Università degli studi di Torino

Dipartimento di Informatica

Corso di laurea in Informatica

Relazione di stage

Anno accademico 2014/'15

Briscola in 5: un sistema esperto per una strategia a regole

Relatore:

Dott. Roberto Micalizio

Candidato: Mattia Vinci

Desidero ringraziare il mio relatore, Roberto Micalizio,
per la gentile disponibilità e la fiducia accordatami;
la mia meravigliosa famiglia tutta,
per l'affetto e il supporto incondizionati;
i miei amici, universitari e non, e Martina,
le cui sorridenti presenze sono fonte di serenità.
Un pensiero va ai nonni Rosina e Carmelo
che tanto avrebbero desiderato vedermi "dottore".

Sommario

Questo lavoro presenta un possibile approccio alla creazione di una piattaforma per il gioco della Briscola in 5 che permetta l'interazione fra giocatori di natura indifferentemente umana e virtuale.
Inizialmente passa in rassegna alcuni fra i principali metodi di risoluzione applicati, nell'ambito dell'Intelligenza Artificiale, al contesto dei giochi da tavolo.
Analizza poi le principali peculiarità della Briscola in 5, fra cui spicca quella di essere un gioco di carte che non può essere definito nè come un ambiente competitivo, nè come uno cooperativo.
Dopo aver messo a confronto le caratteristiche del gioco con i requisiti dei metodi di risoluzione presi in considerazione, giustifica la scelta di una soluzione tramite un sistema esperto basato su regole.
Descrive quindi un framework di funzioni e regole, scritte per il motore Jess, che permetta l'implementazione di strategie di gioco offrendo contemporaneamente un metodo relativamente semplice per espandere le stesse.
Propone infine la realizzazione pratica di una piattaforma multiagente, basata sul sistema Jade, che permetta lo svolgimento di partite in rete fra giocatori virtuali che seguano la strategia a regole e giocatori umani; una piattaforma che offra inoltre ai giocatori esperti la possibilità di suggerire nuove strategie allegandole ad una qualsiasi mossa avvenuta durante una partita.

Indice

Indice III

Introduzione 1

1.1 Contributi originali 2

1.1.1 Piattaforma per il gioco della briscola 2

1.1.2 Expert system 2

1.1.3 Interfaccia per l'utente esperto 3

1.2 Struttura della tesi 4

La briscola in cinque 5

2.1 La briscola classica 5

2.1.1 Svolgimento della partita 5

2.1.2 Punteggi e ordine delle carte 7

2.1.3 La variante a 4 giocatori 7

2.2 La briscola in cinque 8

2.2.1 L'asta 8

2.2.2 La fase di gioco 9

2.2.3 Terminologia 9

Stato dell'arte 11

3.1 Intelligenza Artificiale e giochi: cenni storici 11

3.2 Ricerca nello spazio degli stati 13

3.2.1 Problemi ben posti 13

3.2.2 Lo spazio degli stati 14

3.2.3 Ricerca nello spazio degli stati per giochi a due giocatori: l'algoritmo minimax . . . 16
3.2.4 La ricerca applicata ai giochi di carte . . . 19
3.3 Teoria dei giochi . . . 21
3.3.1 Rappresentazione dei giochi . . . 21
3.3.2 Forma normale e forma estesa . . . 22
3.3.3 Il concetto di equilibrio . . . 22
3.3.4 Game theory per i giochi di carte . . . 25
3.4 Sistemi esperti . . . 27
3.5 Il caso particolare della briscola in 5 . . . 30
3.5.1 Inapplicabilità di algoritmi di ricerca e teoria dei giochi . . . 30
3.5.2 Soluzione adottata . . . 33

Architettura e interazioni degli agenti **34**
4.1 Gli agenti . . . 35
4.1.1 L'agente mazziere . . . 35
4.1.2 L'agente giocatore . . . 36
4.1.3 Comunicazione fra gli agenti . . . 36
4.2 Svolgimento della partita . . . 39
4.2.1 Apertura del tavolo e registrazione . . . 39
4.2.2 La fase dell'asta . . . 39
4.2.3 La fase di gioco . . . 40
4.2.4 La raccolta di strategie . . . 41

Strategie di gioco **42**
5.1 L'asta . . . 43
5.2 La fase di gioco . . . 45
5.2.1 La rappresentazione della conoscenza . . . 45
5.2.2 La modifica della *KB* . . . 47
5.2.3 Le regole per le strategie di gioco . . . 51
5.2.4 La decisione finale . . . 54
5.2.5 Riassumendo . . . 55

Implementazione degli agenti **57**
6.1 I *Behaviours* . . . 58
6.2 GeneralAgent . . . 60
6.3 MazziereAgent . . . 61
6.3.1 Avvio dell'agente: il metodo `setup` . . . 61
6.3.2 Chiusura dell'agente: il metodo `takeDown` . . . 63
6.4 L'agente PlayerAgent . . . 64
6.4.1 Avvio dell'agente: il metodo `setup` . . . 64
6.4.2 Alcuni metodi che s'interfacciano con il reasoner . . . 64

Sperimentazione **67**
7.1 Random vs Random . . . 67
7.2 Un giocatore a strategia vs Random . . . 68
7.3 Una squadra a strategia vs Random . . . 70
7.4 Strategia vs Strategia . . . 72
7.5 Interpretazione dei dati sperimentali . . . 73

Conclusioni **74**
8.1 Sviluppi futuri . . . 75

Appendice A: Jade **76**

Appendice B: Jess **77**

Bibliografia **78**

Introduzione

L'ambito di questa tesi è quello dell'Intelligenza Artificiale nella sua applicazione al mondo dei giochi da tavolo e in particolare a quelli di carte. Tale ambito di applicazione risulta interessante fin dagli albori della disciplina: nel mondo dei giochi, infatti, si possono trovare non solo dei problemi avvincenti, ma spesso anche modelli di situazioni che possono essere applicati nel mondo reale.
Numerosi sono gli approcci esplorati nel tentativo di affrontare i problemi sorgenti nella modellazione di un'intelligenza artificiale per i giochi; in questo lavoro vengono presi in considerazione alcuni fra i metodi più largamente utilizzati, in particolare la *ricerca nello spazio degli stati*, la *teoria dei giochi* e i *sistemi esperti*.
L'intenzione è quella di valutare e infine applicare uno dei suddetti approcci al popolare quanto singolare gioco della Briscola in cinque, con l'obiettivo di creare una piattaforma che permetta lo svolgersi di partite fra giocatori di natura indifferentemente umana o virtuale.
Il gioco della Briscola in cinque, complice il fatto di essere diffuso quasi esclusivamente in Italia, è stato raramente preso in considerazione dai ricercatori di Intelligenza Artificiale; quando anche si siano condotti studi sulla Briscola in cinque, lo si è fatto solo parzialmente, limitandosi alla sola fase più "classica" del gioco, che non presenta grandi difficoltà rispetto agli altri giochi di carte.
Qui ci si è invece dedicati proprio alla fase più peculiare di questo gioco, ovvero quella in cui ci si trova ad affrontare giocatori dei quali non si conosce il ruolo (compagni o avversari), che dev'essere invece desunto dalle loro mosse, tenendo presenti i possibili tentativi di *bluff* e depistaggio.
A seguito delle dovute valutazioni, si è deciso di porre le basi per un sistema esperto che permetta l'implementazione di strategie che rendano competitivo anche un giocatore virtuale.

1.1 Contributi originali

1.1.1 Piattaforma per il gioco della briscola

Per poter simulare efficacemente una partita di Briscola in 5 per prima cosa si è visto necessario disporre di una piattaforma software che permettesse l'interazione di più agenti di varia natura e fosse in grado di regolare una partita rispettandone i tempi e le regole.
Volendo permettere anche a giocatori umani di prendere parte al gioco, si è reso auspicabile lo sviluppo di una piattaforma multiagente che funzioni anche in rete e sia il più possibile indipendente dal sistema sottostante.
Per queste motivazioni si è deciso di usare JADE - JAVA AGENT DEVELOPMENT FRAMEWORK, una piattaforma open source per applicazioni di agenti *peer 2 peer* facilmente estendibile con codice JAVA. (Maggiori informazioni su JADE sono reperibili nell'appendice A e sul sito ufficiale Telecom Italia Lab [2001]).
Il risultato è una piattaforma che mette a disposizione dei tavoli da gioco virtuali con degli agenti "mazziere" che gestiscono le partite, degli agenti virtuali capaci di giocare seguendo diverse strategie facilmente estendibili e un'interfaccia grafica per gli agenti umani.

1.1.2 Expert system

Si sono raccolti e catalogati strategie, informazioni e suggerimenti da conoscenti, manuali, siti internet e forum dedicati alla Briscola in 5.
Successivamente si è formalizzata questa conoscenza basilare e tradotta in un sistema a regole scritto in JESS - THE RULE ENGINE FOR THE JAVA PLATFORM. (Maggiori informazioni su JESS sono reperibili nell'appendice B e sul sito ufficiale Ernest Friedman-Hill [2008]).
Queste regole sono state poi implementate all'interno di un framework sviluppato sempre in JESS che fornisse un ambiente adatto alla loro applicazione. Tale framework mette a disposizione le regole del gioco insieme a strutture adatte all'analisi della situazione e un sistema che, tramite l'uso di valori di priorità (*salience*) associati a ciascuna regola implementata, fornisca un valido supporto per la *risoluzione dei conflitti* di strategie candidate all'uso, così come la possibilità

di variare facilmente priorità ai singoli componenti del ragionamento, in modo da permettere una fase di *"tuning"* dell'insieme di regole.
Le regole implementate tentano contemporaneamente di

- analizzare le giocate altrui per tentare di evincere il ruolo dei singoli giocatori
- decidere, ad ogni mano, la carta da giocarsi, tenendo presente quando possibile dei risultati conseguiti nella fase di analisi

1.1.3 Interfaccia per l'utente esperto

In questo lavoro ci si è orientati maggiormente allo sviluppo di un framework che permettesse la raccolta e l'implementazione di strategie piuttosto che alla redazione delle strategie stesse.
Per questo motivo nell'interfaccia grafica per i giocatori umani si è introdotta la possibilità di inserire commenti associandoli alle singole giocate effettuate: tali commenti vengono a fine partita salvati insieme al contesto in cui sono stati scritti in un file di log CSV, il quale può successivamente essere analizzato per la traduzione dei consigli del giocatore umano in strategie comprensibili ed utilizzabili dall'agente virtuale.

1.2 Struttura della tesi

Vengono innanzitutto presentate, nel secondo capitolo, l'insieme delle regole che descrive il gioco *Briscola in 5* nella versione presa in considerazione in questo lavoro. Nel terzo capitolo si passano brevemente in rassegna alcune fra le più diffuse tecniche di risoluzione applicate ai giochi nell'ambito dell'Intelligenza Artificiale, valutando la loro applicabilità al gioco preso qui in considerazione. Nel quarto capitolo è descritta l'architettura del sistema multiagente che permette lo svolgimento delle partite. Il quinto capitolo è dedicato alle strategie usate dall'agente per affrontare il gioco nelle sue diverse parti. Nel sesto capitolo si dà una breve panoramica sull'implementazione più a "basso livello" della piattaforma multiagente. Il settimo capitolo espone e commenta gli esiti di alcuni esperimenti condotti. Le conclusioni presentano alcune riflessioni sul lavoro fatto così come spunti per possibili sviluppi. Le appendici forniscono qualche dettaglio sulle tecnologie utilizzate per lo sviluppo della piattaforma multiagente e del reasoner.

La briscola in cinque

Prima di spiegare in dettaglio le regole della Briscola in 5, verranno delineate quelle della Briscola classica, da cui la versione a cinque è derivata.

2.1 La briscola classica

Briscola è un popolare gioco di carte. Avente radici nei Paesi Bassi di fine Cinquecento, arrivato nella penisola grazie ai francesi, ha nel frattempo subito variazioni così profonde da poter essere considerato un gioco di origine puramente italiana. Bono [2010]
Briscola si gioca con un mazzo di 40 carte con i valori A, 2, 3, 4, 5, 6, 7, donna, cavallo, re, di semi italiani (un esempio nella Figura 2.1) o francesi.
Si può giocare in due, in quattro a coppie di due, in tre eliminando un 2 qualsiasi oppure in sei, tre contro tre, eliminando tutti e quattro i 2. Bono [2010]
I punti disponibili per ogni gioco sono in totale 120: vince chi ne realizza almeno 61. Se i punti sono 60 per entrambi i giocatori o coppie la partita è pareggiata.
I valori di presa sono nell'ordine decrescente: Asso, 3, Re, Cavallo, Donna o Fante, 7, 6, 5, 4 e 2 (Vedi Tab. 2.1).

2.1.1 Svolgimento della partita

Disposti i giocatori, il mazziere distribuisce 3 carte ciascuno e lascia una carta sul tavolo coprendola per metà con il mazzo posto trasversalmente ad essa, in modo che rimanga visibile a tutti per l'intero gioco: questa carta segnerà il seme di briscola e sarà l'ultima carta ad essere pescata.
Partendo dal giocatore a destra del mazziere (Bono [2010]) e continuando in senso

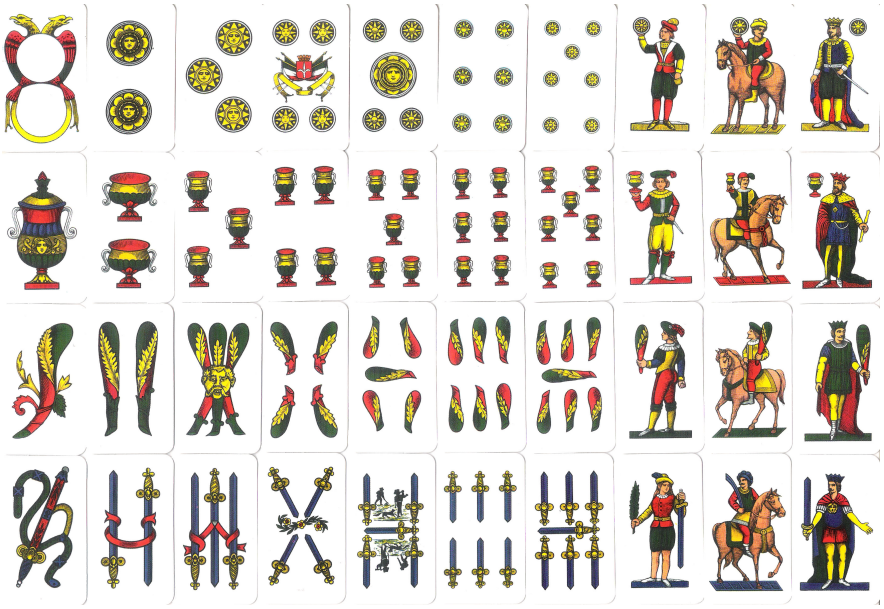


Figura 2.1: Mazzo da briscola

antiorario, (Sidoti [2010]) ogni giocatore calerà la carta che riterrà più opportuna, con lo scopo di aggiudicarsi la mano o di totalizzare il maggior numero di punti da solo o nel gioco di coppia o di gruppo. Da parte dei giocatori non esiste alcun obbligo di giocare un particolare tipo di seme, come invece avviene in altri giochi basati sull'*atout*.

L'aggiudicazione della mano avviene secondo regole molto semplici:

- il primo giocatore di mano determina il seme di mano calando la sua carta, detta dominante, diventando temporaneamente il vincitore della mano;
- la mano può essere temporaneamente aggiudicata ad un altro giocatore se questi posa una carta del seme di mano con valore di presa maggiore (si dice che il giocatore ha "strozzato" la dominante), oppure giocando una qualsiasi carta del seme di briscola, anche con valore di presa inferiore rispetto alla carta dominante. Bisogna sottolineare che non vi è obbligo di risposta al seme della dominante.

carta	A	3	K	Q	J	7	6	5	4	2
punteggio	11	10	4	3	2	0	0	0	0	0
ordine	1	2	3	4	5	6	7	8	9	10

Tabella 2.1: Carte e loro punteggi

Alla fine la mano è vinta dal giocatore che ha calato la carta di briscola col valore di presa maggiore o, in mancanza di questa, dal giocatore che ha calato la carta del seme di mano con il valore di presa maggiore. Se nessuno ha strozzato la dominante (cioè non ha giocato una carta più alta dello stesso seme) e se nessuno ha giocato una briscola, la mano è vinta dal primo giocatore di mano. Si tenga presente che il seme di mano, essendo determinato di volta in volta dalla prima carta giocata nella mano, può anche incidentalmente coincidere col seme di briscola; in questo caso la mano se la aggiudica o il primo di mano o colui che strozza giocando la briscola maggiore.
Il giocatore che vince la mano prende tutte le carte poste sul tavolo e le ripone coperte davanti a sé; in seguito sarà il primo a prendere la prima carta dal "tallone", seguìto da tutti gli altri sempre in senso antiorario e sarà il primo ad aprire la mano successiva e quindi a decidere il nuovo seme di mano.

2.1.2 Punteggi e ordine delle carte

L'ordine o potenza delle carte, ovvero la capacità di presa a parità di seme è descritta dalla Tabella 2.1. In alcune versioni del gioco il 3 è sostituito dal 7.

2.1.3 La variante a 4 giocatori

Nelle partite a 4, i giocatori si dividono in due squadre e ognuno si siede di fronte al proprio compagno. Le regole rimangono le stesse della Briscola a 2. Il primo a iniziare è il giocatore alla destra di chi ha distribuito le carte. Ad ogni turno successivo il primo a giocare è chi si è aggiudicato la mano precedente. Ad eccezione della prima mano, è permesso parlare e segnalare al compagno con gesti convenzionali le carte possedute o quella da giocare. Una volta che sono state

pescate le ultime 4 carte, i compagni si mostrano le carte passandosele prima di giocare l'ultima mano. Al termine della partita i punteggi dei compagni vengono sommati. Come nella briscola a 2 vince la coppia che ha fatto più punti.

2.2 La briscola in cinque

La *briscola a 5*, detta anche *briscola a chiamata*, *la matta*, *bugiarda* e in molti altri modi fantasiosi, è una delle varianti più popolari e divertenti della classica versione del gioco a due o quattro giocatori della briscola. Oltre al nome, la *briscola a 5* eredita dalla sua più nota parente anche il meccanismo delle prese: prende la carta più alta del seme che "domina" la mano (ovvero che è stato giocato per primo) a meno che non vi siano briscole; in quest'ultimo caso, sarà la briscola più alta a prevalere.
La peculiarità di questa versione del gioco è che a differenza della versione classica, la composizione delle squadre non è nota fin dall'inizio della mano, ma viene stabilita durante il gioco stesso, in particolare nella sua prima fase: l'asta.
Esistono moltissime varianti di questo gioco; ci si limiterà ad illustrare le regole della versione che è stata presa in considerazione in questo lavoro.

2.2.1 L'asta

Dopo che tutte le 40 carte sono state distribuite ai giocatori, che ne avranno quindi 8 ciascuno, comincia la fase dell'asta. È al termine di tale fase cruciale che verranno stabiliti i ruoli di ogni giocatore. A turno, in base alle carte che si posseggono, si può decidere di "chiamare" una carta, dicendone ad alta voce il valore (ma non il seme!), che si pensa sarebbe utile fosse in mano del proprio compagno (ancora ignoto). Ogni giocatore può chiamare una carta di valore più basso di quella chiamata dal giocatore che lo precede - oppure passare, rinunciando così alla possibilità di partecipare all'asta al turno successivo. L'asta, nella versione delle regole presa qui in considerazione, termina quando la carta più bassa è stata chiamata (il 2) oppure quando tutti i giocatori tranne uno hanno passato. Il vincitore dell'asta dichiara ad alta voce il seme della carta chiamata: chi fra gli altri giocatori possiede tale carta diventa il suo compagno, prestando

attenzione a non farlo vedere. Il vincitore dell'asta viene generalmente detto *il chiamante* o *il giaguaro*; la persona con la carta chiamata viene detta *il chiamato* o *il socio*; gli altri tre giocatori, formando una squadra che si oppone al sodalizio uscito dalla fase dell'asta, vengono detti *i compari* o *i villani*.

2.2.2 La fase di gioco

Le regole del gioco sono le stesse nella versione classica della briscola. La carta più potente della partita è l'asso di briscola. Le carte del seme di briscola vincono su tutti gli altri semi. Se in tavola non è presente la briscola, la prima carta giocata è la carta che comanda, e solo le carte dello stesso seme più alte di valore possono prendere la mano in tavola. A partire dal primo giocatore a destra del *chiamante*, tutti i cinque giocatori devono giocare una carta. Colui che effettua la presa avrà diritto ad iniziare il gioco nella mano successiva. Si continua fino all'esaurimento delle otto carte. Esaurite le carte in mano, si contano i punti, e la squadra che totalizza più della metà dei punti (> 60) vince la partita. Con 60 punti si termina con un pareggio. Generalmente si punta a vincere 2 partite su 3 o 3 su 5.
La particolarità del gioco è che solo il chiamato conosce fin dall'inizio la composizione delle squadre e si scopre agli altri solamente giocando la carta chiamata. A differenza della Briscola classica è proibito qualsiasi tipo di segnale. Lamberto [2004]
Finchè il chiamato non si svela, gli altri giocatori possono solo intuire le composizioni delle squadre in base ai comportamenti di gioco altrui.

2.2.3 Terminologia

La briscola in 5 presenta una coloratissima terminologia che varia in base alla zona geografica. Di seguito alcuni i termini usati in questo lavoro

Ruoli dei giocatori

giaguaro o *chiamante*:	colui che si aggiudica l'asta e ha quindi la facoltà di scegliere il seme di briscola
socio o *chiamato*:	colui che possiede la carta chiamata dal giaguaro; generalmente ha interesse a tenere nascosto il proprio ruolo
villani o *compari*:	gli altri tre giocatori

Tipi di giocata

carico:	una carta di valore ≥ 10 di seme diverso dalla briscola
carichino:	una carta di valore < 10 e > 0 di seme diverso dalla briscola
strozzo:	quando si gioca un carico del seme della carta (non di briscola) che sta comandando il gioco (acquisendo diritto di presa)
strozzino:	quando si gioca un carichino del seme della carta (non di briscola) che sta comandando il gioco (acquisendo diritto di presa)
liscio:	una carta non del seme di briscola senza valore
taglio:	quando si gioca una carta di briscola che batte quelle in tavolo (facendo aggiudicare temporaneamente la presa)

Stato dell'arte

3.1 Intelligenza Artificiale e giochi: cenni storici

Nell'estate del 1956 si tenne al Dartmouth College (Hanover, New Hampshire, USA) un incontro tra ricercatori americani provenienti da campi quali teoria degli automi, reti neurali e studi sull'intelligenza, organizzato dagli scienziati John McCarthy, Marvin Minsky, Claude Shannon e Nathaniel Rochester. Questo incontro è convenzionalmente considerato come la nascita ufficiale degli studi di Intelligenza Artificiale; fra i molti progetti presentati dai partecipanti, ve ne furono alcuni dedicati ai giochi, come ad esempio quello della *dama*. (Russell and Norvig [2010])

Altri lavori ancora precedenti, come il programma che Christopher Strachey scrisse all'Università di Manchester sempre sulla *dama* o quello di Dietrich Prinz per gli scacchi (Jack Copeland [2000]), contribuiscono a dimostrare come l'interesse verso il mondo dei giochi sia stato un filone presente all'interno dell'Intelligenza Artificiale fin dagli albori della disciplina. Nonostante già i primi programmi per la dama e gli scacchi fossero in grado di tenere testa ad un medio giocatore, negli anni la ricerca ha fatto passi da gigante, culminando con lo sviluppo di Deep Blue da parte dell'IBM, che fu il primo programma informatico a battere un campione mondiale di scacchi (il russo *Garry Kasparov* nel 1997). Oltre a far aumentare di 18 milioni di dollari il capitale azionario dell'IBM, (Russell and Norvig [2010]) l'evento dimostrò al mondo le potenzialità dell'Intelligenza Artificiale in un certo tipo di giochi nei quali è addirittura in grado di battere i propri creatori. Grazie alla capacità computazionale del calcolatore su cui gira, infatti, l'algoritmo per il gioco degli scacchi (scritto in C) è capace di calcolare e valutare 100 milioni di posizioni al secondo, una cifra impensabile per un giocatore uma-

no. È interessante notare come le sue funzioni di valutazione fossero scritte con parametri determinati dal sistema stesso, analizzando migliaia di partite reali di campioni dell'epoca.
Con il sofisticarsi degli algoritmi, ma soprattutto con l'aumento della capacità computazionale dei dispositivi, si sono raggiunti risultati straordinari; basti pensare che nel 2008 il software Pocket Fritz 3, installato su uno smartphone, era capace di vincere un torneo di categoria VII, riuscendo a valutare circa 20000 posizioni al secondo. (poc [2008])
È importante tener presente come il gioco degli scacchi possegga alcune caratteristiche che lo rendono particolarmente adatto ad essere facilmente ed efficientemente risolto per mezzo di alcuni tipi di algoritmi, che verranno presentati nella sezione successiva.
Altri tipi di giochi, quali i giochi di carte, non sempre si rivelano così efficientemente trattabili da una macchina. Questi infatti, a causa dell'incertezza derivante dalle mosse avversarie e dall'aleatorietà con cui le carte coperte sono ordinate nel mazzo e nelle mani altrui, necessitano di approcci differenti.
Negli anni il *Bridge* si è affermato come un ottimo caso studio esemplare di alcune delle caratteristiche principali comuni ai giochi di carte.
Alcuni dei suoi più efficaci metodi di risoluzione si basano sui già menzionati algoritmi di ricerca abbinati a metodi di decisione basati sui modelli della *teoria dei giochi*. (Ian Frank; David Basin [1998], Pavel Cejnar [2008])

3.2 Ricerca nello spazio degli stati

È importante notare alcune caratteristiche del gioco degli scacchi che lo rendono particolarmente adatto ad essere risolto efficientemente da un Intelligenza Artificiale; esso infatti è, secondo la definizione di Russell and Norvig [2010], un ambiente:

- osservabile,
- conosciuto,
- discreto,
- deterministico.

Osservabile si dice di un ambiente le cui caratteristiche salienti (in questo caso le pedine e relative posizioni) sono note all'agente che deve effettuare una scelta. L'ambiente degli scacchi è *conosciuto* nel senso che le sue "leggi fisiche", ovvero le norme che regolano le possibili mosse delle pedine, sono risapute.
È *discreto* perchè può trovarsi in un numero finito di stati distinti gli uni dagli altri; infine è *deterministico* in quanto ogni stato è determinato esclusivamente dallo stato precedente e dall'azione svolta dall'agente.
Inoltre, sempre secondo la definizione di Russell and Norvig [2010], gli scacchi sono un ambiente *competitivo*, in quanto un agente (o giocatore), tentando di massimizzare il proprio punteggio, cerca allo stesso tempo di minimizzare quello dell'avversario.
Tralasciando quest'ultima caratteristica, in generale un agente che si trovi a dover raggiungere un *obiettivo* all'interno di un ambiente osservabile, conosciuto, discreto e deterministico, può farlo tramite una *ricerca*. Per fare ciò è però altresì necessario che il problema sia ben posto.

3.2.1 Problemi ben posti

Un *problema* può essere formalizzato in cinque componenti: (Russell and Norvig [2010])

1. Uno *stato iniziale* s_0 da cui partire.

2. Una serie di *azioni* o, nel caso degli scacchi, di *mosse*: dato un particolare stato della scacchiera s, le mosse che si possono fare rispettando le regole del gioco vengono dette *azioni applicabili in* s. La funzione virtuale ACTIONS(s) fornisce l'insieme delle azioni legali nello stato s.

3. Un *modello di transizione* è una funzione RESULT(s,a) che definisce lo stato che si raggiunge applicando l'azione a allo stato s. Un *percorso* nello spazio degli stati è una sequenza di stati connessi da una sequenza di azioni

4. Un *goal test* è una funzione che, applicata ad uno stato, permette di sapere se esso è uno stato obiettivo oppure no.

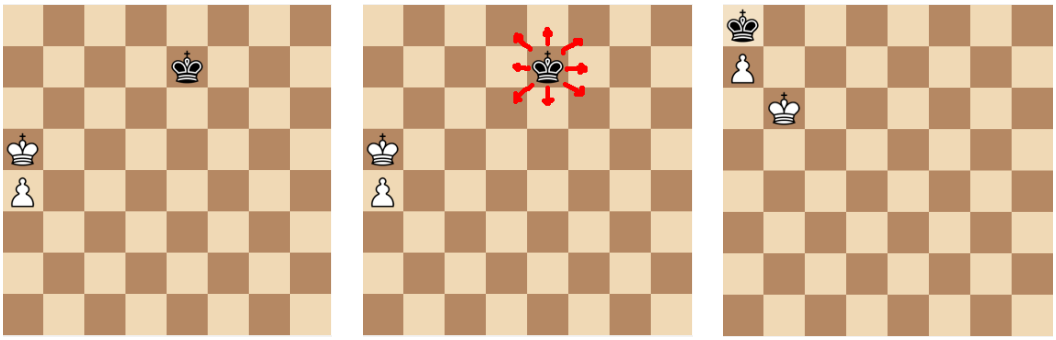
5. Una *funzione di costo* è una funzione che assegna un valore numerico (detto *costo* o *peso*) a ciascun percorso nello spazio degli stati.

Esempio applicato al gioco degli scacchi

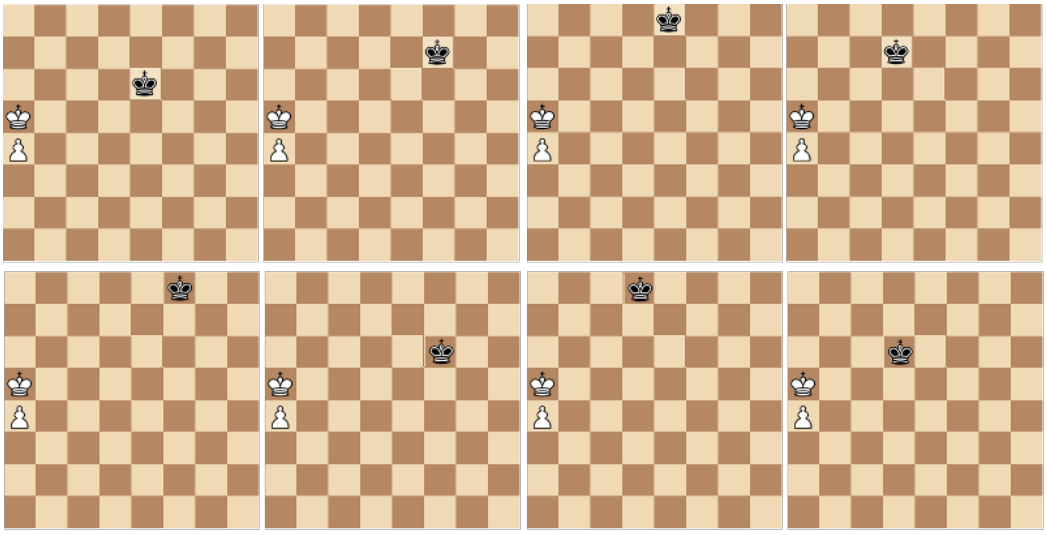
Per fare un esempio applichiamo la definizione di *problema ben posto* alla chiusura *Hansen vs. Larsen, Odense 1988*. L'agente che prendiamo in considerazione è il giocatore nero. In questo caso l'obiettivo non è quello di dare scacco matto all'avversario (obiettivo impossibile) ma di raggiungere una situazione di pareggio invece che una di sconfitta.
Lo *stato iniziale* s_0 è rappresentato dalla situazione nella Figura 3.1(a).
Le *azioni* ACTIONS(s_0), ovvero le possibili mosse applicabili allo stato iniziale, equivalgono alle sole mosse effettuabili dal re (unica pedina nera in gioco) che, secondo le regole degli scacchi, può spostarsi di una casella in ogni direzione (fig. 3.1(b)).
Il *modello di transizione* RESULT(s_0,a) è rappresentato nella Figura 3.1(d) per ogni azione legale a descritta in ACTIONS(s_0).
Lo stato di *goal* (o *obiettivo*) è rappresentato nella Figura 3.1(c).

3.2.2 Lo spazio degli stati

Dato un problema ben posto è quindi possibile rappresentarne lo *spazio degli stati* come un albero in cui



(a) Stato Iniziale s_0 (b) Azioni ACTIONS(s_0) (c) Stato *goal*



(d) Stati del modello di transizione raggiungibili da s_0

Figura 3.1: Definizione di problema ben posto applicato al gioco degli scacchi

- la *radice* rappresenta lo *stato iniziale*
- i *rami* sono le *azioni applicabili*
- i *nodi* sono gli stati appartenenti allo spazio degli stati del problema

Per costruire tale albero è possibile procedere per successiva *espansione dei nodi*: dato lo stato rappresentato da un nodo, ad esso vengono applicate tutte le azioni possibili e, tramite il modello di transizione, si ricavano gli stati successivi che diventeranno ulteriori nodi figli del precedente. L'insieme dei nodi raggiunti ma ancora da espandere viene chiamato *frontiera*. Viene detto *branching factor* il

numero (esatto o medio) di figli di ogni nodo, ovvero di azioni applicabili ad uno stato. È immediato notare come un branching factor elevato renda praticamente impossibile la costruzione ed esplorazione dell'intero spazio degli stati, data la natura esponenziale della crescita dei nodi da un livello al successivo.
Questo vale ovviamente anche per il gioco degli scacchi, il cui branching factor — ovvero il numero medio di mosse che un giocatore può effettuare — è stato calcolato essere 35 (François Dominic Laramée [2000]). Considerando una media di 50 mosse per giocatore come durata di una partita, la cardinalità dello spazio degli stati raggiungerebbe l'intrattabile numero di 35^{100}
Per questo motivo è necessario che gli algoritmi di ricerca implementino delle strategie utili a espandere solo i nodi considerati più convenienti per raggiungere un nodo goal.

3.2.3 Ricerca nello spazio degli stati per giochi a due giocatori: l'algoritmo minimax

Una volta conosciuto lo spazio degli stati si rende necessario scegliere, ad ogni livello dell'albero, quale sia la mossa ottimale fra quelle applicabili. Prendiamo qui in considerazione uno degli algoritmi più utilizzati per raggiungere tale scopo. Chiamiamo i due giocatori MIN e MAX e, seguendo Russell and Norvig [2010], formalizziamo il gioco in questi elementi

- lo stato iniziale s_0
- la funzione PLAYER(s) che indica il giocatore cui spetta muovere nello stato s
- l'insieme di *azioni applicabili* ad un dato stato s, ACTIONS(s)
- il modello di transizione RESULT(s,a)
- un *test di fine gioco*, TERMINAL-TEST(s), che indica se nello stato s la partita è conclusa
- una *funzione di utilità* UTILITY(s,p) che assegna un valore numerico allo stato finale s per il giocatore p (per esempio negli scacchi potrebbe essere 1 per la vittoria, -1 per la sconfitta e 0 per il pareggio)

Nota: l'algoritmo richiede che il gioco sia classificabile come un *zero-sum game*: ovvero un gioco in cui i valori della funzione di utilità in uno stato finale del gioco sono sempre uguali ed opposti. Questo vincolo vale per la maggior parte dei giochi trattati in AI, compreso quello degli scacchi.

Nei giochi non è possibile definire in anticipo il percorso che giunge ad uno stato di goal (uno stato terminale in cui l'agente si aggiudichi la partita) dato che il percorso che seguirà la partita è deciso anche dal (o dai) giocatore avversario. Per risolvere questa difficoltà si assume quindi che l'avversario giochi una partita ottimale; è dimostrabile come questo garantisca una decisione vincente (anche se non necessariamente la migliore) anche in situazioni in cui l'avversario non dovesse giocare ottimamente.
Prima di presentare un algoritmo che permetta di scegliere, ad ogni nodo, l'azione ottimale, è necessario definire il valore *minimax* associato ad un nodo. Informalmente, si definisce $\textsc{Minimax}(n)$ per un giocatore p la funzione di utilità che il nodo n ha associata per p, assumento che entrambi i giocatori seguano una strategia ottimale. Segue dalla definizione che il valore *minimax* di uno stato terminale è semplicemente il valore della sua *funzione di utilità*. Formalmente:

$$\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal-test}(s) \\ max_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\ min_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{min} \end{cases} \tag{3.1}$$

Il risultato di una tale definizione applicata al gioco del tris è visibile nella Figura 3.2.

L'algoritmo Minimax

L'algoritmo Minimax eseguito da un agente giocatore assegna ad ogni stato, ovvero ad ogni nodo dell'albero, il suo *valore di utilità*, applicando ricorsivamente la definizione 3.1. Minimax è quindi un algoritmo di esplorazione in profondità:

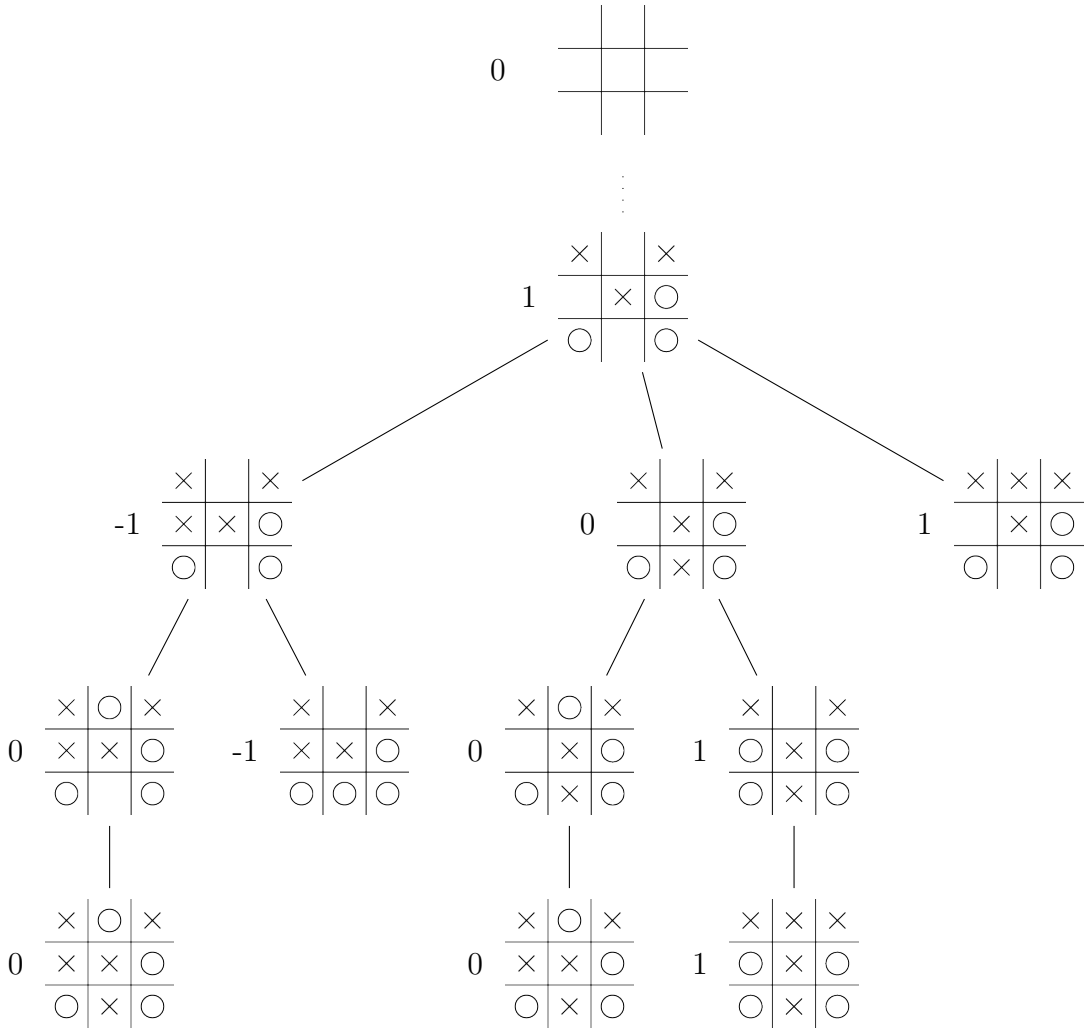


Figura 3.2: Esempio di albero degli stati per il gioco del tris, annotato con il valore *minimax* per ogni stato. Il giocatore × tenta di mazzimizzare la propria funzione obiettivo, mentre ◯ tenta di minimizzarla.

chiamando m la profondità dell'albero e b il branching factor, la sua complessità risulta essere $O(b^m)$.

L'Agoritmo 1 può facilmente essere esteso ad una situazione a più di due giocatori modificando la funzione UTILITY in modo che restituisca un vettore di valori invece che uno solo. (Si noti che nei giochi *zero sum* a due soli avversari è

Algorithm 1 L'algoritmo minimax. La funzione MINIMAX-DECISION applicata ad uno stato s restituisce la mossa ottimale applicabile in s. Russell and Norvig [2010]

```
function minimax-decision (s : state) : action
    return argmax_{a∈Actions(s)} minvalue(RESULT(s, a))

function maxvalue(s : state) : utility value
    if TERMINAL-TEST(s) then return UTILITY(s)
    v ← −∞
    for each a in ACTIONS(state) do
        v ← max(v, minvalue(RESULT(s, a)))
    return v

function minvalue(s : state) : utility value
    if TERMINAL-TEST(s) then return UTILITY(s)
    v ← ∞
    for each a in ACTIONS(state) do
        v ← min(v, maxvalue(RESULT(s, a)))
    return v
```

sufficiente un solo valore per entrambi dato che uno è l'opposto dell'altro, ma la cosa equivale ad utilizzare un vettore a due valori).
Come precedentemente anticipato la generazione dell'intero spazio degli stati, e a maggior ragione la sua visita completa in profondità, sono computazionalmente intrattabili per qualsiasi gioco non banale. Per ovviare a questo problema esistono vari metodi che permettono di generare solo un sottoinsieme interessante dell'intero spazio degli stati e di farlo mano a mano che lo si esplora. Tali metodi applicati all'albero sono detti di *pruning*.

3.2.4 La ricerca applicata ai giochi di carte

I giochi di carte presentano delle particolarità che li rendono più complessi da trattare rispetto a giochi quali la dama o gli scacchi. Secondo la definizione di Russell and Norvig [2010] infatti essi sono ambienti

- *stocastici* in oppozione a *deterministici* essendo la distribuzione delle carte

nel mazzo aleatoria; per questo motivo, a partire da due configurazioni identiche nella loro parte osservabile (scoperta), l'applicazione di una stessa strategia può portare a risultati molto diversi;

- solo *parzialmente* osservabili, in quanto le carte nel mazzo e in mano agli avversari possono essere nascoste all'agente gocatore

In questo caso l'informazione posseduta dall'agente giocatore è *incompleta* ma non del tutto *casuale*. Si può quindi tentare di applicare il metodo del MINMAX anche a questa situazione. Per farlo, si "sviluppa" l'informazione mancante, generando tutti i possibili stati s ed associando ad ognuno una probabilità $P(s)$. A quel punto si risolve ognuno di questi possibili "mondi" come se fossero *completamente osservabili* e infine si sceglie la mossa che abbia la migliore *utility function* pesata su tutti i casi in base alla loro probabilità. La mossa da effettuare è quindi:

$$argmax_a \sum_s P(s)\text{MINIMAX}(\text{RESULT}(s, a)).$$

Un ulteriore problema è che nella maggior parte dei giochi di carte il numero dei "mondi possibili" è estremamente grande. In un gioco come il *bridge*, in cui un giocatore può vedere metà delle carte, vi sono due mani nascoste di 13 carte ciascuna. Il numero di mondi possibili è quindi di $\binom{26}{13} = 10400600$ (Russell and Norvig [2010]).
Per ovviare a questo secondo inconveniente un metodo spesso utilizzato è quello di costruire un'approssimazione *Monte Carlo* dell'intero spazio degli stati. Si considera un campione casuale di N mondi possibili tale che la probabilità che uno stato s appartenga al campione è proporzionale a $P(s)$:

$$argmax_a \frac{1}{N} \sum_{i=1}^{N} \text{MINIMAX}(\text{RESULT}(s_i, a)) \text{ Russell and Norvig [2010]}$$

e a tale albero si applica l'algoritmo *minmax* o una sua derivazione.

3.3 Teoria dei giochi

La *Teoria dei giochi* è una branca della matematica che studia "modelli di conflitto e cooperazione tra agenti razionali capaci di "intraprendere decisioni", Roger B. Myerson [1991] ovvero le strategie da seguire per effettuare decisioni in maniera ottimale.
Gli ambiti applicativi delle teoria dei giochi sono diversi: Economia, Politica, Logica, Biologia, Informatica e persino Psicologia.
Sviluppatasi nella prima metà del Novecento, questa disciplina conobbe un forte aumento di interesse da parte dei ricercatori a partire dagli anni '50, dopo la pubblicazione del volume "Theory of Games and Economic Behavior" da parte del matematico John von Neumann insieme all'economista Oskar Morgenstern. Agli inizi degli anni 2000 la Teoria dei giochi ebbe modo di essere conosciuta anche dal pubblico non specialista grazie alla celebre pellicola holliwoodiana *A Beautiful Mind*, ispirata alla vita e agli studi del famoso matematico John Nash, vincitore del premio Nobel per l'Economia nel 1994 con lavori di Teoria dei giochi. (Economist [2015])

3.3.1 Rappresentazione dei giochi

Un gioco può essere formalmente definito da quattro elementi: (Rasmusen [2006])

1. i *giocatori* o, più in generale, gli agenti partecipanti al processo;
2. le *informazioni* disponibili ai giocatori in ciascun momento decisivo;
3. le *azioni* a disposizione dei giocatori in ciascun momento decisivo;
4. i *payoff* o le funzioni di utilità per ogni giocatore, per ogni possibile risultato finale.

A partire da questa formalizzazione, si può applicare al gioco una regola formale che descrive le sue previsioni su come il gioco sarà svolto dai vari giocatori, ovvero quali strategie verranno adottate; tali regole vengono dette *solution concept*. Tramite questo processo si cercano di dedurre delle strategie che conducano a situazioni di *equilibrio*.

3.3.2 Forma normale e forma estesa

I giochi in cui i partecipanti agiscono contemporaneamente, o comunque senza conoscere le azioni degli avversari, sono detti *simultanei*.
Il modello di tali giochi viene generalmente rappresentato in **forma normale**, cioè come una matrice che rappresenti i giocatori, le possibili strategie e i relativi *payoff* (un esempio: 3.1).
Se il gioco è invece di tipo *sequenziale*, allora si usa la **forma estesa**: un albero di decisione finito che ha per nodi le possibili mosse (vedi 3.3). Le mosse possono essere *personali* o *casuali*: le prime sono le mosse direttamente riconducibili alla scelta di un giocatore, le seconde invece a un evento ambientale ai cui possibili esiti è associata una precisa probabilità; fanno parte di questo genere di mosse il lancio di un dado o l'estrazione di una carta da un mazzo non ordinato. Un nodo posto alla radice, generalmente rappresentato con la forma di un diamante, rappresenta lo stato iniziale S_0 del gioco. Un "gioco" (o *play*) α viene realizzato a partire dal nodo radice, effettuando una scelta (da parte dei giocatori o del caso) ad ogni ramo e terminando in un nodo foglia.
In questo caso il *payoff* può essere assegnato tramite una *funzione di utilità* del gioco, così che il *payoff* del giocatore i sul gioco α secondo la utility function K è dato dal valore $K_i(\alpha)$. Ian Frank; David Basin [1998]

3.3.3 Il concetto di equilibrio

> Un gioco può essere descritto in termini di strategie che i giocatori devono seguire nelle loro mosse: l'equilibrio c'è, quando nessuno riesce a migliorare in maniera unilaterale il proprio comportamento. Per cambiare, occorre agire insieme. *John Nash*

La precedente citazione introduce in maniera informale il concetto di *equilibrio*, cardine della teoria dei giochi, secondo il quale la migliore strategia è quella che consente il miglior *payoff* a tutti i partecipanti senza offrire a nessuno di essi la possibilità di deviare da tale strategia guadagnandoci.

Più formalmente si può definire (Marco Li Calzi [2002]) il concetto come segue,

caratterizzando un gioco come:

- un insieme G di giocatori o agenti, di cardinalità N
- un insieme S di strategie, costituito a sua volta da un insieme di N vettori S_i, ciascuno dei quali contenga l'insieme delle strategie che il giocatore i-esimo ha a disposizione; si indichi con s_i la strategia scelta dal giocatore i;

$$S_i = (s_{i,1}, s_{i,2}, ...s_{i,j}, ...s_{i,M_i})$$

- Un insieme U di funzioni

$$u_i = U_i\left(s_1, s_2, ..., s_i, ..., s_N\right)$$

che associno ad ogni giocatore i il payoff u_i derivante da una data combinazione di strategie (proprie e degli avversari).

Un equilibrio di Nash per un gioco così definito è una combinazione di strategie

$$s_1^*, s_2^*, ..., s_N^*$$

tale che

$$\forall i \forall s_i : U_i\left(s_1^*, s_2^*, ..., s_i^*, ..., s_N^*\right) \geq U_i\left(s_1^*, s_2^*, ..., s_i, ..., s_N^*\right).$$

Informalmente: se un gioco ammette un equilibrio di Nash, ogni agente ha a disposizione una strategia s^* che massimizza il proprio payoff quando anche tutti gli altri agenti stiano seguendo una strategia scelta con lo stesso criterio.
Da questa definizione è facile comprendere il significato intuitivo di strategia di equilibrio come strategia che non permetta agli altri partecipanti di deviare da essa senza diminuire il proprio payoff: l'unica variabile su cui il giocatore i ha influenza nella formula finale è la scelta della propria strategia; ma s_i^* costituisce già la strategia che consente di massimizzare il proprio payoff.

	B non tradisce	B tradisce
A non tradisce	A: 1; B: 1	A: 3; B: 0
A tradisce	A: 0; B: 3	A: 2; B: 2

Tabella 3.1: Dilemma del prigioniero in forma normale; per i prigionieri A e B sono indicati come *payoff* (in questo caso negativi) gli anni di carcere cui vanno incontro nelle varie situazioni possibili

Esempio: il dilemma del prigioniero

Un famoso esempio di situazione in cui è utile l'applicazione della teoria dei giochi è il *Prisoner's Dilemma*, formulato per la prima volta da Merrill Flood e Melvin Dresher nel 1950 e in seguito formalizzato nella versione qui presentata da Albert W. Tucker nel 1992. Il dilemma è il seguente:

> Due criminali appartenenti alla stessa banda vengono arrestati e imprigionati. Entrambi si trovano in una situazione di isolamento senza alcuna possibilità di scambiare messaggi con l'altro. Gli accusatori non hanno le prove necessarie per condannare i due in base all'accusa principale; i due infatti sperano di essere condannati a un anno di prigione per l'accusa minore.
> Gli accusatori però offrono loro l'opportunità di tradire ciascuno l'altro per vedersi ridotta la pena. Queste le condizioni precise:
>
> - se entrambi tradiscono, vengono condannati a scontare 2 anni di pena;
> - se uno solo dei due tradisce l'altro, il traditore viene immediatamente liberato, mentre l'altro condannato a 3 anni;
> - se nessuno dei due accusa l'altro, entrambi vengono condannati a un anno di pena.

Si chiamino i due prigionieri A e B. La situazione è descritta dalla forma normale 3.1 e dalla forma estesa 3.3. Si può notare che entrambi i prigionieri ottengono un maggior *payoff* tradendo il compagno piuttosto che cooperando con lui rimanendogli fedele. Si consideri il prigioniero B: può tradire il compagno o cooperare

con lui. Se A collaborasse, a B converrebbe tradire, dato che in questo modo uscirebbe di prigione invece che scontare un anno di pena. Se invece A tradisse, anche a B converrebbe farlo, dato che sconterebbe un solo anno invece che 3. Si conclude che in ogni caso al giocatore B conviene tradire. Lo stesso ragionamento può essere seguito simmetricamente per A.

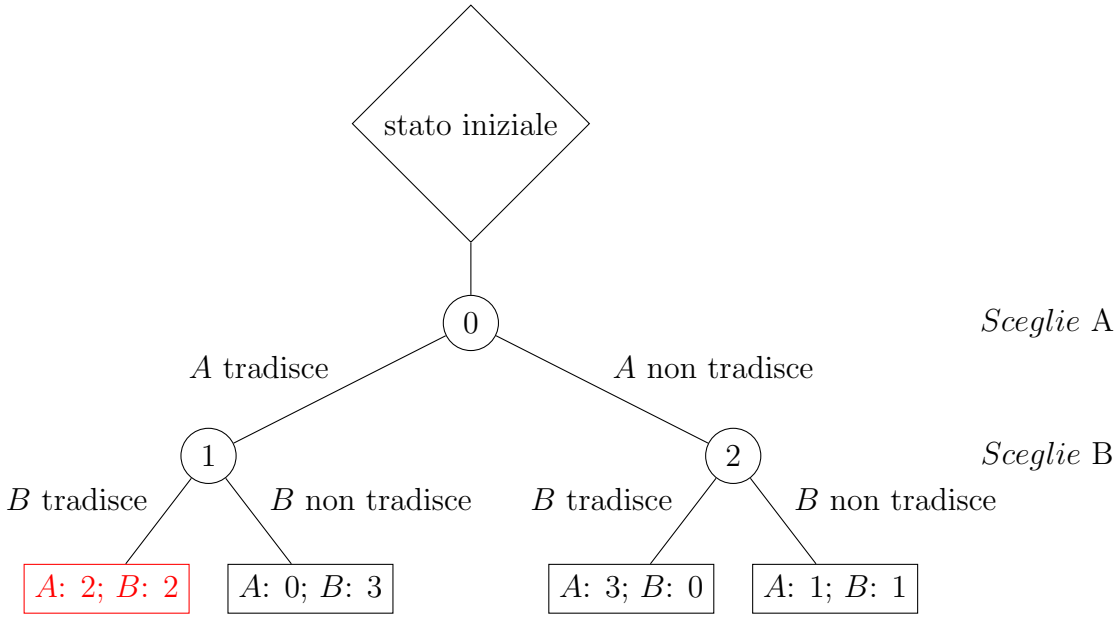


Figura 3.3: Il dilemma dei prigionieri in forma estesa; in rosso la situazione di equilibrio

3.3.4 Game theory per i giochi di carte

Secondo la classificazione della teoria dei giochi, i giochi di carte appartengono alle seguenti categorie di giochi:

- *asimmetrici*: i payoff risultanti da una stessa strategia dipendono da quale giocatore la segue;
- *sequenziali* in quanto si sviluppano nel tempo e i giocatori hanno la possibilità di muovere uno alla volta dopo aver visto le mosse altrui;

- *a informazione imperfetta* perchè le carte nelle mani altrui non sono visibili.

Nonostante alcuni giochi di carte siano sia *competitivi* che *cooperativi*, come ad esempio il *Bridge* o la *Briscola a 4* in cui due squadre di giocatori fra loro cooperanti si affrontano competitivamente, è generalmente possibile semplificare le dinamiche del gioco considerando i giocatori di una stessa squadra come un unico agente, riconducendo l'ambiente ad uno semplicemente *competitivo*. (Pavel Cejnar [2008])
Per la classificazione cui appartengono, i giochi di carte sono adatti ad essere rappresentati in forma estesa, tramite un *albero di decisione* che ne rappresenti tutte le possibili mosse da parte di tutti i giocatori e comprenda per i nodi foglia, cioè le mosse finali, i relativi *payoff*.
Essendo i giochi di carte ad *informazione imperfetta* sorgono alcune complicazioni nella costruzione della forma estesa del gioco. Infatti, alcune scelte possono non essere direttamente disponibili al giocatore; questo può coinvolgere sia le scelte effettuate dal caso, come quando si tratti della distribuzione delle carte in mano all'avversario, sia le scelte dell'avversario stesso, come quando questo giochi una carta mantenendola però coperta.
Questo fa sì che un giocatore possa non sapere dove si trovi all'interno dell'albero della forma estesa.
Nonostante alcune soluzioni per superare queste difficoltà siano già state formulate, (si veda R.D. Lute; H. Raiffa [1957]) per la maggior parte dei giochi di carte l'albero rimarrebbe comunque troppo esteso per poter essere trattato computazionalmente in pratica. (Ian Frank; David Basin [1998])
Anche in questo caso quindi, valgono le considerazioni fatte per gli algoritmi di ricerca e la possibilità di utilizzare strategemmi approssimativi unite ad algoritmi di pruning (Pavel Cejnar [2008]) per rendere l'esplorazione eseguibile in un tempo ragionevole.

3.4 Sistemi esperti

I *sistemi esperti*, o *sistemi a regole*, costituiscono un approccio per affrontare problemi che simula la conoscenza umana. Sono costituiti da due componenti principali:

- Una *base di conoscenza* o *knowledge base* (KB);
- un *motore inferenziale* che combina ed applica le nozioni contenute nella KB.

I sistemi esperti possono essere *a regole* oppure *basati su alberi*. In questo lavoro vengono presi in considerazione i sistemi esperti a regole.
Lo sviluppo di un sistema esperto segue generalmente le seguenti fasi: (Friedman-Hill [2003])

1. *knowledge engineering*: innanzitutto è necessario raccogliere la conoscenza relativa all'ambito del problema che si vuole affrontare. Per questo si fa riferimento sia a testi e manuali che a interviste a persone esperte nel campo.
2. Segue una fase di *strutturazione della conoscenza* durante la quale le informazioni raccolte precedentemente vengono organizzate in maniera tale da essere facilmente formalizzate in regole comprensibili dal sistema esperto.
3. Una prima fase di *testing* va effettuata a questo punto;
4. viene poi costruita l'*interfaccia* che permette al sistema esperto di comunicare con eventuali altri componenti (database, interfacce utente, sensori...).
5. Infine avviene la *codifica delle regole* raccolte, che portanno così finalmente essere eseguibili.

Le regole sono codificate come degli statement **if - then**: `if` *condizione* `then` *azione*. Questo fa sì che le regole siano costituite da due parti principali: un *antecedente* che descrive una condizione che può verificarsi; un *conseguente* che definisce l'azione da essere eseguita quando l'antecedente si verifica.

Esempio

Per esempio si potrebbe avere una KB contenente la regola 3.1: in tal caso, se si verificassero i fatti `piove` e `finestra-aperta` verrebbe eseguita l'azione `chiudi-finestra` e ritirato il fatto `finestra-aperta`.

```
1  if
2     piove and finestra-aperta
3  then
4     exec      chiudi-finestra
5     retract   finestra-aperta
6  end
```

Listing 3.1: Esempio di regola per sistemi esperti

Fra i sistemi esperti basati su regole più diffusi vi sono CLIPS, scritto in C, con un'interfaccia simile al LISP, e JESS, che eredita l'interfaccia da CLIPS ma si basa su JAVA.
I sistemi esperti possono servire a diversi scopi; nella Tabella 3.2 riportiamo la classificazione fornita in Hayes-Roth [1983] per una panoramica sulla diversità dei contesti in cui questo tipo di soluzione viene adottata.

Category	Problem Addressed	Examples
Interpretation	Inferring situation descriptions from sensor data	Hearsay (Speech Recognition), PROSPECTOR
Prediction	Inferring likely consequences of given situations	Preterm Birth Risk Assessment
Diagnosis	Inferring system malfunctions from observables	CADUCEUS, MYCIN, PUFF, Mistral, Eydenet, Kaleidos
Design	Configuring objects under constraints	Dendral, Mortgage Loan Advisor, R1 (Dec Vax Configuration)
Planning	Designing actions	Mission Planning for Autonomous Underwater Vehicle
Monitoring	Comparing observations to plan vulnerabilities	REACTOR
Debugging	Providing incremental solutions for complex problems	SAINT, MATHLAB, MACSYMA
Repair	Executing a plan to administer a prescribed remedy	Toxic Spill Crisis Management
Instruction	Diagnosing, assessing, and repairing student behavior	SMH.PAL, Intelligent Clinical Training, STEAMER
Control	Interpreting, predicting, repairing, and monitoring system behaviors	Real Time Process Control, Space Shuttle Mission Control

Tabella 3.2: Categorizzazione degli expert system in base al contesto di utilizzo, Hayes-Roth [1983]

3.5 Il caso particolare della briscola in 5

La briscola in 5 presenta una peculiarità rispetto alla maggior parte degli altri giochi di carte e da tavolo in generale: la suddivisione dei cinque giocatori in due squadre, necessariamente assimetriche, non avviene prima dell'inizio della partita, bensì all'interno della stessa. Inoltre, i giocatori posseggono una diversa quantità d'informazione riguardante la formazione delle squadre, che per alcuni può rimanere incerta per gran parte della partita.
Per questo è auspicabile considerare le due fasi del gioco separatamente: dapprima vi è una fase in cui la formazione delle squadre non è nota a tutti; una volta che il *socio* gioca la carta chiamata, palesandosi, ha inizio la seconda fase, durante la quale i ruoli di tutti i giocatori diventano chiari.
La seconda fase del gioco quindi non si discosta troppo da altri giochi già molto trattati nell'ambito dell'AI, come il *Bridge*. Uno studio di Andrea Villa [2013] ad esempio, applica a questa fase della briscola in 5 il metodo della ricerca nello spazio degli stati con approssimazione Montecarlo, soluzione già indicata come comune nella risoluzione di giochi di carte.
Non siamo invece riusciti a trovare studi che trattino la prima fase della briscola in 5, quella durante la quale non è chiaro a tutti chi siano gli avversari e chi i compagni. Infatti, quello della briscola in 5 nella sua prima fase non può essere considerato un ambiente nè *competitivo* nè *cooperativo*.
Questo rende impossibile l'applicazione diretta dei metodi indicati nelle precedenti sezioni così come sono e chiama un approccio di tipo differente.

3.5.1 Inapplicabilità di algoritmi di ricerca e teoria dei giochi

Volendo tentare di applicare l'algoritmo Minimax alla risoluzione di una partita di briscola in 5 ci si scontra immediatamente con una difficoltà insormontabile per l'algoritmo nella sua versione classica: essendo impossibile definire con certezza una funzione di utilità, è inapplicabile la definizione ricorsiva 3.1 del *mini-max value*. Questo accade perchè, pur assumendo di aver costruito l'intero albero rappresentante lo spazio degli stati fino alle foglie che contengono gli stadi finali

della partita, è impossibile assegnare un segno, positivo o negativo, al modulo delle funzioni di utilità: esso sarebbe positivo se il giocatore che si aggiudica la presa fosse un compagno, viceversa, sarebbe negativo. Ma essendo sconosciuto il ruolo del giocatore che "prende", rimane incerto il valore della funzione di utilità. Per poter applicare l'approccio MINIMAX anche in questo scenario, si renderebbe necessario aumentare il numero di mondi possibili, moltiplicandolo per 3 (il numero massimo dei giocatori dei quali non si conosce il ruolo) per considerare tutte le situazioni possibili. Ma in questo caso, essendo le funzioni di utilità dei differenti mondi diametralmente opposte, sarebbe anche auspicabile possedere delle valide euristiche che permettano di assegnare con buon grado di confidenza una probabilità ad ogni possibile "mondo terminale" (ovvero alla possibilità che ognuno dei 3 giocatori di cui non si conosce il ruolo sia il socio).

Esempio

Si può immaginare una situazione in cui un giocatore, P, nel ruolo di *villano*, durante la fase in cui il *socio* non si sia ancora rivelato, si trovi ultimo di mano in una partita contro i giocatori A, B, C e G. G sia il *chiamante*.
In tavolo non vi siano briscole; il seme regnante sia ♡.
La presa spetti al giocatore, A del quale ancora non si conosce il ruolo.
A prenda con un Asso di ♡.
In tavolo vi siano anche: K ♡, J ♢, 2 ♠.
Il seme di briscola sia ♣.
In mano si abbiano 2 ♣ e 2 ♡ : una briscola ed un liscio.
Immaginiamo, per facilitare i calcoli, che quasi tutti i punti siano già stati giocati e che rimanga una sola mano da giocarsi con un punteggio inferiore a quello in tavola.
In una situazione del genere un giocatore umano incerto sui ruoli degli altri giocatori giocherebbe certamente la briscola (quindi il 2 ♣), aggiudicandosi la mano senza per questo dover rinunciare a dei punti.
Ma la situazione è utile per mostrare l'inapplicabilità della definizione del *minimax value*: se si decidesse di giocare il 2 ♡, infatti, i punti raccolti dal giocatore A che ha giocato A ♡, sarebbero a proprio vantaggio o svantaggio?

Giocando il 2 ♣ si è certi che, qualsiasi sia lo svolgersi della mano successiva, il punteggio della propria squadra aumenterà di 16 punti.
Giocando il 2 ♡, invece, i 16 punti della mano potrebbero essere a favore della propria squadra come di quella avversaria, in base all'affiliazione del giocatore che si aggiudica la presa.
Allo stesso tempo, se il giocatore P fosse sicuro di essere in squadra con A, dovrebbe certamente "andare liscio" giocando il 2 ♡, mantenendo in mano in questo modo una briscola che potrebbe rivelarsi molto utile nella mano successiva (anche se in questo caso abbiamo ipotizzato, per semplificare, che tutti i punti siano già stati giocati).

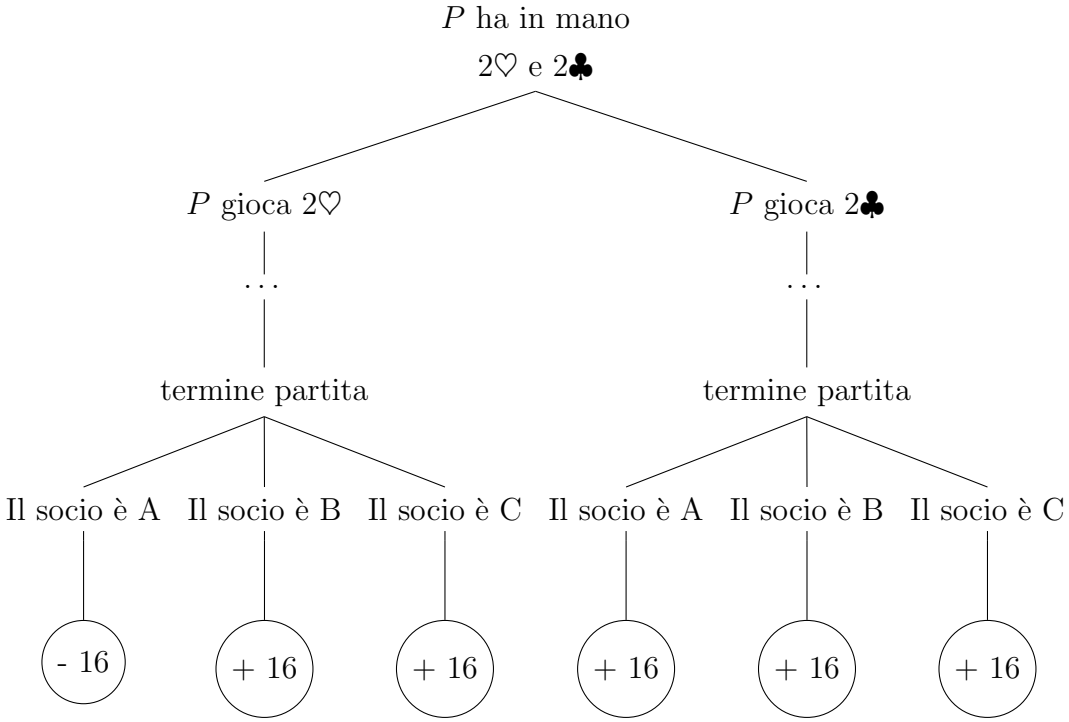


Figura 3.4: Valori indefiniti per la utility function

Come è stato detto, l'applicazione del modello della teoria dei giochi alla Briscola in 5 richiede comunque lo sviluppo di un albero di decisione; oltre ai già trattati (e superabili) problemi di estensione di tale albero, il problema descritto in 3.4

si ripresenta nel momento in cui si voglia applicare il modello della teoria dei giochi a un ambiente come quello della Briscola in 5 che non è definitamente nè competitivo nè cooperativo.
Esattamente come i valori della *utility function* usati dall'algoritmo MINIMAX, anche i *payoff* necessari alla realizzazione della forma estesa del gioco della Briscola in 5 non sarebbero definibili se non conoscendo in anticipo la formazione delle squadre.

3.5.2 Soluzione adottata

Per le difficoltà di cui sopra, si è deciso di scartare per ora un approccio puramente algoritmico per la soluzione. Si è invece optato per gestire la fase di gioco tentando di simulare i comportamenti umani di giocatori esperti.
Questo viene fatto sia mettendo a disposizione un insieme di strategie basilari all'interno di un framework appositamente pensato per facilitare la scrittura delle stesse, sia proponendo un sistema che permetta all'utente esperto di suggerire nuove strategie per l'ampliamento, l'integrazione e la modifica di quelle pre-esistenti.
L'utilizzo di questo tipo di approccio per la prima fase del gioco non esclude la possibilità di integrarlo con metodi di soluzione più classici come quelli accennati nelle precedenti sezioni: inadatti alla prima fase della partita, questi potrebbero infatti benissimo essere usati per la soluzione della seconda, una volta che le squadre siano a tutti note.

Architettura e interazioni degli agenti

In questo capitolo viene presentata l'architettura generale del sistema e si descrivono le interazioni che i singoli componenti hanno fra di loro.
L'architettura è distribuita su diversi agenti indipendenti e progettata in maniera tale da poter funzionare anche in rete. È costituita da due tipi di agenti: l'agente **mazziere** e l'agente **giocatore**.
Questi due tipi di agenti comunicano fra di loro direttamente ed esclusivamente tramite scambi di messaggi.
La regolamentazione della partita è affidata interamente all'agente **mazziere**; questa centralizzazione presenta alcuni vantaggi, tra cui:

- l'assicurazione che la partita venga svolta secondo le regole, qualsiasi sia il comportamento dei singoli agenti giocatori;
- una più semplice gestione di piattaforme in cui si vogliano mantenere aperti dei "tavoli virtuali" per un numero imprecisato di giocatori;
- l'opportunità, da parte di un agente onniscente, di redigere un file di log di ogni partita, utile per future analisi.

I singoli agenti possono comunque comunicare tra di loro direttamente tramite un servizio di chat.

4.1 Gli agenti

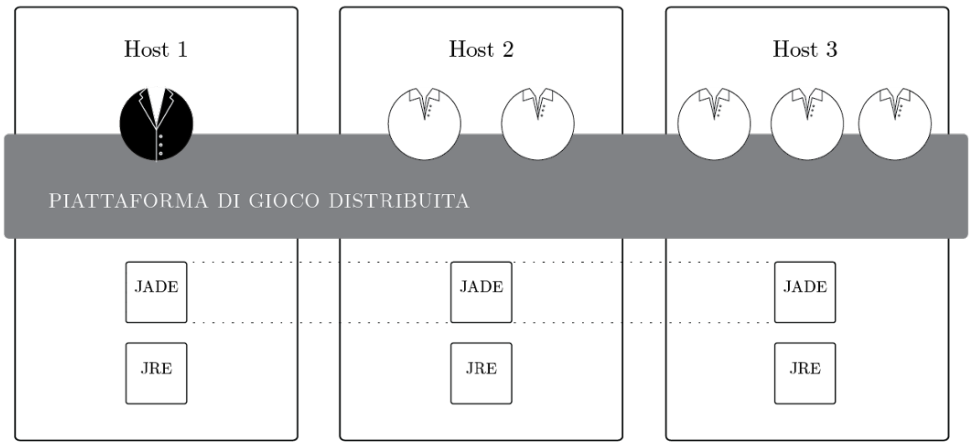


Figura 4.1: Architettura della piattaforma: nell'esempio il mazziere risiede su un host remoto e i giocatori sono suddivisi su altri due host

4.1.1 L'agente mazziere

L'agente **mazziere** regola e gestisce la partita. Ha il compito di "aprire un tavolo", ovvero rendersi disponibile per condurre una partita verso un massimo di cinque giocatori. Dopo aver raggiunto un tavolo completo (con cinque iscritti) l'agente mazziere, pur comunicando le informazioni sui giocatori seduti al tavolo a tutti gli agenti iscritti, si fa carico di gestire la comunicazione fra di essi: ogni messaggio passa prima dal mazziere che decide se e a chi dev'essere ri-spedito.
Unica eccezione a tale modalità di comunicazione avviene con la chat: in questo caso i messaggi spediti da un agente vengono immediatamente ricevuti da tutti gli altri presenti al tavolo (compreso il mazziere) (4.2).
L'agente ha anche il compito di tenere un file di log che viene compilato al termine di ogni partita e contiene, in formato `CSV`, lo storico delle giocate effettuate dai singoli giocatori, gli eventuali commenti allegati da un giocatore umano alle singole giocate, i punteggi ottenuti e i ruoli dei giocatori.

È ovviamente possibile avere più agenti mazziere in esecuzione in contemporanea sulla stessa piattaforma per permettere più partite simultaneamente.

4.1.2 L'agente giocatore

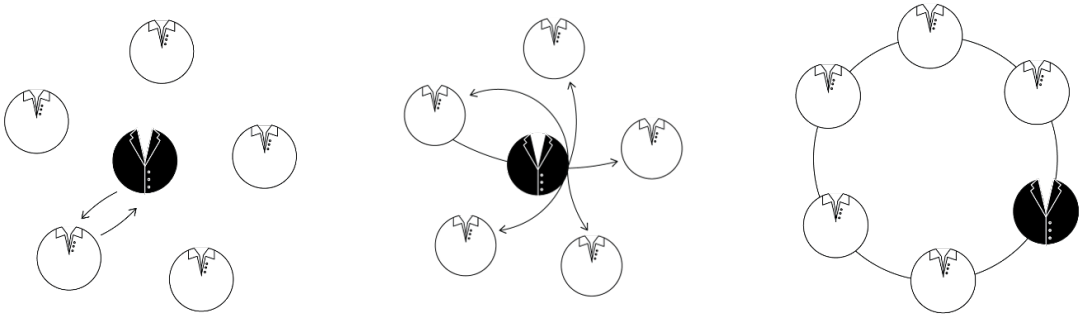
L'agente giocatore è quello che prende parte alla partita e s'impegna a fare una mossa legale quando gli è richiesta. Esso può essere inizializzato in tre principali modalità:

- Manuale
- Random
- Strategia da file

Nel primo caso sarà un giocatore umano che di volta in volta deciderà e comunicherà tramite un'interfaccia grafica le mosse da effettuare.
Nel caso in cui l'agente sia di tipo *Random*, al momento di compiere un'azione ne selezionerà casualmente una all'interno di un sottoinsieme di azioni disponibili e legali.
Caricando una strategia da file invece, l'agente deciderà la mossa da compiere dopo una fase di *ragionamento* che avverrà sulle regole implementate nel file.

4.1.3 Comunicazione fra gli agenti

Gli agenti comunicano fra di loro tramite scambi di messaggi. Fatta eccezione per i messaggi della chat, che vengono recapitati contemporaneamente a tutti gli agenti iscritti ad uno stesso tavolo (mazziere compreso), tutti gli altri messaggi passano tramite il mazziere prima di essere consegnati al (o ai) destinatari effettivi.
Questa struttura pur presentando un collo di bottiglia costituito dal passaggio di tutti i messaggi attraverso il mazziere, permette di assicurarsi che la partita venga svolta secondo le regole anche quando un agente giocatore dovesse provare a compiere una mossa illegale.



(a) Messaggio giocatore - mazziere

(b) Comunicazione di una mossa

(c) Chat

Figura 4.2: Comunicazione fra agenti

L' approccio centralizzato permette inoltre di redigere un unico file di log.
Infine, nell'ottica di una piattaforma di gioco online, la presenza di agenti mazzieri residenti su un server in attesa di giocatori da remoto facilita il coordinamento degli agenti e rende possibile l'organizzazione di campionati e classifiche a punteggi.

Messaggi bloccanti e non bloccanti

I messaggi scambiati possono essere di due tipi, *bloccanti* e *non bloccanti*.
Nel caso di messaggi ad effetto bloccante, l'agente, dopo la spedizione di un messaggio, rimane in attesa del messaggio di conferma della ricezione da parte del destinatario, interrompendo il proprio flusso di esecuzione.
L'invio di messaggi non bloccanti invece non influisce direttamente sul flusso di esecuzione del mittente. Questa divisione si è rivelata utile per distinguere fra messaggi la cui mancata ricezione può compromettere il flusso regolare della partita e messaggi la cui eventuale perdita non ha gravi effetti sullo svolgersi del gioco. Per esempio, mentre i messaggi di chat vengono inviati in maniera non bloccante, dato che la loro perdita non influirebbe sul regolare andamento della partita, i messaggi che gestiscono i turni di giocata sono inviati come bloccanti.

Formato dei messaggi

Dal punto di vista del livello dell'applicazione, i messaggi hanno formati variabili in base al loro utilizzo; restando fermi i campi *Oggetto* e *Destinatario/i* (un singolo agente o una lista di agenti), possono avere un *Identificatore della conversazione*, usato per esempio nei messaggi bloccanti per distinguere fra le conferme di avvenuta ricezione o nella chat, ed un *Contenuto* che può essere meramente testuale (es. chat) come anche un formato complesso (es. messaggi di offerta durante la fase dell'asta).

4.2 Svolgimento della partita

4.2.1 Apertura del tavolo e registrazione

Perchè degli agenti giocatori possano iniziare una partita è necessario che vi sia almeno un mazziere disponibile.
All'avvio, l'agente mazziere "apre un tavolo", ovvero si iscrive ad un *registro pubblico* offrendo i propri servigi di coordinatore.
L'agente giocatore che voglia iniziare una nuova partita, analizza il succitato registro pubblico memorizzando gli indirizzi dei mazzieri disponibili e spedendo loro una richiesta d'iscrizione.
I mazzieri risponderanno con un'offerta a validità temporanea limitata; l'agente giocatore replicherà ad una sola delle offerte fattegli, legandosi a quel punto ad un mazziere e rimanendo in attesa dell'inizio della partita, che avverrà quando cinque giocatori avranno confermato la propria partecipazione al tavolo.

4.2.2 La fase dell'asta

Appena cinque giocatori si ritrovano seduti contemporaneamente ad uno stesso tavolo, il mazziere partiziona l'insieme di carte di un mazzo da 40 in 5 sottoinsiemi da 8, inviando a ogni giocatore un messaggio contenente le carte di uno di questi sottoinsiemi.
Distribuite le carte, il mazziere dà inizio alla prima fase del gioco: la fase dell'*asta*.
Il mazziere gestisce le offerte dei vari giocatori tramite messaggi dal formato specializzato. Un messaggio usato durante la fase dell'asta ha il seguente formato:

- `bestBid`: la carta che sta aggiudicandosi l'asta (la più bassa finora chiamata)
- `bestBidder`: l'agente giocatore che sta aggiudicandosi l'asta
- `justBid`: l'ultima chiamata effettuata: può essere la carta più bassa oppure una carta nulla (*passo*)
- `justBidder`: l'agente giocatore autore dell'ultima chiamata

- `next:` l'agente da cui si attende la prossima giocata
- `counter:` il numero di chiamate fin'ora effettuate
- `done:` valore booleano che indica se l'asta è terminata (aggiudicata da `bestBidder`) o meno

L'agente giocatore che si veda indicato nel campo `next` sa di dover spedire la propria offerta legale, che può essere una carta inferiore a `bestBid` o un *passo* – abbandonando in quest'ultimo caso l'asta senza possibilità di rientrare al turno successivo.
L'ordine seguito dal mazziere per raccogliere le offerte riflette quello che si usa ad un tavolo circolare; l'ordine al tavolo ricalca l'ordine d'iscrizione degli agenti giocatori al tavolo del mazziere.
L'asta termina dopo che un giocatore chiama il 2 (la carta più bassa) o dopo che tutti hanno passato un turno.
A questo punto il mazziere chiede al vincitore dell'asta il seme della carta chiamata per poi comunicarlo a tutti e cinque i giocatori. Questi, ricevute le informazioni sul vincitore dell'asta e sulla carta chiamata, analizzeranno le proprie carte in mano per scoprire il proprio ruolo.

4.2.3 La fase di gioco

La fase di gioco è suddivisa in 8 *mani* da 5 *giocate*.
Il mazziere gestisce questa fase in maniera simile alla fase dell'asta: prima di ogni giocata, spedisce a tutti i giocatori un messaggio specializzato formato dai seguenti campi:

- `counter:` contatore della giocata
- `mano:` contatore della mano
- `justPlayer:` ultimo agente ad aver effettuato una giocata
- `justCard:` carta giocata da `justPlayer`
- `next:` prossimo agente a dover giocare

I primi due campi sono usati per identificare univocamente la giocata aiutando i processi di sincronizzazione; i campi `justPlayer` e `justCard` servono a tenere tutti i giocatori al corrente di quanto accade al tavolo; il campo `next` serve a identificare l'agente al quale si richiede la prossima giocata.
Al termine di ogni mano il mazziere calcola quale giocatore si è aggiudicato la *presa* — che sarà lo stesso agente a dover giocare la mano successiva — e comunica a tutti i giocatori tale informazione, così come il punteggio contenuto nella mano.
In base al tipo di agente giocatore (manuale, casuale, a strategia da file) la selezione della carta da giocare al proprio turno avverrà in maniera differente: rispettivamente, tramite selezione manuale dell'utente nell'interfaccia grafica, in maniera pseudocasuale o in seguito ad un ragionamento sulle strategie implementate. Una volta terminata la fase di gioco il mazziere comunica a tutti i giocatori i singoli punteggi ottenuti da ciascuno.

4.2.4 La raccolta di strategie

Durante la fase di gioco, l'agente giocatore di tipo *manuale*, offre un'interfaccia grafica che permette di selezionare singolarmente le giocate effettuate dai giocatori da uno storico della partita e di scrivere un commento da associarvi. Questo commento viene immediatamente spedito al mazziere, che ne memorizza contenuto, mittente, e associazione con la giocata.
Il giocatore manuale può associare commenti alle mosse anche dopo il termine della fase di gioco; nel caso di presenza di giocatori manuali (o in generale, con interfaccia grafica), infatti, il mazziere non considererà la partita terminata finchè non verrà premuto l'apposito bottone da parte di tutti i giocatori con interfaccia grafica.
Questo permette al mazziere di sapere quando chiudere il file di log, liberare la propria memoria dalle informazioni sulla partita appena conclusasi, disallocare le relative strutture ed iscriversi nuovamente al registro dei tavoli.

Strategie di gioco

Le due fasi principali del gioco vengono gestite in maniera molto diversa.
La *fase dell'asta* viene gestita in maniera totalmente deterministica ed automatica per ogni tipo di giocatore.
La *fase di gioco* viene invece gestita in maniere diverse, a seconda del tipo agente giocatore, che può essere *umano*, *casuale* o utilizzare una *strategia da file*.
Nel primo caso è il giocatore umano a selezionare di volta in volta, tramite l'interfaccia grafica, la carta da giocare. Nel caso di un giocatore casuale, invece, viene generato un numero pseudocasuale sulla base del quale viene estratta dal mazzo la carta da giocarsi.
Di seguito, sono descritte le strategie usate dal giocatore che implementa il sistema esperto. Essendo queste ultime soggette a modifiche ed integrazioni, ci si è soffermati particolarmente sul framework che permette l'esecuzione e la codifica delle strategie più che sulle strategie in sè.
Si presenta inoltre l'algoritmo usato per gestire la fase d'asta.

5.1 L'asta

La fase dell'asta è una fase discriminante nel gioco della Briscola in 5, in quanto è proprio durante questa fase che vengono decise le composizioni delle squadre. Obiettivo di ogni giocatore in questa fase in cui i ruoli non sono ancora assegnati è quello di ottenere la vittoria con la carta più alta possibile se si hanno buone carte in mano e viceversa di non lasciare una carta troppo alta al vincitore se non si hanno carte abbastanza buone da cercare di giocare nel ruolo del chiamante. Nonostante l'importanza di questa fase di gioco sia innegabile, i suoi meccanismi sono decisamente più deterministici e meno interessanti della fase di gioco vera e propria.
Per questa ragione abbiamo deciso di semplificare l'implementazione di tale fase seguendo un approccio già intrapreso in Andrea Villa [2013]. Per prima cosa si è eliminata la *chiamata in mano*, ovvero la possibilità da parte di un giocatore con ottime carte di chiamare una carta in proprio possesso, trovandosi quindi, in caso di vittoria dell'asta, a giocare da solo contro gli altri quattro.
Abbiamo inoltre eliminato la possibilità di continuare l'asta dopo la chiamata del 2: mentre in molte versioni della briscola è possibile rilanciare scommettendo (questa volta al rialzo) sul punteggio ottenibile al termine della partita, in questo lavoro si è deciso di far sì che il primo giocatore a chiamare il 2 si aggiudichi l'asta. La fase dell'asta è l'unica a essere gestita nello stesso modo da tutti e tre i tipi di agenti giocatori.
La gestione è deterministica e si basa sulla combinazione di due elementi che riguardano le carte in mano:

- La distribuzione dei semi
- Il punteggio totale della mano

La combinazione di queste due informazioni forma una tabella (5.1) che ha al suo interno due ulteriori informazioni:

- le carte che è necessario possedere per poter chiamare
- il limite inferiore di punteggio della carta da chiamare

punti distrib	15-20	21-25	26-30	31-35	36-40	41-45	46-50	>50
8-0-0-0	Sempre; Lim: 4	Sempre; Lim: 2	Sempre; Lim: 2	Imp.	Imp.	Imp.	Imp.	Imp.
7-1-0-0	Sempre; Lim: 6	Sempre; Lim: 5	Sempre; Lim: 4	Sempre; Lim: 2	Sempre; Lim: 2	Sempre; Lim: 2	Imp.	Imp.
6-2-0-0 6-1-1-0	Sempre; Lim: 7	Sempre; Lim: 6	Sempre; Lim: 5	Sempre; Lim: 4	Sempre; Lim: 2	Sempre; Lim: 2	Sempre; Lim: 2	Sempre; Lim: 2
5-3-0-0 5-2-1-0 5-1-1-1	A o 3; Lim: J	A o 3; Lim: 7	A o 3 o K; Lim: 6	A o 3 o K; Lim: 5	A o 3 o K; Lim: 4	A o 3 o K; Lim: 2	A o 3 o K; Lim: 2	Sempre; Lim: 2
4-4-0-0 4-3-1-0 4-2-2-0 4-2-1-1	A o 3; Lim: Q	A o 3; Lim: J	A o 3 o K; Lim: 7	A o 3 o K; Lim: 6	A o 3 o K; Lim: 5	A o 3 o K; Lim: 4	A o 3 o K; Lim: 2	A o 3 o K; Lim: 2
3-3-2-0 3-3-1-1 3-2-2-1	A o 3; Lim: K	A o 3; Lim: Q	A o 3 o K; Lim: J	A o 3 o K; Lim: 7	A o 3 o K; Lim: 6	A o 3 o K; Lim: 5	A o 3 o K; Lim: 4	A o 3 o K; Lim: 2
2-2-2-2	PASSO	A e 3; Lim: J	A e 3; Lim: 7	A e 3; Lim: 6	A e 3; Lim: 5	Due tra: A,3,K; Lim: 4	Due tra: A,3,K; Lim: 4	Due tra: A,3,K; Lim: 2

Tabella 5.1: Tabella per la gestione dell'asta. L'azione è decisa in base al punteggio totale in mano (colonne) e alla distribuzione dei semi (righe). La cella corrispondente indica quali carte sono necessarie per poter effettuare la chiamata e fino a quale carta si può ribassare.

5.2 La fase di gioco

Durante tutta la partita l'agente giocatore tenta di mantenere una conoscenza il più possibile completa della situazione al fine di selezionare la carta più vantaggiosa da giocare.
Fa questo utilizzando il già accennato metodo dell'*expert system*: data una *KB* nella quale sono di volta in volta memorizzate ed aggiornate le conoscenze formalizzate riguardo la situazione corrente, ad essa vengono applicate al momento giusto delle strategie, codificate come insiemi di regole, che a loro volta aggiornano la *KB* o eseguono azioni vere e proprie (come quella di giocare una carta).

5.2.1 La rappresentazione della conoscenza

La conoscenza formalizzata nella *KB* è suddivisa in *fatti*. Questi fatti possono essere semplici stringhe oppure, più comunemente, informazioni strutturate. In questo secondo caso va definita la loro struttura, detta *template*. (Friedman-Hill [2003]) Un template è formato da

- un nome dato all'informazione strutturata,
- degli *slot* nominati, assimilabili ai nomi delle colonne di un database relazionale.

Esempio di un template

Volendo rappresentare le informazioni sugli studenti di una scuola, si potrebbe definire un template del genere:

```
( deftemplate studente
   (slot nome)      (slot cognome)      (slot matricola)
)
```

Per inserire nella *KB* le informazioni riguardo allo studente Gianluca Malatesta con matricola 147865 bisognerebbe asserire il fatto

```
(studente (nome Gianluca) (cognome Malatesta) (matricola 147865))
```

Alcuni esempi di fatti

I fatti definiti e utilizzati nella *KB* sono di due tipi se considerati in base alla loro durata temporale:

- Fatti perduranti l'intera partita
- Fatti che vengono cancellati ad ogni inizio di mano

I fatti che vengono mantenuti per l'intera durata della partita rappresentano condizioni generali (seppur mutevoli); alcuni esempi: 5.1.

```
1 ( deftemplate in-mano "carte che posso ancora giocare"
2     (slot card)    (slot rank)    (slot suit)
3 )
4
5 ( deftemplate in-mazzo "carte ancora da scoprire"
6     (slot card)    (slot rank)    (slot suit)
7 )
8
9 ( deftemplate lisci-in-mano "Le carte di tipo liscio che posso
    giocare"
10    (slot card)    (slot rank)    (slot suit)
11 )
12
13 ( deftemplate carico-piu-basso "La minor carta di tipo carico che ho
     in mano"
14    (slot card)    (slot rank)    (slot suit)    (slot points)
15 )
16
17 ( deftemplate giaguaro  (slot player) )
18 ( deftemplate socio (slot player) )
19 ( deftemplate villano   (slot player) )
20 ( deftemplate briscola (slot card) (slot rank) (slot suit) )
```

Listing 5.1: Alcuni template di fatti che rimangono nella *KB* per l'intera partita. Il contenuto di questi fatti viene aggiornato mano a mano che nuove informazioni si rendono disponibili all'agente giocatore

Altri tipi di fatti (5.2), invece, si riferiscono esclusivamente alla mano in corso e vengono quindi rimossi ogni volta che tutti e cinque i giocatori hanno effettuato la loro giocata e che il mazziere ha dato inizio ad una nuova mano.

```
1 ( deftemplate in-tavolo "carte sul tavolo"
2     (slot card)    (slot player)    (slot rank)    (slot suit)
3 )
4
5 ( deftemplate da-giocarsi "carte attivate per il gioco e loro -
    salience-"
6     (slot card)    (slot sal)
7 )
8
9 ( deftemplate turno "ordine in cui si gioca la mano"
10    (slot player)    (slot posizione)
11 )
12
13 ( deftemplate prende "chi, per ora, prende"
14    (slot player)    (slot card)    (slot rank)    (slot suit)
15 )
16
17 ( deftemplate giocata "info sulle giocate"
18    (slot player)    (slot card)    (slot rank)    (slot suit)    (
        slot mano)    (slot turno)    (slot tipo)
19 )
```

Listing 5.2: Alcuni template di fatti la cui validità è limitata alla giocata corrente

5.2.2 La modifica della *KB*

I fatti all'interno della *KB* possono essere fatti "primitivi", cioè riguardanti eventi che sono stati direttamente comunicati all'agente, come le carte possedute in mano o la giocata effettuata da un altro giocatore; oppure possono essere fatti "derivati": questo accade quando il *reasoning* applica le strategie al contenuto della *KB* e ne deduce informazioni che sono a loro volta asserite all'interno della *KB*. I fatti primitivi sono quindi adatti ad essere asseriti dall'agente giocatore stesso all'infuori del meccanismo di reasoning. Questo viene realizzato tramite le API Java di Jess.

Esempio: l'asserimento di fatti primitivi riguardanti una giocata appena effettuata

Quando un agente giocatore riceve dal mazziere la comunicazione di una nuova giocata avvenuta, sia essa la propria o quella di un altro giocatore, deve aggiornare la conoscenza del mondo circostante all'interno della *KB*. Sono infatti disponibili nuove informazioni riguardo le giocate effettuate finora e le carte in tavola (e di conseguenza le carte che non sono più nascoste nel "mazzo", ovvero nelle mani altrui); è inoltre possibile che sia stata giocata la carta chiamata: in questo caso si rendono noti i ruoli di tutti i giocatori, che quindi vanno aggiornati definitivamente.

```
1  /**
2   * Every time a card is played, we need to update our internal
3   * representation.
4   * - add the card onto the table
5   * - if it was us playing, remove the card from our "mano"
6   * - if it was others playing, remove the card from the hidden deck
7   * - if the card was the briscola, we now know the teams
8   *
9   * @param mano              # of "mani" played so far
10  * @param counter           # of "giocate" so far
11  * @param justPlayer        who has just played
12  * @param justCard          which card has just been played
13  * @throws JessException
14  */
15 public void addGiocata(int mano, int counter, Player justPlayer,
16                         Card justCard) throws JessException {
17
18     Fact z = new Fact("nuova-giocata", rete);
19     z.setSlotValue("player", new Value(justPlayer));
20     z.setSlotValue("card", new Value(justCard));
21     z.setSlotValue("rank", new Value(justCard.getRank()));
22     z.setSlotValue("suit", new Value(justCard.getSuit()));
23     rete.assertFact(z);
24
25     if (justCard.equals(briscolaCard)) {
26         Fact s = new Fact("socio", rete);
27         s.setSlotValue("player", new Value(justPlayer));
```

```
28          rete.assertFact(s);
29          for (Player p : players) {
30              if (!p.equals(justPlayer) && !p.equals(this.getPlayer())
                    ) {
31                  s = new Fact("villano", rete);
32                  s.setSlotValue("player", new Value(p));
33                  rete.assertFact(s);
34              }
35          }
36      }
37      if (justPlayer.equals(this.getPlayer())) {
38          Fact f = new Fact("in-mano", rete);
39          f.setSlotValue("card", new Value(justCard));
40          f.setSlotValue("rank", new Value(justCard.getRank()));
41          f.setSlotValue("suit", new Value(justCard.getSuit()));
42          rete.retract(f);
43      } else {
44          Fact f = new Fact("in-mazzo", rete);
45          f.setSlotValue("card", new Value(justCard));
46          f.setSlotValue("rank", new Value(justCard.getRank()));
47          f.setSlotValue("suit", new Value(justCard.getSuit()));
48          rete.retract(f);
49      }
50
51      Fact q = new Fact("in-tavolo", rete);
52      q.setSlotValue("card", new Value(justCard));
53      q.setSlotValue("player", new Value(justPlayer));
54      q.setSlotValue("rank", new Value(justCard.getRank()));
55      q.setSlotValue("suit", new Value(justCard.getSuit()));
56      rete.assertFact(q);
57
58      rete.run();
```

Listing 5.3: Funzione che viene richiamata ogni volta che si riceve la comunicazione di una nuova giocata (quindi anche la propria). Vengono aggiornati i fatti riguardanti la giocata stessa, la situazione in tavola e nel mazzo (ovvero nelle mani altrui) ed eventualmente i ruoli

I fatti "derivati", invece, sono asseriti dopo un processo di *reasoning* applicato ai fatti già contenuti nella *KB*. Il reasoning, come si può vedere dall'ultima linea del

Listato 5.3, viene attivato ad ogni nuova giocata ricevuta, così che sia possibile ottimizzare i tempi portandosi avanti sull'analisi della situazione prima che arrivi il proprio turno.

Esempio: fatti derivati asseriti applicando le strategie alle informazioni relative a una nuova giocata appena memorizzate nella KB

La regola `nuova-giocata` (Listato 5.4) viene eseguita dal reasoner successivamente all'esecuzione della funzione `addGiocata` (5.3) che ne asserisce i fatti contenuti nell'antecedente. Questa regola permette nello specifico di analizzare una giocata appena ricevuta e di classificarla in uno dei modi previsti dal sistema (liscio, taglio, strozzo...). Vengono inoltre modificati i fatti `seme-mano` e `prende` per aggiornare, rispettivamente, quale sia il nuovo seme regnante e chi si stia aggiudicando la mano.

```
1    ( defrule nuova-giocata "Ricevo una giocata: aggiorno la situa"
2     ?w <- (nuova-giocata (player ?p) (card ?c) (rank ?r) (suit ?s))
3     (prende (player ?prende-player) (card ?prende-card))
4     ?y <- (giocata-numero ?counter-giocata)
5     (seme-mano-fact (suit ?seme-mano))
6     (mano-numero ?mano-numero)
7  =>
8     (bind ?tipo "liscio")
9      ;;  Aggiorniamo chi prende
10    (if (= ?counter-giocata -1) then
11        (remove prende)
12        (assert (prende (player ?p) (card ?c) (suit (?c getSuit)) (
              rank (?c getRank))))
13        (remove seme-mano-fact)
14        (assert (seme-mano-fact (suit ?s)))
15        (bind ?seme-mano ?s)
16     else
17        (if (batte ?c ?prende-card ?seme-mano )  then
18            (remove prende)
19            (assert (prende (player ?p) (card ?c) (suit (?c getSuit)
                  ) (rank (?c getRank))))
20            (if (= ?seme-mano ?s) then
```

```
21                  (if (< (?r getValue) 10) then
22                      (bind ?tipo "strozzino")
23                  else
24                      (bind ?tipo "strozzo")
25                  )
26              else
27                  (bind ?tipo "taglio")
28              )
29          else
30              (if ( > (?r getValue) 9) then
31                  (bind ?tipo "carico")
32              else
33                  (if (> (?r getValue) 0) then
34                      (bind ?tipo "carichino")
35                  )
36              )
37          )
38      )
39      (bind ?new-counter-giocata (+ ?counter-giocata 1))
40      (retract ?w)
41      (retract ?y)
42      (assert (giocata-numero ?new-counter-giocata))
43      (assert (giocata (player ?p) (card ?c) (rank ?r) (suit ?s) (
            turno ?new-counter-giocata) (mano ?mano-numero) (tipo ?tipo))
            )
44  )
```

Listing 5.4: Regola che analizza una nuova giocata ricevuta, la classifica e aggiorna alcuni fatti temporanei

5.2.3 Le regole per le strategie di gioco

Le regole fin qui indicate, insieme ad altre dalla natura simile, permettono di creare un ambiente intuitivamente descrivibile tramite fatti e, di conseguenza, la relativamente semplice implementazione di svariate strategie di gioco anche da parte di non esperti di programmazione, complicata solo da alcune caratteristiche proprie del linguaggio Jess, come ad esempio il fatto che le regole vadano scritte usando la notazione prefissa degli operatori e il non molto intuitivo metodo per

selezionare fatti in base alle caratteristiche del contenuto degli slot.

Anche le strategie di gioco si dividono in due principali categorie:

- le strategie di *analisi*;
- le strategie di *decisione*.

Le prime sono quelle volte all'assegnamento, con un grado di fiducia puramente euristico, di ogni giocatore al suo ruolo. Servono a tentare di capire chi sia il socio in base alle giocate osservate.
Le strategie di decisione, invece, sono quelle che permettono di selezionare una carta tra quelle in mano per essere giocata.
Sebbene il framework costruito non vincoli in alcun modo i due tipi di strategie, nelle regole implementate durante la sperimentazione è parso ragionevole collegare le une alle altre.
È immediatamente evidente come una strategia di decisione possa basarsi sul risultato ottenuto da una di analisi: per esempio, se quest'ultima avesse assegnato, con un'incertezza molto bassa, un dato giocatore al ruolo di socio, la strategia di decisione eviterebbe di far giocare un carico quando tal giocatore si trovi ultimo di mano dopo di sè.
Allo stesso modo è possibile basare le strategie di analisi su quelle di decisione, assumendo che l'avversario segua strategie decisionali simili alle proprie. L'idea è quella di "rovesciare" le regole di decisione per costruire quelle di analisi, scambiando antecedente e conseguente.
Per chiarire l'intuizione, si assuma che il giocatore P abbia fra le sue regole una per cui: se il suo ruolo è *socio*, dopo di lui c'è il *chiamante* ultimo di mano, allora gioca un carico; P osservando un giocatore A penultimo di mano prima del *chiamante* giocare un carico, assumendo che A stia seguendo strategie simili alle proprie, potrà concludere con un certo grado di incertezza che il ruolo di A è quello del *socio*.
Tale approccio è limitato da due fattori: in primis non è detto che l'avversario segua effettivamene le stesse identiche strategie decisionali del giocatore "analista", per cui l'assunzione non è necessariamente corretta, soprattutto in casi in cui l'obiettivo dell'avversario sia quello di depistare il giocatore con un *bluff*.

Inoltre le strategie decisionali hanno nel proprio antecedente anche delle condizioni che riguardano le carte in mano, quindi un'informazione privata non disponibile a nessun altro giocatore.
Nonostante questi limiti ci è parso che un tentativo in questa direzione potesse costituire una buona approssimazione in mancanza di migliori strategie.

Esempio: il socio tiene il giaguaro ultimo; se tiene il giaguaro ultimo è il socio

Un esempio dell'approccio appena accennato è l'implementazione delle regole 5.5 e 5.6. Nel primo caso, del quale esistono diverse versioni in base alla precisa situazione ma di cui è qui presentata solo la più semplice, è formalizzata una regola piuttosto ovvia:

> *Ruolo del socio è quello di favorire il chiamante (per esempio prendendo subito dopo di lui) e di tenerlo il più possibile ultimo, specialmente nelle mani finali.*

```
1  ( defrule socio-tiene-giaguaro-ultimo
2      ?w <- (calcola-giocata)
3      (mio-ruolo socio)
4      (giaguaro (player ?g))
5      (mio-turno-numero ?n)
6      (turno (player ?player&:(= ?player ?g)) (posizione ?pos&:(= ?n (
           mod (+ ?pos 1) 5) ) ) )
7      (briscola (card ?b))
8      (posso-prendere (card ?c&:(<> ?c ?b)))
9  =>
10     (gioca ?c (- 100 (?c getValue)))
11     (assert (ora-di-giocare))
12 )
```

Listing 5.5: Se sono il socio e gioco subito dopo di lui, se posso prendere senza palesarmi, prendo per lasciarlo ultimo la mano successiva

Applicando il metodo dell'inversione delle regole, risulta la seguente 5.6.
Si noti che, come anticipato, la precedente regola 5.5 esiste in diverse versioni,

di modo che nelle mani finali la sua applicazione sia più probabile. Questo giustifica il diverso valore di incertezza che viene assegnato alla validità dell'analisi effettuata dalla regola 5.6: se si è nelle fasi finali della partita, sarà più probabile che il giudizio sia corretto.

```
1  ( defrule vs-socio-tiene-giaguaro-ultimo
2      (not(exists(socio (player ?player))))
3      (mano-numero ?mano-numero)
4      (giaguaro (player ?g))
5      (turno (player ?p&:(= ?p ?g)) (posizione ?pos-giaguaro))
6      (giocata (player ?soc) (tipo ?t&:( or ( = ?t "taglio") (or (= ?t
           "strozzino") (= ?t "strozzo") ) )  ))
7      (turno (player ?pl&:(= ?pl ?soc)) (posizione ?pos-soc&:( = ?
          pos-soc (mod (+ ?pos-giaguaro 1) 5) )))
8  =>
9      (if (> ?mano-numero 3) then
10         (bind ?new-sal 80)
11     else
12         (bind ?new-sal 40)
13     )
14     (aumenta-sal-socio ?soc ?new-sal)
15 )
```

Listing 5.6: Se qualcuno prende lasciando il giaguaro ultimo nella mano successiva, probabilmente è il socio, soprattutto se siamo durante la fase finale della partita

5.2.4 La decisione finale

In base alla situazione rappresentata nella *KB*, un diverso numero di regole può essere attivato contemporaneamente.
Questo può portare a una situazione in cui il reasoner si trovi ad avere dei fatti di tipo `da-giocarsi` (quelli che contengono le carte selezionate per la giocata) in conflitto fra loro.
Per risolvere questo tipo di conflitti si è preso ad esempio il sistema di *conflict resolution* interno adottato da JESS: (Friedman-Hill [2003]) ogni fatto `da-giocarsi` non contiene solo l'indicazione della carta che una regola ha ritenuto opportuno

giocare, bensì anche un valore euristico che indica la validità di tale carta. La funzione 5.7, al momento di giocare, sceglie fra le carte selezionate quella con migliore priorità per essere effettivamente giocata. In questo modo una regola "di difesa" che venga attivata quasi sempre e scelga una carta da giocarsi in modo da limitare al minimo le perdite, viene surclassata da una regola "di attacco" che permetta di prendere un numero decisivo di punti in tavola.
Tale metodo permette inoltre un immediato "tuning" delle regole che può avvenire non solo con l'aggiunta e le modifiche di queste, ma anche con l'aggiornamento dei loro valori di priorità, a volti detti *salience*.

```
1 ( deffunction carta-da-giocarsi ()
2     (bind ?it (run-query* da-giocarsi))
3     (if (?it next) then
4         (bind ?card (?it getObject c))
5         (bind ?sal (?it getObject n))
6         (while (?it next)
7             (if ( > (?it getObject n) ?sal) then
8                 (bind ?card (?it getObject c))
9                 (bind ?sal (?it getObject n))
10            )
11        )
12        return ?card
13    )
14 )
```

Listing 5.7: funzione che sceglie fra tutte quelle selezionate la carta da giocarsi

5.2.5 Riassumendo

Un *framework* di basso livello mette a disposizione strutture dati, funzioni e strategie generali che permettono una più semplice scrittura di nuove strategie, così come l'ampliamento delle esistenti, tramite il linguaggio Jess.
Le strategie attualmente implementate, raccolte da varie fonti, riflettono la conoscenza di giocatori esperti ed i loro comportamenti in alcune situazioni di carattere generale.
Gran parte dei comportamenti da seguirsi dipendono strettamente dal ruolo che si ha nella partita. Questa osservazione ha reso possibile lo sviluppo di regole

di *analisi*, di supporto a quelle di *decisone*, che permettono di assegnare con un margine di incertezza euristico un giocatore al suo ruolo presunto in base ai suoi comportamenti.
La possibilità di estendere l'insieme di strategie applicabili dal giocatore può dare adito all'attivazione di più regole contrastanti contemporaneamete e di conseguenza a problemi di *conflict resolution*; questi sono stati resi risolubili dando la possibilità di assegnare ad ogni strategia un valore euristico di priorità.

Implementazione degli agenti

Per l'implementazione degli agenti si è scelto di basarsi direttamente sulla piattaforma JADE sottostante, che fornisce una classe `Agent`. Per questo si è creato la classe `GeneralAgent`, che estende, appunto, `Agent` di JADE.
Questa classe è a sua volta estesa dalle classi `PlayerAgent` e `MazziereAgent` che implementano rispettivamente l'agente giocatore e l'agente mazziere.
In relazione a queste tre classi ne esistono altre tre che ne gestiscono l'interfaccia grafica.
Sono state inoltre ovviamente messe a disposizione degli agenti varie classi che rappresentino il mondo circostante: dalle carte, al tavolo, ai diversi giocatori.
Infine, gli agenti dispongono di classi utilizzate per mantenere la memoria della partita, in modo da poter usare queste informazioni per la fase di *reasoning* nel caso degli agenti giocatori, oppure per redigere il file di log nel caso dell'agente mazziere.

6.1 I *Behaviours*

Come in ogni sistema multi agente, il cuore dell'interazione fra gli attori del sistema è definito nei *task* assegnati agli agenti.
JADE mette a disposizione la classe `Behaviour` per implementare i *task*, così come alcune sue estensioni per tipi particolari di compiti (maggiori informazioni sono reperibili nell'appendice A).
Seguendo questa impostazione, nello sviluppo della nostra piattaforma si sono estese queste classi fornite da JADE per implementare i comportamenti specifici che gli agenti giocatore e mazziere dovessero tenere per interagire in modo da condurre una regolare partita di Briscola in 5.
Di seguito indichiamo i principali *Behaviours* implementati per i vari tipi di agente.

Classi usate da entrambi i tipi di agente:

- `GetChatMessage` è un behaviour ciclico che recupera e gestisce i messaggi di chat;
- `GetErrorMessage` come `GetChatMessage` ma volto ai messaggi di errore;
- `SendAndWait` spedisce messaggi di tipo bloccanti (estende `SendMessage`);
- `SendMessage` spedisce messaggi di tipo semplice.

Behaviour specifici per l'agente mazziere:

- `AskBriscola` richiede la briscola al vincitore dell'asta;
- `BeginGame` dà inizio alla partita;
- `DistributeHands` distribuisce le carte;
- `EndGame` gestisce le operazioni da eseguirsi al termine della partita;
- `GetGiocataComment` behaviour ciclico che recupera e gestisce i commenti alle giocate da parte dei giocatori umani;
- `ManageBid` gestisce la fase dell'asta;

- `OfferAChair` risponde alla richiesta di partecipazione da parte di un giocatore non ancora iscritto;
- `OpenTable` si iscrive al servizio pagine gialle e offre i propri servigi;
- `PlayGame` gestisce la fase di gioco;
- `WaitForSubscriptionConfirmation` richiamato dopo `OfferAChair`, attende limitatamente la conferma d'iscrizione da parte di un giocatore.

Behaviour specifici per l'agente giocatore:

- `BeginGame` gestisce la prima fase del gioco;
- `DeclareBriscola` il vincitore dell'asta risponde al mazziere dichiarando il seme di briscola;
- `PlayAuction` gestisce la fase d'asta;
- `PlayGame` gestisce la fase di gioco;
- `ReceiveHand` attende dal mazziere la propria mano di carte;
- `ReceiveScore` attende dal mazziere le informazioni sui punteggi;
- `Subscribe` iscrive il giocatore a un tavolo libero presso un mazziere;
- `WaitForBriscola` il villano attende dal mazziere la comunicazione del seme di briscola.

6.2 GeneralAgent

Questa classe fornisce i metodi di base degli agenti coinvolti nella piattaforma.
In concerto con la classe `GeneralGUI` che implementa i metodi base dell'interfaccia grafica degli agenti, `GeneralAgent` definisce i metodi per stampare il log delle attività, i messaggi di chat e altre informazioni utili quali l'elenco dei giocatori, dei punteggi ecc.
Dispone anche di diversi metodi per l'invio diretto di messaggi. Per una maggiore libertà nelle modalità di invio di messaggi, si è fatto largo ricorso all'*overloading* di questi metodi.
Oltre ad altri metodi di varia utilità comuni a tutti gli agenti, `GeneralAgent` fornisce anche un sistema per l'organizzazione dei *Behaviours*, ovvero dei *task* al livello di framework.
Purtroppo JADE non fornisce metodi per gestire i *Behaviours* ad alto livello; questo significa che anche solo per ottenere la lista dei *Behaviours* associati ad un agente è necessario agire a livello del sistema operativo.
Per evitare questa faticosa e poco elegante soluzione, si è deciso di estendere il metodo `addBehaviour` 6.2 della classe `Agent` di JADE nella classe `GeneralAgent`: in questo modo, aggiungendo un *Behaviour* a partire da un agente mazziere o giocatore, il *Behaviour* viene aggiunto ad una lista all'interno dell'istanza da cui è stato chiamato.

```
1 protected List<Behaviour> behaviours;
2 /*     ...        */
3     @Override
4     public void addBehaviour(Behaviour b) {
5         super.addBehaviour(b);
6         this.behaviours.add(b);
7     }
```

6.3 MazziereAgent

La classe `MazziereAgent` implementa evidentemente l'agente mazziere; come già detto, fa questo estendendo la classe `GeneralAgent`, dalla quale eredita i metodi base di un agente.

6.3.1 Avvio dell'agente: il metodo `setup`

La classe `Agent` di JADE fornisce un metodo `setup()` che viene chiamato quando la piattaforma avvia un nuovo agente. `MazziereAgent` estende questo metodo per adattarlo alle proprie esigenze.
In questa fase (6.1) l'agente mazziere analizza innanzitutto gli argomenti con cui è stato avviato:

1. nome dell'agente
2. scelta fra modalità grafica e testuale
3. indirizzo del file di log

inizializza le proprie strutture dati e il proprio reasoner (usato per codificare le regole del gioco).

```
1         private static final String rulesFile = "briscola/reasoner/
              mazziere.clp";
2
3         ...
4
5         //  SETTING UP THE RETE INSTANCE FOR JESS RULE PROCESSING
6         rete = new Rete();
7         try {
8             rete.batch(rulesFile);
9             rete.reset();
10        } catch (JessException ex) {
11            System.out.println("Impossibile aprire il file " +
                  rulesFile);
12            ex.printStackTrace();
13            takeDown();
```

```
14          }
```

Listing 6.1: stralcio del metodo setup() di MazziereAgent: inizializzazione strutture e reasoner

L'agente mazziere si registra al registro pubblico fornito da JADE per segnalare agli altri agenti della piattaforma la propria disponibilità a gestire una partita (6.2).

```
1           private DFAgentDescription dfd;
2           private ServiceDescription sd;
3
4           ...
5
6           //  REGISTER TO YELLOW PAGES
7           dfd = new DFAgentDescription();
8           dfd.setName(getAID());
9           sd = new ServiceDescription();
10          sd.setType(MAZZIERE);
11          sd.setName(name);
12          dfd.addServices(sd);
13          try {
14              DFService.register(this, dfd);
15          } catch (FIPAException fe) {
16              say("Errore durante la registrazione alle pagine gialle"
                    );
17              fe.printStackTrace();
18              takeDown();
19          }
```

Listing 6.2: stralcio del metodo setup() di MazziereAgent: registrazione al servizio pagine gialle

Infine, richiama il proprio metodo `startNewGame()` per dare inizio alla partita, avviando i *Behaviours* che si occupano dell'apertura del tavolo e della ricezione degli eventuali commenti alle giocate. Inizializza inoltre un nuovo "registro della partita" che costituisce un vero e proprio log delle attività svoltesi, che è implementato nella classe `GameMemory`.

6.3.2 Chiusura dell'agente: il metodo `takeDown`

Un altro metodo della classe `Agent` di JADE che viene qui esteso è il metodo chiamato alla chiusura dell'agente, ovvero `takeDown()`. In questa fase (6.3) l'agente mazziere si preoccupa di disallocare le proprie strutture e rimuovere il proprio nome dal registro pubblico.

```
1     @Override
2     protected void takeDown() {
3
4         say("Felice di aver giocato con voi. Addio!");
5         dfd.removeServices(sd);
6
7         if (graphic) {
8             gui.dispose();
9         }
10        if (writeCSV) {
11            try {
12                csvWriter.close();
13            } catch (IOException ex) {
14                ex.printStackTrace();
15            }
16        }
17    }
```

Listing 6.3: il metodo takeDown di MazziereAgent

Gli altri metodi implementati in questa classe sono principalmente metodi di utilità o metodi per interagire con il registro in memoria della partita.

6.4 L'agente PlayerAgent

Anche la classe `PlayerAgent` è, come si è anticipato, un'estensione della più generale `GeneralAgent`; essa implementa l'agente giocatore, ridefinendo i principali metodi della classe `Agent` fornita da JADE e fornendone ulteriori di varia utilità.

6.4.1 Avvio dell'agente: il metodo `setup`

All'avvio l'agente giocatore riceve da linea di comando i seguenti parametri:

- un *nome* che verrà utilizzato all'interno della piattaforma di gioco
- un'indicazione sulla *strategia* da seguire: questa può essere *manuale* nel caso di un giocatore umano, *random* per un gioco pseudo-casuale o il percorso ad un *file* che contenga le regole per il sistema esperto
- un flag *visible* che indichi se avviare o meno l'interfaccia grafica

Una volta impostati i succitati parametri, l'agente giocatore aggiunge alla propria lista di *behaviours* quello per l'iscrizione alla piattaforma e la ricerca di un tavolo, che dà inizio al flusso sequenziale dei *task* che regolano lo svolgimento della partita.

6.4.2 Alcuni metodi che s'interfacciano con il reasoner

Come precedentemente detto, l'agente giocatore si interfaccia direttamente con il sistema esperto attraverso i propri metodi; questo è possibile grazie alle *API* JAVA fornite da JESS, che permettono di asserire e modificare fatti nella *KB* così come eseguire funzioni.
Nel Listato 6.4 è possibile vedere come questo avvenga all'inizio di ogni mano: una volta che l'agente mazziere ha comunicato a tutti i giocatori il resoconto della mano precedente e l'imminente inizio di una nuova, questi eseguono il proprio metodo `initMano` che calcola il nuovo ordine da assegnare ai giocatori e poi asserisce le relative informazioni come fatti nella *KB*.

```
1    public void initMano(int mano, Player next) throws JessException
         {
```

```
2
3           // comunichiamo al reasoner l'inizializzazione di una nuova
                mano
4           Value v = new Funcall("init-mano", rete).arg(
5               new Value(mano, RU.INTEGER)).execute(rete.
                    getGlobalContext());
6
7           // assegnamento dell'ordine dei turni ai giocatori
8           int firstIndex = -1;
9
10          for (int i = 0; i < players.size(); ++i) {
11              if (players.get(i).equals(next)) {
12                  firstIndex = i;
13                  break;
14              }
15          }
16
17          for (int i = 0; i < players.size(); ++i) {
18              Fact f = new Fact("turno", rete);
19              Player p = players.get((i + firstIndex) % 5);
20              f.setSlotValue("player", new Value(p));
21              f.setSlotValue("posizione", new Value(i, RU.INTEGER));
22              rete.assertFact(f);
23              if (p.equals(this.getPlayer())) {
24                  rete.assertString("(mio-turno-numero " + i + ")");
25              }
26          }
27
28          rete.run();
29          gui().initMano(mano, next);
30      }
```

Listing 6.4: il metodo `initMano` di PlayerAgent

La stessa cosa succede in risposta alla ricezione dell'informazione (da parte del mazziere) di una nuova giocata: in questo caso l'agente giocatore dovrà innanzitutto aggiornare le immediate informazioni che descrivono, all'interno della *KB*, le carte in tavolo e quelle ancora coperte; inoltre, se la carta giocata è la *chiamata*, vengono aggiornate anche le informazioni riguardanti i ruoli dei giocatori.
Inoltre viene aggiunto alla *KB* anche un fatto complesso che riassuma la giocata,

contenente le informazioni sul giocatore autore della stessa e la carta che è stata giocata; queste informazioni verranno poi integrate all'interno del reasoner dopo che questo avrà attivato le proprie regole di analisi e quindi classificato la giocata come appartenente a uno dei predefiniti gruppi (*taglio*, *carico*, *liscio*...).
Parte del codice della funzione si trova nel Listato 5.3.
Infine, anche l'oggetto `PlayerAgent` fa largo uso di classi di supporto utili soprattutto a memorizzare gli eventi della partita, sia per quanto riguarda la fase di gioco che per quella di asta.

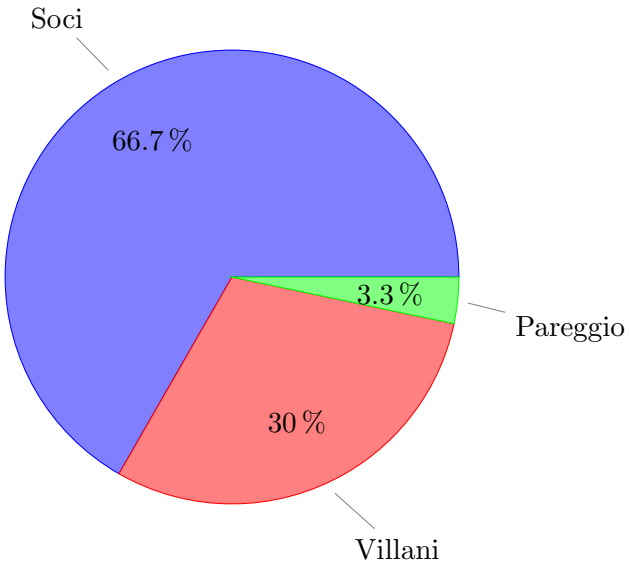
Sperimentazione

Una volta in possesso di un framework generale che permetta l'implementazione di strategie arbitrarie, è auspicabile poter disporre di un metodo di valutazione della bontà di tali strategie.
Le difficoltà riscontrate nell'affrontare il problema della decisione della mossa da effettuarsi in maniera puramente algoritmica si ripropongono nel momento in cui si voglia applicare lo stesso approccio algoritmico ad un tentativo di valutazione della bontà delle singole mosse.
Per questa ragione si è pensato di provare a valutare un giocatore empiricamente, in base ai successi conseguiti — espressi in termini di percentuali di vittorie sul totale delle partite — confrontandoli con quelli di un giocatore che abbia una strategia puramente casuale.
Data la struttura dell'agente giocatore e la fissità della strategia con cui gestisce la fase di asta (fase decisiva nella formazione delle squadre), non è possibile associare a priori la natura (casuale o strategica) di un giocatore al suo ruolo. È stato quindi necessario in alcuni casi eseguire un gran numero di prove per ottenerne una quantità accettabile che rispondesse all'interesse sperimentale.

7.1 Random vs Random

Per ottenere un indice di confronto da cui partire si sono innanzitutto condotte 30 partite con cinque giocatori a strategia completamente casuale; i risultati ottenuti sono espressi nella Figura 7.1.
Si nota immediatamente come in queste condizioni la squadra del chiamante sia decisamente avvantaggiata; volendo provare a dare un'interpretazione di tale fatto si può avanzare l'ipotesi per cui, mentre la squadra del chiamante sfrutta appieno

il vantaggio derivato dalla fase dell'asta (che viene sempre e comunque effettuata in maniera oculata anche nel caso dei giocatori random), costituito dall'avere in mano le carte migliori, la squadra dei villani non ricorre al vantaggio derivato dalla propria superiorità numerica che potrebbe essere invece sfruttato facendo gioco di squadra.
Va inoltre considerato che anche nella tradizione è convinzione comune che una buona conduzione della fase dell'asta porti alla squadra del chiamante un considerevole vantaggio rispetto alla squadra avversaria.



(a) Percentuale di vittorie conseguite per squadra

chiamante + socio	villani	pareggi	totale prove svolte
20	9	1	30

(b) Numero assoluto di vittorie conseguite per squadra su un totale di 30 partite

Figura 7.1: Random vs Random: esiti delle partite in base alla squadra di appartenenza

7.2 Un giocatore a strategia vs Random

Nel secondo esperimento condotto si sono giocate 50 partite nelle quali un solo giocatore utilizzava il sistema a strategia mentre tutti gli avversari giocavano ca-

sualmente.
I risultati sono espressi nelle Figure 7.2 e 7.3.
Mentre l'incremento di vittorie da parte della squadra del chiamante [Fig. 7.3(a)] era abbastanza atteso, stupisce a prima vista il netto peggioramento della squadra dei compari [Fig. 7.3(b)], che risulta avere più successo se composta da giocatori del tutto casuali rispetto ad averne uno che segua delle strategie.
Questo fatto è però facilmente spiegabile tramite l'osservazione per cui le strategie implementate assumono la complicità dei propri compagni di squadra; per esempio, un villano che si trovi a giocare per terzo, prima dei suoi compagni, molto probabilmente giocherà il carico di maggior valore che possiede in mano, nella speranza che uno dei propri compagni abbia la possibilità di prenderlo. Se però questi compagni giocano casualmente è facile che lascino i punti alla squadra avversaria.

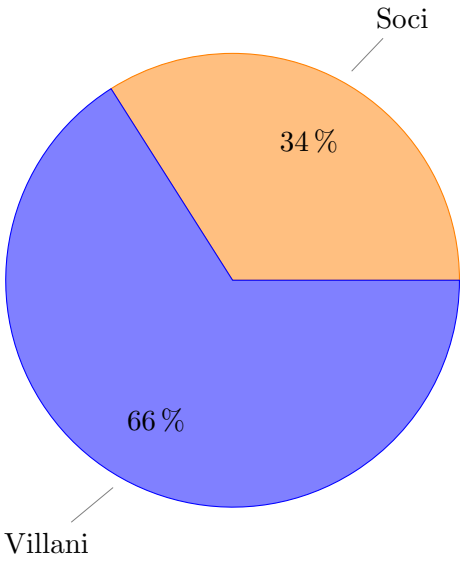
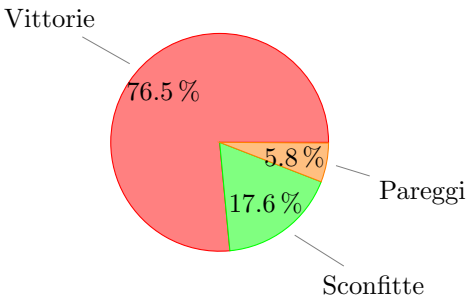
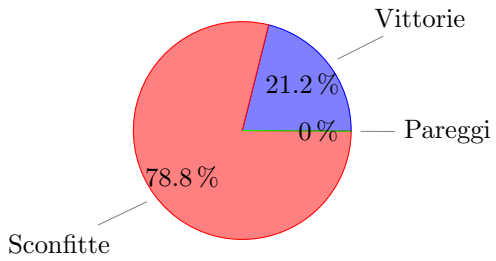


Figura 7.2: Suddivisione delle partite in base alla squadra di appartenenza del giocatore a strategie



(a) Esiti delle prove in cui il giocatore a strategie era fra i soci



(b) Esiti delle prove in cui il giocatore a strategie era fra i villani

	soci	villani
vittorie	13	7
sconfitte	3	26
pareggi	1	0
totale	17	33

(c) Esiti delle partite suddivise in base al ruolo del giocatore a strategia

Figura 7.3: Un giocatore a strategia vs Random: esiti delle partite in base alla squadra di appartenenza

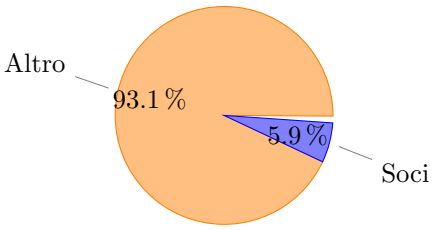
7.3 Una squadra a strategia vs Random

Si sono poi condotte 170 partite con due giocatori a strategia, ottenendone 10 in cui entrambi i giocatori fossero nella squadra dei soci e 115 con tre giocatori a strategia ottenendone 10 in cui questi fossero tutti tra i villani. In questo modo è stato possibile valutare le prestazioni di una squadra a strategie (sia essa dei villani o dei soci) contro una random.

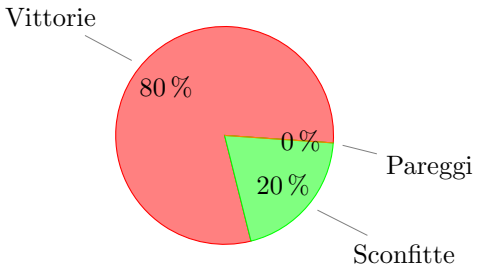
Confrontando i risultati di questo esperimento, illustrati nelle Figure 7.4(b) e 7.5(b), con quelli ottenuti con squadre a giocatori casuali, [Fig. 7.6(a)], è immediato notare come, per entrambe le squadre, ci sia stato un notevole vantaggio derivato dall'adozione (da parte dell'intera squadra) delle strategie basi finora implementate.

Questo ha infatti portato a un incremento del 13.3 % di vittorie sul numero assoluto di partite per la squadra del chiamante e del 30 % per la squadra dei villani, che equivale al doppio della percentuale di partite vinte dalla stessa squadra

formata da giocatori a strategia casuale.



(a) Frazione delle partite "utili" (in cui i due giocatori a strategie erano soci) sul totale

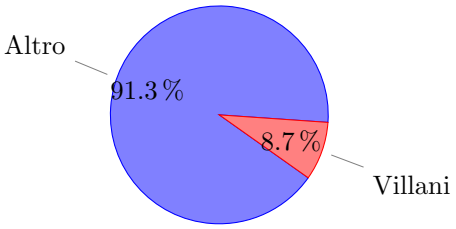


(b) Esiti delle sole prove "utili"

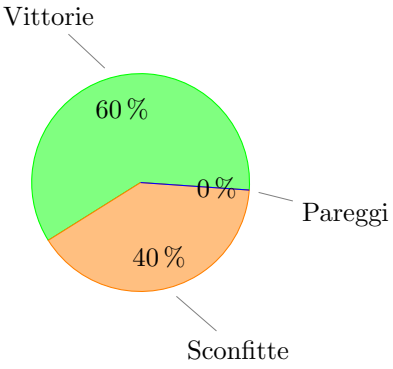
totale	utili	vittorie tra le utili
170	10	8

(c) Esiti delle partite in valore assoluto. Con *utili* si indicano quelle partite in cui i due giocatori a strategia erano soci

Figura 7.4: Esiti delle partite svolte con due giocatori a strategia



(a) Frazione delle partite "utili" (in cui i tre giocatori a strategie erano villani) sul totale



(b) Esiti delle sole prove "utili"

totale	utili	vittorie tra le utili
115	10	6

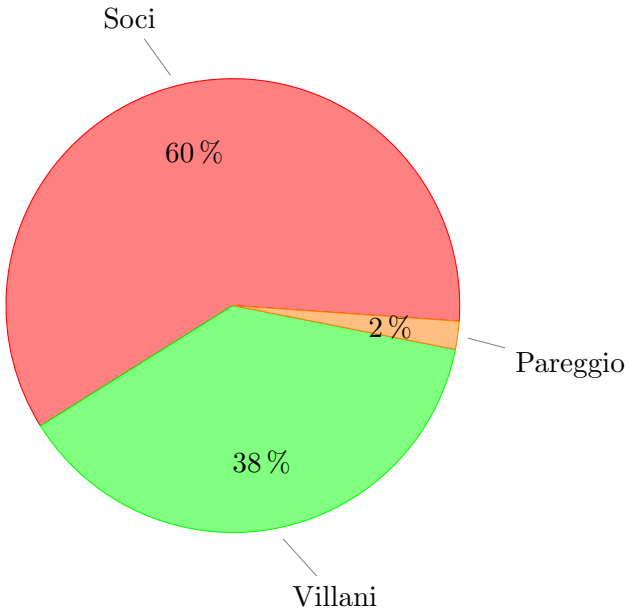
(c) Esiti delle partite in valore assoluto. Con *utili* si indicano quelle partite in cui i due giocatori a strategia erano villani

Figura 7.5: Esiti delle partite svolte con tre giocatori a strategia

7.4 Strategia vs Strategia

Infine si sono condotte 50 prove con con tutti e cinque i giocatori a strategia, i cui esiti sono raccolti nella Figura 7.6.
Usando questa configurazione si è tornati a una situazione molto simile a quella iniziale di random vs random (Fig. 7.1), con un lieve incremento di vittorie da parte dei villani a discapito dei soci.
Questa variazione degli equilibri, oltre a poter dipendere da una maggior bontà delle strategie implementate specificamente per i villani rispetto a quelle per i soci, potrebbe essere anche giustificabile tramite le stesse ipotesi avanzate nel tentativo di spiegare la disparità nei risultati random vs random: mentre il vantaggio connaturato nel ruolo di soci rimane, spostando favorevolmente verso questi le percentuali di vittoria, si perde l'ulteriore svantaggio che avevano i soci nel caso random vs random. Allora infatti essi erano impossibilitati a contrapporre al vantaggio dei soci derivato dalle migliori carte possedute, la propria superiorità numerica; potendo invece qui far gioco di squadra tale disparità, se non sparisce,

si riduce sensibilmente.



(a) Percentuale di vittorie conseguite per squadra

chiamante + socio	villani	pareggi	totale prove svolte
30	19	1	50

(b) Numero assoluto di vittorie conseguite per squadra su un totale di 50 partite

Figura 7.6: Strategia vs Strategia: esiti delle partite in base alla squadra di appartenenza

7.5 Interpretazione dei dati sperimentali

Nonostante le difficoltà riscontrate nel generare un test-set significativo, l'incremento di prestazioni ottenuto dai giocatori a strategia rispetto a quelli random è chiaramente visibile. Considerando inoltre la natura piuttosto basilare delle strategie implementate, i dati raccolti sono molto incoraggianti. Queste osservazioni ci fanno concludere che il sistema costruito sia potenzialmente valido e che aumentando l'ampiezza dell'insieme delle strategie, anche tramite l'uso dell'interfaccia per l'utente esperto, sia possibile ottenere giocatori sempre più performanti in questo tipo di valutazione.

Conclusioni

Lo scopo di questo lavoro è stato quello di creare una piattaforma per il gioco della Briscola in 5 sulla quale potessero confrontarsi giocatori sia umani che virtuali.
Per raggiungere tale scopo si è innanzitutto valutato il metodo migliore per la realizzazione di un'intelligenza artificiale ad uso del giocatore virtuale; dopo aver passato in rassegna le principali tipologie di soluzione applicate ai giochi in letteratura, si è optato per l'uso di un sistema esperto.
La ragione principale di questa scelta, che costituisce anche la maggiore difficoltà nel gioco della Briscola in 5, risiede nel fatto che si tratta di gioco non classificabile in assoluto nè come competitivo nè come cooperativo.
Si è quindi sviluppato, tramite l'ausilio del *rule engine* Jess, un framework per l'implementazione e l'estensione di strategie, insieme con alcune strategie basilari raccolte da diverse fonti. Si sono previste due principali categorie di strategie: quelle di analisi, che permettono di capire empiricamente la formazione delle squadre prima che queste siano palesi, e quelle di gioco che selezionano la carta migliore da giocarsi ad ogni mano.
La piattaforma di gioco vera è propria, realizzata a partire dal framework Jade, è stata progettata con l'intento di renderla adatta all'uso da parte di giocatori esperti, in modo che questi, giocando, possano annotare e consigliare nuove strategie. Tale piattaforma permette inoltre lo svolgimento di partite in rete, pur mantenendo un agente "mazziere" che consente la centralizzazione dello svolgimento della partita e di conseguenza del log degli avvenimenti.
Proprio a partire da questa attività di log è stato possibile valutare empiricamente l'efficiacia delle seppur basilari strategie implementate: mettendo a confronto le percentuali di partite vinte da un tipo di squadra in base alla tipologia di strategie

adottate dai suoi giocatori, si è reso evidente l'incremento di successi ottenuti dai giocatori a strategia rispetto a quelli casuali, incoraggiando l'ulteriore sviluppo del lavoro in questo senso.

8.1 Sviluppi futuri

Nell'ottica di un miglioramento del lavoro fin qui svolto, potrebbero innanzitutto essere prese in considerazione ulteriori soluzioni per la realizzazione dell'intelligenza artificiale del giocatore a strategie. Sarebbe infatti interessante valutare l'applicabilità di metodi di *machine learning* alla raccolta di strategie.
Inoltre, la stessa strategia del sistema esperto adottata potrebbe essere integrata con altri strumenti: in primis, si potrebbe pensare di limitare l'uso dell'expert system alla sola fase della partita in cui la formazione delle squadre non è certa, integrando poi nella seconda parte del gioco un metodo di soluzione diverso, come una ricerca nello spazio degli stati. Si potrebbe poi pensare di integrare alle strategie di analisi un sistema di *reputation* che permetta di assegnare con maggiore precisione le probabilità che un dato giocatore appartenga ad una certa squadra.
Sarebbe infine auspicabile esplorare altri sistemi e criteri di valutazione della validità di una strategia; tale valutazione potrebbe essere fatta sia in maniera formale, a partire da un osservatore onniscente che possa calcolare in maniera rigorosa la bontà di una mossa, sia in maniera più empirica, mettendo direttamente a confronto l'intelligenza artificiale con degli avversari umani.

Appendice A: Jade

JADE, acronimo di *Java Agent DEvelopment Framework*, è un framework sviluppato interamente in JAVA atto a essere utilizzato come *middle-ware* nello sviluppo di applicazioni multiagente.
È implementato rispettando le specifiche *FIPA*, acronimo di *Foundation for Intelligent Physical Agents*, un'organizzazione internazionale per lo sviluppo di standard per l'implementazione e la comunicazione di sistemi multi-agente.
JADE è free software, distribuito sotto la licenza LGPL e sviluppato da TELECOM ITALIA. Fornisce un sistema di astrazione per gli agenti, un modello per la composizione ed esecuzione di *task* e un servizio di *pagine gialle*. Le applicazioni multiagente sviluppate con JADE possono essere distribuite fra diversi host connessi in rete. Su ogni macchina che esegue la piattaforma viene attivata una sola applicazione JAVA all'interno della JVM che funge da "contenitore" di agenti fornendo anche un ambiente completo per l'esecuzione concorrente di più agenti contemporaneamente.
L'architettura di comunicazione offre un sistema di messaggistica flessibile ed efficiente: JADE crea e gestisce una coda privata di messaggi ACL per ogni agente; gli agenti possono accedere alla propria coda in diversi modi.
Gli agenti sono implementati come un singolo thread per agente, ma con la possibilità di associare ad ognuno più *behaviour*, ciascuno dei quali viene eseguito in un thread dedicato, grazie all'ambiente multi-thread offerto da JAVA.
La piattaforma fornisce anche un'interfaccia grafica (GUI) per la gestione remota degli agenti, il loro controllo e monitoraggio.

Appendice B: Jess

Jess è insieme un *rule egine* (*motore inferenziale*) e un ambiente di scripting, scritti interamente in Oracle Java da Ernest Friedman-Hill per i Sandia National Laboratories di Livermore, California.
Con Jess è possibile creare del software Java che sia capace di "ragionare" secondo regole scritte in maniera dichiarativa.
Pur essendo un sistema leggero, Jess risulta essere uno dei più veloci motori referenziali in circolazione. Ulteriore potenzialità è l'accesso completo all'insieme di API Java a partire dal linguaggio di scripting.
Per la selezione delle regole da attivarsi, il motore inferenziale utilizza una versione modificata dell'algoritmo Rete. Quest'ultimo è un meccanismo molto efficiente che permette di risolvere il complesso problema del matching molti-a-molti (si veda, a riguardo, Charles L. Forgy [1982]).
Alcuni punti di forza di Jess rispetto ad altri sistemi dello stesso tipo includono la possibilità di effettuare il *backward chaining* delle regole, di eseguire query dirette alla memoria di lavoro e di ragionare e manipolare direttamente oggetti Java. È disponibile liberamente per uso accademico.

Bibliografia

Pocket Fritz 3 Wins 'Human' Chess Tourney. http://www.chess.com/news/pocket-fritz-3-wins-human-chess-tourney, 2008. [Online; accessed 27-May-2015]. 12

Andrea Villa. Algoritmi di Ricerca ad Albero Monte Carlo Applicati all'Intelligenza Artificiale nel Gioco della Briscola a Cinque. 2013. 30, 43

Marina Bono. *Giochi di carte*. KeyBook, 2010. 5

Charles L. Forgy. Rete: A Fast Algorithm for the Many Pattern/ Many Object Pattern Match Problem. 1982. 77

The Economist. Finding equations to explain the world. http://www.economist.com/blogs/freeexchange/2015/05/remembering-john-nash, 2015. [Online; accessed 8-Jun-2015]. 21

Ernest Friedman-Hill. Jess 7. http://jessrules.com, 2008. [Online; accessed 25-May-2015]. 2

François Dominic Laramée. Chess Programming Part IV: Basic Search. http://www.gamedev.net/page/resources/_/technical/artificial-intelligence/chess-programming-part-iv-basic-search-r1171, 2000. [Online; accessed 26-May-2015]. 16

Ernest Friedman-Hill. *Jess in Action.* Manning, 2003. 27, 45, 54

Donald; Lenat Douglas Hayes-Roth, Frederick; Waterman. *Building Expert Systems.* Addison-Wesley, 1983. 28, 29

Ian Frank; David Basin. Search in games with incomplete information: a case study using Bridge card play. 1998. 12, 22, 26

Jack Copeland. A Brief History of Computing. http://www.alanturing.net/turing_archive/pages/ReferenceArticles/BriefHistofComp.html, 2000. [Online; accessed 26-May-2015]. 11

G. Farina; A. Lamberto. *Enciclopedia delle carte. La teoria e la pratica di oltre 1000 giochi.* L'Airone, 2004. 9

Marco Li Calzi. Un eponimo ricorrente: Nash e la teoria dei giochi. 2002. 22

Pavel Cejnar. Computing ϵ-Optimal Strategies in Bridge and Other Games of Sequential Outcome. 2008. 12, 26

Eric Rasmusen. Games and Information: An Introduction to Game Theory. https://books.google.it/books?id=5XEMuJwnBmUC&printsec=fnd&pg=PR5&hl=it, 2006. [Online; accessed 8-Jun-2015]. 21

R.D. Lute; H. Raiffa. Games and Decisions-Introduction and Critical Survey. 1957. 26

Roger B. Myerson. Game Theory: Analysis of Conflict. 1991. 21

Stuart Russell and Peter Norvig. *Artificial Intelligence, A Modern Approach*. Pearson, 2010. 11, 13, 16, 19, 20

Andrea Angiolino; Andrea Sidoti. *Dizionario dei giochi*. Zanichelli, 2010. 6

Telecom Italia Lab. JADE (Java Agent DEvelopment Framework). http://jade.tilab.com, 2001. [Online; accessed 25-May-2015]. 2