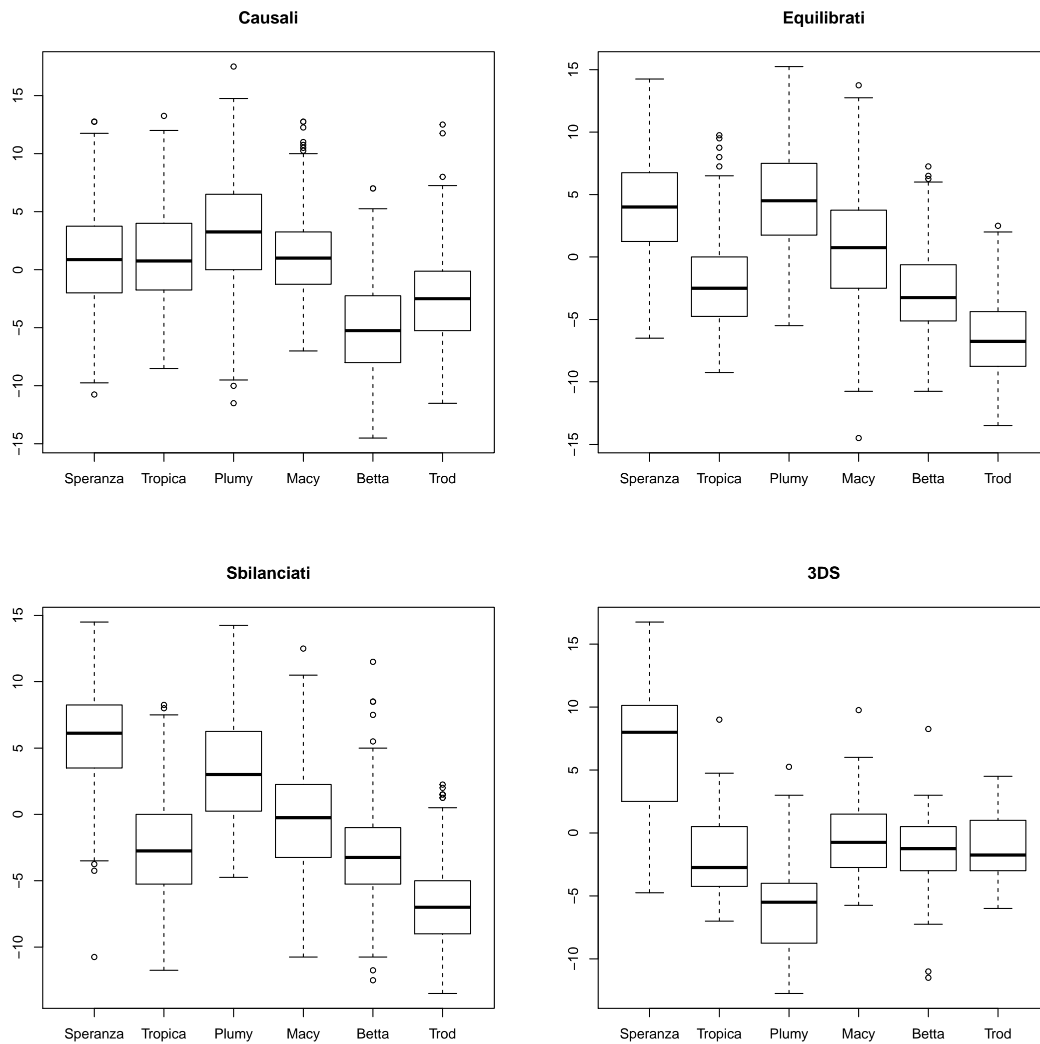


Figura 4.1: Boxplot di confronto delle simulazioni.

4.4.1 Analisi qualitativa

Utilizzando questo grande dataframe è stato realizzato un boxplot per ogni giocatore, mettendo in relazione lo scarto dei cuori dalla media della partita e il numero di iterazioni. Non è stata eseguita una analisi rigorosa ma qualitativa, per avere un'idea di come l'aumento delle iterazioni potesse influire sulle prestazioni dei singoli giocatori.

Se si fosse eseguita una tale analisi semplicemente sul numero assoluto di cuori guadagnati durante le partite, ci sarebbe stata una crescita, seppur leggera, del numero di cuori guadagnato.

Casuali	
Pokémon	**Media**
Plumy	3.269934
Macy	1.201433
Tropica	1.179530
Speranza	0.874500
Trod	-2.345536
Betta	-4.864821

Equilibrati	
Pokémon	**Media**
Plumy	4.7272727
Speranza	4.0260000
Macy	0.7026515
Tropica	-2.1531987
Betta	-2.7345201
Trod	-6.5195925

Sbilanciati	
Pokémon	**Media**
Speranza	5.8410000
Plumy	3.3043860
Macy	-0.1988255
Tropica	-2.5964052
Betta	-2.9046823
Trod	-6.8589744

3DS	
Pokémon	**Media**
Speranza	6.70909090
Macy	-0.3783784
Trod	-1.2954545
Betta	-1.5517241
Tropica	-1.7177419
Plumy	-6.1142857

Tabella 4.1: Tabelle di confronto delle simulazioni.

	Speranza	**Tropica**	**Plumy**	**Macy**	**Betta**	**Trod**
Casuali	24.40%	32.21%	49.17%	25.80%	7.82%	10.36%
Equilibrati	44.60%	7.74%	52.19%	27.27%	7.74%	0.63%
Sbilanciati	67.00%	4.25%	29.12%	17.79%	4.68%	0.64%
3DS	70.91%	9.68%	5.71%	16.22%	6.90%	9.09%

Tabella 4.2: Tabelle di confronto delle vittorie.

Nei boxplot in figura 4.2 è mostrato l'andamento dei singoli giocatori all'aumentare delle iterazioni, con l'aggiunta in basso della percentuale di vittorie.

Il risultato interessante è che non si notano differenze significative al crescere delle iterazioni. Infatti, considerando per esempio le iterazioni da 100 in su, tutti i giocatori mantengono all'incirca lo stesso scarto di cuori dalla media della partita. La spiegazione si può ricercare nel fatto che, all'aumentare delle iterazioni per tutti i giocatori, le abilità rispettive nello sfruttare le mosse migliorano allo stesso modo e quindi gli equilibri tra i giocatori non variano molto.

Le uniche piccole eccezioni sono Speranza, che ha un andamento che tende a un leggero aumento, e Trod, che ha un andamento che tende al ribasso. Questo è spiegabile in quanto Speranza, come già ripetuto più volte, possiede molte mosse che possono garantirle un guadagno di un gran numero di cuori, in particolare grazie a un uso oculato di Mimica. Quindi, più Speranza gioca bene e più guadagna cuori, di conseguenza ottiene uno scarto dalla media della partita maggiore tanto meglio gioca. Trod, dal canto suo, per quanto possa tentare di giocare bene non riuscirà mai a guadagnare molti cuori, essendo che difficilmente riesce a guadagnarne più di 3 per turno. Quindi, nonostante

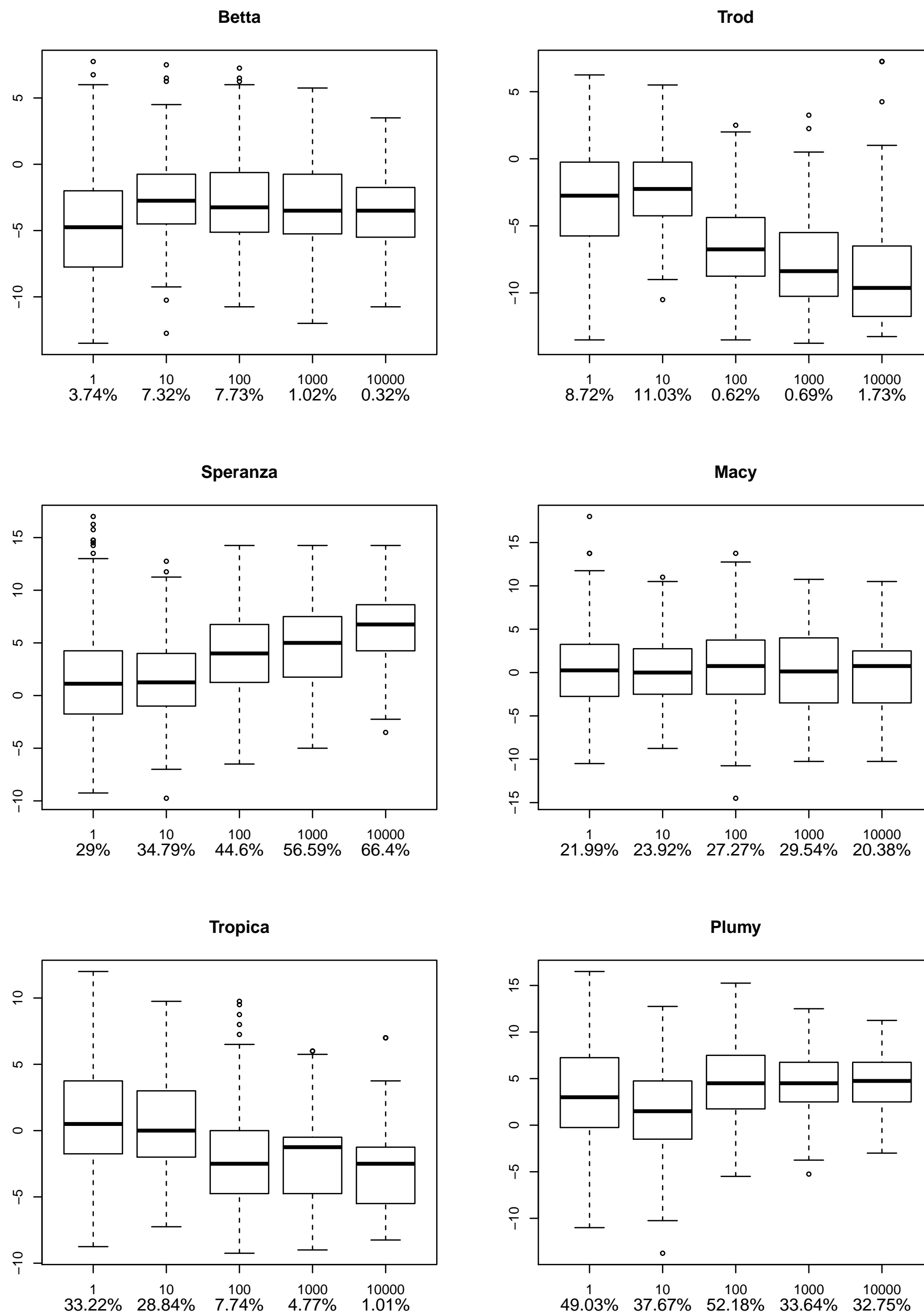


Figura 4.2: Boxplot di confronto dei giocatori con MCTS equilibrato.

l'algoritmo tenda a fargli eseguire mosse migliori, non riuscirà a guadagnare molti cuori: rispetto agli altri, quindi, avrà uno scarto sempre più basso. In pratica, gli altri giocatori

migliorano molto di più di Trod.

4.4.2 Cosa si può migliorare

I risultati sono molto soddisfacenti e mostrano come l'algoritmo riesca a sfruttare appieno le potenzialità delle mosse di Speranza. In particolare, le prestazioni contro l'intelligenza artificiale del 3DS sono molto promettenti.

Tuttavia, i margini di miglioramento sono molto ampi. Di seguito alcune idee:

- Focalizzarsi più su altri giocatori: infatti per tutto il lavoro della tesi ci si è sempre basati su Speranza e come creare un algoritmo che sfruttasse appieno le sue capacità. Sarebbe interessante variare l'algoritmo per adattarlo a diverse strategie, per esempio più aggressive. Trod, per esempio, mostra che questo algoritmo non lo sfrutta appieno e, anche se dubito che possa raggiungere risultati molto migliori, probabilmente ci sono altre strategie più adatte alle sue mosse.
- Implementare il gioco nella sua interezza, con tutte le mosse e tutti i possibili Pokémon incontrabili in ogni Gara di ogni livello. Infatti, sarebbe interessante scoprire come si comporta l'algoritmo in altre Gare, all'infuori di quelle di Bellezza di livello Master trattate nella presente tesi.
- Ottimizzare l'algoritmo MCTS, in particolare mediante la gestione degli eventi casuali, che potrebbe aumentare le prestazioni dell'intelligenza artificiale.
- Aggiungere un algoritmo di *machine learning*, in modo che il giocatore impari dalle partite giocate in precedenza e, quindi, si rafforzi sempre più.

Appendice A

Lua

Il Lua è un linguaggio di scripting di alto livello potente [12], veloce, leggero e facilmente incorporabile. Il nome significa *luna* in portoghese ed è, infatti, stato creato ed è tuttora mantenuto da un team a PUC-Rio, l'Università Cattolica Pontificia di Rio de Janeiro in Brasile.

Lua combina una semplice sintassi procedurale con potenti costrutti di descrizione dei dati basati su vettori associativi (le tabelle) e con semantica estensibile (quasi tutto può essere riscritto a proprio piacimento in modo semplice).

Il Lua possiede molti vantaggi:

Robusto e collaudato Lua è stato usato in molte applicazioni professionali (come Adobe Photoshop Lightroom), con un'enfasi per i videogiochi (come World of Warcraft e Angry Birds). È il linguaggio di scripting per videogiochi più usato. Ha un manuale di riferimento consistente e esistono molti libri che ne parlano. Dalla sua creazione nel 1993, sono state rilasciate e usate in svariati ambiti molte versioni.

Veloce Lua ha una meritata reputazione per le prestazioni. Molti benchmark mostrano che Lua è il linguaggio di scripting più veloce.

Portatile Lua è distribuito in un piccolo pacchetto e compila senza problemi in tutte le piattaforme che hanno un compilatore C standard. Lua funziona su tutte le varianti di Unix e Windows, su dispositivi mobile (con sistema Android, iOS, BREW, Symbian, Windows Phone), su microprocessori embedded (come ARM, per applicazioni come i Lego MindStorms), su mainframe IBM, eccetera.

Incorporabile Lua è un linguaggio molto veloce a basso impatto che è facilmente incorporabile in qualsiasi applicazione. È facile estendere Lua con librerie scritte in altri linguaggi e anche estendere programmi scritti in altri linguaggi con Lua.

Potente ma semplice Un concetto fondamentale nel design di Lua è il *metameccanismo* per implementare caratteristiche, invece di fornire un sacco di caratteristiche direttamente nel linguaggio. Per esempio, anche se Lua non è puramente un

linguaggio orientato ad oggetti, dà il metameccanismo per implementare classi e eredità. I metameccanismi permettono di usare pochi concetti di base e mantenere il linguaggio piccolo, ma al contempo lasciare che la semantica possa essere estesa in modi non convenzionali. Questo lo rende un po' più difficile da usare, per certi versi, ma molto flessibile.

Piccolo e gratuito Lua è molto piccolo: il codice sorgente e la documentazione di Lua 5.3.1 pesano solo 276KB compressi e 1.1MB non compressi. Inoltre, è un programma libero e open-source, distribuito sotto la licenza MIT, che è una licenza molto libera: permette di usare Lua per ogni scopo, anche commerciale, senza costi.

A.1 LÖVE

Per la creazione dell'interfaccia grafica del videogioco è stato utilizzato LÖVE, un *framework* che si può usare per creare giochi 2D in Lua. È libero, open-source, e funziona su Windows, Mac OS X e Linux.

LÖVE fornisce molte librerie per visualizzare elementi grafici e creare interattività. In questo modo, è stato possibile aggiungere agli script Lua, che gestiscono le regole del gioco e l'intelligenza artificiale, una interfaccia grafica che richiama quella originale del gioco per 3DS e mostra in modo immediato sia lo svolgimento della partita che l'albero che crea l'algoritmo.

Voglio approfittare di questo spazio per ringraziare la comunità di LÖVE, che mi ha sempre aiutato prontamente ad affrontare i dubbi che ho avuto durante la programmazione, e Constanza Amanda Pelissero, per aver creato tutti gli elementi grafici presenti nell'interfaccia basandosi su quelli del gioco originale.

A.2 Programma e codice sorgente

È possibile scaricare il programma che è il risultato principale di questa tesi in diverse versioni.

Versione Windows

`http://matteosilvestro.altervista.org/files/tesi/contest.zip`

Questa versione è formata da un archivio zip al cui interno è presente il file `contest.exe` che, avviato, farà partire il programma.

Versione universale

`http://matteosilvestro.altervista.org/files/tesi/contest.love`

Questa versione è formata dal file `contest.love`, che però richiede prima l'installazione del programma LÖVE, reperibile su `https://love2d.org/`, nel computer. Tale programma funziona su Windows, Mac OS X e Linux e ne permette quindi l'utilizzo su tutte queste piattaforme.

Codice sorgente

http://matteosilvestro.altervista.org/files/tesi/contest_source.zip

Infine è disponibile tutto il codice sorgente, con alcuni commenti per rendere più facile la lettura del codice. Per la visione ed eventuale modifica consiglio vivamente l'IDE ZeroBrane Studio, reperibile gratuitamente all'indirizzo http://studio.zerobrane.com/.

Il codice sorgente è rilasciato sotto lincenza MIT; per maggiori informazioni consultare il file LICENSE.

Bibliografia

[1] Peter Auer, Nicolò Cesa-Bianchi e Paul Fischer. «Finite-time Analysis of the Multiarmed Bandit Problem». In: *Kluwer Academic Publishers* (2002).

[2] Cameron Browne et al. «A Survey of Monte Carlo Tree Search Methods». In: *IEEE Transactions on Computational Intelligence and AI in Games* 4 (2012).

[3] Paolo Cermelli. «Appunti di Modelli Matematici per le Applicazioni e Elementi di Teoria dei Giochi e delle Reti». 2015.

[4] Alex J. Champandard. *Monte-Carlo Tree Search in TOTAL WAR: ROME II's Campaign AI.* 2014. URL: `http://aigamedev.com/open/coverage/mcts-rome-ii/`.

[5] Guillaume Chaslot et al. «Monte-Carlo Tree Search: A New Framework for Game AI». In: *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference* (2008).

[6] L. Stephen Coles. «Computer Chess: The Drosophila of AI». In: *Dr Dobb's* (2002).

[7] European Go Congress. *Computer program MogoTW with 7 stone handicap beat European professional 5 dan Catalin Taranu.* 2010. URL: `http://www.egc2010.fi/news.php`.

[8] Feng-hsiung Hsu. «IBM's Deep Blue Chess grandmaster chips». In: *Micro, IEEE* 19 (1999), pp. 70–81.

[9] Malvin H. Kalos e Paula A. Whitlock. *Monte Carlo Methods.* Wiley-VCH, 2008.

[10] Levente Kocsis e Csaba Szepesvári. «Bandit based Monte-Carlo Planning». In: *ECML* (2006).

[11] John McCarthy. *What Is Artificial Intelligence?* 2007. URL: `http://www-formal.stanford.edu/jmc/whatisai/whatisai.html`.

[12] PUC-Rio. *Lua: about.* 2015. URL: `http://www.lua.org/about.html`.

[13] Tapani Raiko. *The Go-Playing Program Called Go81.* Helsinki University of Technology, Neural Networks Research Centre.

[14] Stuart Russell e Peter Norvig. *Artificial Intelligence: A Modern Approach (3rd Edition).* Pearson, 2009.