UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA DELL'INFORMAZIONE

Implementazione della "briscola a due" in linguaggio C

Relatore: **Prof. Giunta Antonio**

Laureando/a: **Gasparini Marco**

ANNO ACCADEMICO 2021 – 2022

Data di laurea 21 / 9 / 2022

Indice

Capitolo 1

Il gioco della Briscola

Quello della *Briscola* è uno dei giochi di carte tra i più conosciuti nella tradizione italiana. Sebbene non siano chiare le sue origini, una delle ipotesi più accreditate è che derivi dal gioco francese della *Brusquembille*, che potrebbe a sua volta essere ispirato ad alcuni giochi nordeuropei. Sembra che fosse diffuso nel nostro Paese già nella prima metà dell'Ottocento: ne è prova un sonetto di Gioacchino Belli del 1847 in cui viene descritto lo svolgimento di una partita.
In questo elaborato si prende in considerazione la versione con due giocatori, facendo riferimento al mazzo classico italiano.

1.1 Regole e scopo del gioco

Il mazzo si compone di quaranta carte divise in quattro semi, che nelle carte nord-italiane sono *Denari, Coppe, Spade* e *Bastoni*.
Il mazzo viene mescolato; dopo, l'ultima sua carta viene scoperta e indica quale sarà la briscola durante la partita.
L'ordine gerarchico delle carte di ogni seme e i loro punti sono i seguenti:

- *Asso* – 11 punti;
- *Tre* – 10 punti;
- *Re* – 4 punti;
- *Cavallo* – 3 punti;
- *Fante* – 2 punti;
- le altre carte seguono in ordine decrescente di valore, e portano tutte 0 punti.

Inoltre, le briscole battono le carte di tutti gli altri semi.

Il mazziere distribuisce tre carte a sé e all'altro giocatore, che inizia la partita giocando una carta. Il mazziere risponde, e il vincitore della mano, che si aggiudica le due carte, viene decretato seguendo la scala gerarchica precedente; ad essa va aggiunto che, in caso di risposta in seconda mano con una carta non di briscola e di seme diverso rispetto alla prima giocata, il turno è vinto dal primo di mano.
In seguito, il mazziere consegna una carta a ciascun giocatore, partendo da chi ha vinto il turno precedente, che sarà anche il primo di mano nel successivo.
La partita prosegue per un totale di venti manche, le cui ultime tre non assegneranno nuove carte poiché il mazzo sarà esaurito.

Al termine ogni giocatore conta i propri punti e vince chi ne ha accumulati di più: poiché il loro totale è pari (120), è possibile anche il risultato di parità.

Capitolo 2

Finalità di questo lavoro

Lo scopo principale di questo lavoro tesi è quello di creare un eseguibile che permetta all'utente di giocare una partita di "Briscola a due" contro il computer.

Prima di poter effettivamente implementare questo progetto in linguaggio C, si rende necessario elaborare una strategia chiara che il giocatore virtuale possa applicare per effettuare la scelta della carta più adatta da giocare in ogni situazione possibile.
Un obiettivo successivo è l'offerta di diversi livelli di difficoltà: esso si traduce nella necessità di rendere le scelte del computer più o meno "imperfette" rispetto alla giocata migliore, secondo la strategia stabilita.

Oltre che elaborare un algoritmo adeguato e tradurlo in codice C, si rende necessario predisporre una fase di test che permetta di scegliere dei valori per i parametri utilizzati che consentano di raggiungere al meglio gli scopi prefissati.

Questo elaborato illustra la soluzione proposta, e contiene in appendice il codice prodotto.

Capitolo 3

Struttura del progetto

L'implementazione in linguaggio C del progetto qui presentato si articola in cinque file sorgenti distinti.
Tutte le funzioni che si occupano della gestione della partita sono dichiarate nel file **BriscolaProt.h** e implementate nel file **BriscolaImplem.c**.
Il file **BriscolaTSCost.h** contiene invece le definizioni dei tipi strutturati utilizzati nel progetto e di alcune costanti utili per rendere più leggibili i sorgenti.
Briscola.c contiene invece la funzione `main` che permette di produrre l'eseguibile **Briscola.exe**, cioè il vero e proprio gioco usufruibile dall'utente.
Anche l'ultimo sorgente, ovvero **BriscolaParTest.c**, contiene la funzione `main`: da esso si genera infatti un omonimo file eseguibile, il cui scopo è ottimizzare i parametri utilizzati nelle varie funzioni, come dettagliato in seguito nel capitolo dedicato.

3.1 Istruzioni per ottenere gli eseguibili

Lo standard di riferimento per la sintassi del codice e per la sua compilazione è l'*ANSI C* e il compilatore, usato in linea di comando, è quello dell'ambiente *GCC*.
Ognuno dei file elencati in precedenza deve trovarsi nella stessa cartella; è necessario aprire l'interprete **cmd.exe** in tale directory.
Per prima cosa, si usa il comando

```
gcc -ansi -pedantic -Wall -Werror -c NomeFile.c
```

per precompilare, compilare ed assemblare singolarmente ciascuno dei tre file con estensione **.c** indicati sopra.
Le opzioni utilizzate restituiscono una compilazione pedante secondo il già citato standard *ANSI C*, segnalando ogni warning ed equiparandolo ad un errore.
In seguito, si usano i comandi

```
gcc -o Briscola.exe Briscola.o BriscolaImplem.o
```

```
gcc -o BriscolaParTest.exe BriscolaParTest.o BriscolaIm-
plem.o
```

per procedere con le operazioni di linking necessarie per produrre gli eseguibili già menzionati, a partire dai file oggetto creati in precedenza.
Un modo per eseguire compattamente questa procedura è l'uso dei file batch; per semplicità, durante lo sviluppo di questo progetto ne è stato utilizzato uno che eseguiva in sequenza le istruzioni dettagliate in questo paragrafo.

3.2 Funzionamento di Briscola.exe

Il file **Briscola.exe** prodotto come illustrato in precedenza viene eseguito da riga di comando, senza fornire alcun argomento. Infatti, il livello di difficoltà viene richiesto esplicitamente in seguito, in fase di inizializzazione della partita: deve essere inserito un carattere tra *f, m* e *d.*

Durante lo svolgimento della partita, tutte le informazioni necessarie sono stampate come output testuale.
In particolare, l'utente sceglie quale carta giocare prendendo visione tra quelle disponibili, e selezionandone una tramite l'inserimento della lettera (*a*, *b* o *c*) corrispondente. Il programma implementa un controllo dell'input per evitare errori dovuti all'inserimento di caratteri non pertinenti.
Al termine vengono visualizzati i punteggi di entrambi i giocatori ed il vincitore.

Capitolo 4

Strategia di gioco

Prima di illustrare le scelte progettuali strettamente legate al linguaggio di programmazione scelto, in questa sezione si chiariscono brevemente le scelte logiche che permettono l'effettiva implementazione del giocatore virtuale.
La strategia sviluppata si basa su tre componenti chiave.

4.1 Peso di una carta

Un obiettivo importante di questo lavoro era quello di dotare il giocatore virtuale di una capacità di memorizzazione più o meno alta, per simulare il giocatore umano più o meno abile e realizzare i diversi livelli di difficoltà.
Tale capacità si può misurare come la probabilità di ricordare una singola carta uscita, e questa eventuale memorizzazione viene ripetuta ad ogni manche e ad ogni turno.

Le informazioni così ottenute vengono quindi sfruttate nel calcolo del *peso* delle carte, una proprietà delle stesse che vuole misurare quanto esse siano sacrificabili o invece utili in un dato momento della partita.
Più precisamente, in esso si sommano:

- una componente proporzionale ai punti della carta considerata;
- un bonus solo se la carta è una briscola;
- il numero di carte contro cui essa potrebbe essere giocata in risposta e vincerebbe la manche, tra quelle che il giocatore virtuale non ricorda essere già uscite.

Mentre le prime due componenti assumono un valore noto non appena sia stata determinata la briscola, la terza è "dinamica", ovvero varia (non crescendo) mentre si svolge una partita, e si calcola quindi contando, per la carta considerata:

- il numero di carte non memorizzate del suo stesso seme che siano perdenti contro di essa;
- solo se essa è una briscola, anche tutte le carte non memorizzate degli altri semi, che sarebbero perdenti in uno scontro.

4.2 Strategia del primo di mano

Quando il giocatore virtuale risulta essere il primo di mano, procede giocando la carta di peso minimo tra quelle che possiede. Questa scelta simula di certo un giocatore umano "prudente", ma è resa precisa grazie a questa minimizzazione, che restituisce, come illustrato, la carta più sacrificabile in un preciso momento della partita.

4.3 Strategia del secondo di mano

Questa strategia è più articolata. Il giocatore virtuale, data la carta *c* giocata dal primo di mano, procede come segue:

- se *c* non è una briscola, gioca la maggior carta del suo stesso seme e che la batta, se esiste tra quelle che possiede;
- se non la trova o se *c* è una briscola, gioca la sua minor briscola, ma solo se *c* vale un minimo punteggio predeterminato;
- se a questo punto non è ancora stata individuata una carta, gioca anche qui la carta di peso minimo tra quelle che ha.

Occorrono un paio di precisazioni. Quando si parla di carta maggiore o minore, si considera la gerarchia tra carte dello stesso seme, che si costituisce per punti decrescenti, e per valori decrescenti tra le carte con zero punti.
Inoltre, la soglia predeterminata a cui si fa riferimento sarà discussa nel dettaglio nel capitolo dedicato all'ottimizzazione dei parametri.

Capitolo 5

Scelte implementative

Questo capitolo analizza le scelte logiche e sintattiche necessarie a implementare il progetto in linguaggio C. Ulteriori informazioni sulle funzioni e strutture dati sono contenute nella documentazione delle stesse all'interno del codice in appendice, in particolare nel sorgente **BriscolaImplem.c** e nel file header **BriscolaProt.h.**

5.1 Strutture dati

Sono stati definiti tre differenti tipi strutturati: segue il loro elenco secondo un ordine bottom-up in termini di incapsulamento.

5.1.1 Carta

Tale struttura contiene le informazioni relative a una carta del mazzo.
Il campo `seme` assume i valori *0, 1, 2, 3* per indicare rispettivamente *Denari, Coppe, Spade* e *Bastoni*; è memorizzato in una variabile di tipo enumerativo omonimo.
Poiché `valore` e `punti` assumono solo valori compresi tra 1 e 11, si è ritenuto opportuno utilizzare il tipo `char` per rappresentarli riducendo lo spazio occupato in memoria.
Il campo `pesoCost` è invece di tipo `double` per poter utilizzare parametri decimali nel suo calcolo: esso è ottenuto sommando una componente proporzionale ai punti della carta e un eventuale bonus se essa è una briscola.

5.1.2 Giocatore

Contiene le informazioni relative a uno dei due giocatori che disputano la partita.
Il campo `mano` è un array di puntatori a `Carta`: viene allocato in maniera dinamica per poter sfruttare agilmente dei puntatori ai suoi elementi.
Il valore del campo `minPunti` è il minimo punteggio della carta giocata dal primo di mano per cui il giocatore virtuale, non trovando una adeguata carta dello stesso seme, risponderà con una briscola per vincere la manche, se la possiede.

Il campo `memProb` specifica la probabilità, in percentuale, con cui il giocatore, se è virtuale, memorizzerà ogni carta giocata.
Questo è possibile grazie al campo `mem`, che è un array di booleani di grandezza pari al numero di carte nel mazzo (40), inizializzati al valore *false* e modificati in *true* se la carta corrispondente viene giocata e deve essere memorizzata.
Per semplificarne l'accesso da parte di altre funzioni, l'ordine delle carte a cui questo array si riferisce è quello per semi crescenti, e, per ogni seme, per valori crescenti: la carta di valore v e seme s avrà quindi indice $s * 10 + v - 1$.
Infine, il campo `errProb` indica la probabilità con cui il giocatore, se è virtuale, giocherà quando è il suo turno una carta casuale.
Questi ultimi quattro campi vengono inizializzati ed utilizzati solo quando il campo `isUtente` assume valore *false*, ovvero se il giocatore è quello virtuale.
Inoltre, per i valori da assegnare alle due probabilità nei vari livelli di difficoltà sono state definite nello stesso file alcune costanti per aumentare la leggibilità.

5.1.3 Partita

Contiene le informazioni necessarie allo svolgimento della partita.
Il campo `mazzo` consiste in un array di puntatori alle carte del mazzo.
Si usano i puntatori per migliorare l'efficienza: ad esempio, la funzione `mescolaCarte` deve così scambiare in memoria il contenuto di puntatori a strutture, che occupano uno spazio inferiore rispetto alla struttura stessa.
Gli stessi puntatori vengono debitamente copiati ad ogni turno nel campo `mano` dei due giocatori, rappresentati dai puntatori `g1` e `g2`, con lo scopo di distribuire le carte.
Il campo `puntPrimaCarta` è un puntatore all' elemento dell'array `mazzo` che identifica la prima carta ancora da distribuire: viene quindi incrementato durante la partita.
Il valore booleano `stampa` indica se le funzioni debbano stampare informazioni sullo svolgimento della partita o meno.
Per ultimo, il booleano `scambiati` indica se i due puntatori ai giocatori sono scambiati rispetto al momento dell'inizializzazione. Lo scambio serve a identificare come `g1` e `g2` rispettivamente il primo e il secondo di mano nella manche attuale.

5.2 Alcune informazioni sulle funzioni

Questa sezione fornisce alcuni dettagli riguardanti le funzioni che gestiscono la partita. Si considerano in particolare gli aspetti più critici; per informazioni complete si rimanda alla documentazione del file sorgente **BriscolaTS.c**.

Una nota comune a tutte le funzioni riguarda la gestione delle informazioni di scambio: quando esse sono delle strutture, esse vengono passate per copia di indirizzo per evitare onerose copie di intere strutture. Inoltre, ogni volta che

non fosse necessario modificare il contenuto delle aree di memoria puntate o effettuare altre operazioni che non lo consentano, i parametri passati in questo modo sono preceduti dal qualificatore `const`.

Un'altra osservazione riguarda la simulazione di eventi aleatori: a tale scopo è stato utilizzato il generatore di interi pseudo-casuali `rand`(), inizializzato dall'invocazione `srand(time(NULL))`.
Esso fornisce un valore compreso tra 0 e `RAND_MAX` (costante di libreria che si è verificato valere 32767): calcolandone il resto della divisione per un valore *bound*, si ottiene un intero nell'intervallo [0; *bound* - 1]. Se *bound* non è divisore della costante, ma è sufficientemente piccolo rispetto ad essa, si può approssimare la distribuzione dei numeri generati come uniforme nell'intervallo scelto.

Adottando questa ragionevole approssimazione, si è sfruttato il generatore in alcune istruzioni condizionate confrontando il numero restituito con una certa soglia.
Ad esempio, volendo eseguire una certa istruzione con probabilità percentuale P, si estrarrà un numero intero pseudo-casuale nell'intervallo [0; 99] (cioè vale *bound* = 100) e si procede con tale esecuzione solo se esso è minore di P: nell'ipotesi di distribuzione uniforme, questo avviene mediamente P volte su 100, ovvero con la probabilità attesa.

5.2.1 allocInizGiocatori

Il ciclo for di questa funzione distribuisce le prime tre carte del mazzo al primo giocatore e le successive tre al secondo; non sono necessarie allocazioni di memoria per le carte stesse perché la loro distribuzione consiste in una copia di puntatori.

5.2.2 stampaSeme

Questa funzione mette in evidenza come la corrispondenza tra i quattro caratteri usati per rappresentare i semi e i nomi dei semi stessi sia rilevante solo nell'interazione con l'utente.

5.2.3 isVincente

C'è una sola combinazione non valutata esplicitamente da nessuna condizione in questa funzione, ovvero quella in cui soltanto la seconda carta giocata nella mano sia una briscola. In tal caso la variabile `h` rimane al valore iniziale *false*, e ciò significa che, correttamente, vince quest'ultima carta.

5.2.4 contaMinoriSeme

La variabile `cont` viene incrementata solo se entrambi i valori booleani valgono *true*, ovverosia se la carta `c2` considerata non è ricordata dal giocatore come già uscita e se la stessa perderebbe giocata contro la carta `c`.
Si noti che la funzione `isVincente` considera anche l'ordine dei suoi argomenti, ma questo non è rilevante se le carte sono dello stesso seme.

5.2.5 getPeso

Questa funzione restituisce il peso di una carta `c`: esso si ottiene sommando alla sua componente costante il numero di carte che soddisfino i seguenti due requisiti:

- non sono state memorizzate dal giocatore che usa `c` nell'array `mem`;
- perderebbero se giocate in risposta a `c`.

In altre parole, questa componente è uguale al numero di carte che il giocatore "crede" di poter battere giocando `c`. Di conseguenza, questo valore è un'informazione tanto più precisa quanto più spesso il giocatore memorizza le carte giocate, ovverosia quanto più è alto il valore del suo campo `memProb`.

5.2.6 Funzioni che ritornano valori di tipo Carta**

La scelta di utilizzare questo puntatore come valore di ritorno è stata fatta per per segnalare, mediante il valore *NULL*, che non è stata trovata una carta rispondente alle specifiche richieste, e perché risultava comodo e leggibile, ad esempio nel momento di distribuire ai giocatori le carte del mazzo.

5.2.7 migliorVincente

La funzione `migliorVincente` implementa la strategia del giocatore virtuale, quando esso sia secondo di mano, in particolare la ricerca della miglior carta vincente da utilizzare. La soglia di punteggio per l'uso, se possibile e necessario, di una briscola, è data dal campo `minPunti` dello stesso giocatore.
Essa ritorna alla funzione chiamante la carta trovata, o in subordine il valore *NULL* se le due ricerche in cascata non hanno prodotto esito.

5.2.8 giocaTurno

`giocaTurno` gestisce il singolo turno del primo o secondo di mano: se si tratta dell'utente si risolve in una lettura da tastiera della carta scelta. In caso contrario il giocatore virtuale decide, in modo aleatorio e con probabilità `errProb`, se usare una carta casuale. Se questo non avviene:

- se il turno è quello del secondo di mano, il computer gioca la carta restituita dalla funzione `migliorVincente`;

- se quest'ultima restituisce però il valore *NULL*, o se il turno è invece quello del primo di mano, il giocatore virtuale usa la carta di peso minore che possiede.

5.2.9 giocaManche

In questa funzione si confrontano, tramite la funzione `isVincente`, le carte giocate, salvate in `c1` e `c2`. Se la seconda risulta vincente, vengono scambiati i puntatori ai giocatori ma anche `c1` e `c2`. In questo modo, viene garantito che la prima mano del turno successivo spetti al vincente di quello attuale, ma anche che lo stesso sia il primo a pescare: le carte estratte dal mazzo sostituiranno quelle puntate da `c1` e `c2`, appena giocate.
Negli ultimi tre turni il mazzo è esaurito e non ci sono carte da pescare: la funzione sovrascrive alla carta appena giocata l'ultima del campo `mano` dei giocatori.
Inoltre, viene invocata la funzione `memCarte` che, con probabilità uguale a quella salvata nel campo `memProb` del giocatore virtuale, memorizza nell'array `mem` dello stesso le due carte giocate nella manche corrente.

Capitolo 6

Ottimizzazione dei parametri

Lo scopo di questa sezione è analizzare come si sono ricavati i migliori valori da assegnare ad alcuni parametri utilizzati nel codice.
In particolare, con l'eseguibile ottenuto dal sorgente **BriscolaParTest.c** sono stati determinati separatamente:

- il valore ottimale del campo `minPunti` del tipo strutturato `Giocatore`;
- i valori migliori per gli argomenti `coeffP` e `pesoBrisc` da utilizzare nell'invocazione alla funzione `allocInizPartita`.

Si è scelto di adottare due approcci diversi e sequenziali, anziché calcolarli in parallelo, per due motivi. In prima battuta, perché le stime sono effettuate con scopi diversi; in seconda, per non appesantire eccessivamente la complessità computazionale: sarebbero infatti necessari tre cicli annidati, con singole iterazioni temporalmente onerose.

Per semplificare alcune operazioni eseguite ripetutamente nel programma, sono stati definiti: il tipo strutturato `Parametri` per incapsulare gli argomenti da fornire alle funzioni che inizializzano la partita, alcune funzioni per modificare i campi di una variabile di questo tipo, e una funzione `testNPartite` per simulare lo svolgimento di un numero di partite predefinito tramite costante tra due giocatori virtuali e restituire il numero di vittorie per ciascun giocatore.

I valori determinati dopo le simulazioni sono memorizzati tramite l'uso di costanti nel file **BriscolaTSCost.h**; i due parametri e il campo di tipo strutturato a cui si riferiscono non vengono comunque eliminati, per evidenziare i passaggi seguiti e permettere la ripetizione del processo indicato di seguito.

6.1 minPunti

Per quanto riguarda questo parametro, si procede simulando un "torneo".
Le partite si svolgono utilizzando valori nulli per `coeffP` e `pesoBrisc`, poiché quelli migliori vengono stimati in seguito; si può comunque a verificare a posteriori come il risultato ottimale per `minPunti` sia lo stesso anche utilizzando questi ultimi.
I campi che rappresentano le probabilità di errore e di memorizzazione sono uguali per entrambi i giocatori e sono modificati a rotazione nell'intervallo [0;

100], passando come argomento il valore *true* per il parametro `genPar` della funzione `testNPartite`.
Il torneo è stato simulato quindi con l'invocazione di quest'ultima funzione per tutte le possibili coppie di giocatori differenti per il valore del rispettivo campo `minPunti`.

I risultati sono stati mostrati su video, che ha indicato quante volte ogni minimo di punti avesse prevalso nel blocco di partite: per il valore **2** è successo in 5 "scontri" su 5, e questo è stato quindi memorizzato nella costante `BEST_MIN_PUNTI`.

6.2 coeffP e pesoBrisc

I valori ottimali per `coeffP` e `pesoBrisc` sono calcolati contemporaneamente per evitare stime falsate dall'uso di un valore arbitrario per il parametro non oggetto di valutazione.
Questa ottimizzazione si propone uno scopo differente: quello di massimizzare la differenza tra il livello più alto e quello più basso di difficoltà.

Ciò è implementato impostando nei due giocatori le probabilità di memorizzazione di tali livelli, e azzerando invece le probabilità di errore di entrambi, per evitare eccessive oscillazioni dei parametri migliori ottenuti in differenti esecuzioni dovute alle scelte aleatorie delle carte.
Due cicli annidati hanno permettono quindi di spaziare all'interno di intervalli arbitrari di valori, rispettivamente per il peso delle briscole e per il coefficiente dei punti. Ogni iterazione del ciclo interno simula quindi un blocco di partite associate tutte a una coppia (`coeffP`, `pesoBrisc`), mentre un'iterazione di quello esterno individua il `coeffP` migliore per un valore fissato di `pesoBrisc`, ovvero quello che massimizzi la differenza tra le partite vinte dal giocatore associato alla modalità difficile e quello associato alla modalità facile.

L'output finale riassume i risultati per ogni valore di `pesoBrisc`, e in conclusione restituisce la coppia di parametri migliori.
Poiché si può osservare un andamento decrescente della suddetta differenza massima al crescere di `pesoBrisc`, e considerando anche la ridotta variabilità dei valori di `coeffP` associati, è possibile ridurre l'ampiezza degli intervalli considerati per i due parametri e di conseguenza anche la loro variazione minima, senza influire eccessivamente sui tempi di esecuzione.
Riducendola a 0.1, vengono determinati i risultati definitivi, salvati anch'essi in opportune costanti nel file già citato:

`BEST_COEFF_PUNTI` **1.9**
`BEST_PESO_BRISC` **0**

Il valore nullo per il peso delle briscole potrebbe sorprendere, ma non deve essere né frainteso, né interpretato come una irrilevanza delle briscole: come evidenziato nel paragrafo 3.2.7, il loro ruolo è strategicamente differente dalle carte di altri semi.
Inoltre, il loro peso è comunque aumentato dalla loro capacità di risultare vincenti in più scontri: si rimanda alle osservazioni fatte in 3.2.5 sul calcolo di tale proprietà.

Conclusioni

Lo scopo principale di questo lavoro era quello di creare un programma che permettesse di giocare una partita di "Briscola a due" contro il computer.
Al di là delle scelte in fase di implementazione, l'algoritmo sviluppato ricalca una strategia potenzialmente applicabile dal giocatore umano; per questo motivo, i suoi pregi e difetti sono prima di tutto nelle scelte logiche effettuate a monte.

Indubbiamente, tale strategia ha ampi margini di miglioramento: ancor prima di considerare rielaborazioni complessive, si potrebbero sviluppare le sue idee chiave, come il peso della carta o la memorizzazione, per gestire in modo più articolato alcune situazioni di gioco.

Un altro elemento che potrebbe essere sfruttato ulteriormente è la simulazione. Nonostante non sia forse del tutto evidente, essa ha ricoperto un ruolo chiave in questo lavoro, perché presenta molti vantaggi. Ad esempio, testare un gran numero di partite mette in luce eventuali errori logici molto più probabilmente di una partita singola (es. errori nell'allocazione della memoria). Da un'adeguata simulazione si possono ottenere anche delle metriche statistiche molto utili per verificare il comportamento del giocatore virtuale al variare dei suoi parametri, e valutare se risponda alle aspettative.

Per concludere, mi esprimo per un'unica volta in prima persona.
Nonostante io sia consapevole dei limiti del mio lavoro, ne sono soddisfatto, perché mi ha permesso di cimentarmi in alcune sfide logiche relativamente semplici, ma stimolanti. Spero si possano apprezzare lo spirito pratico e il pizzico di creatività messi in campo per risolverle al meglio.

Appendice

Briscola.c

```
/* Marco Gasparini */

#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <stdbool.h>
#include "BriscolaTSCost.h"
#include "BriscolaProt.h"

int main(void) {
      Partita* p;
      srand(time(NULL));
      p = malloc(sizeof(Partita));
      assert(p != NULL);
      inizPartitaUtente(p);
      stampaEsito(giocaPartita(p), p);
      freePartita(p);
      return 0;
} /* main */
```

BriscolaTSCost.h

```
#define ERR_PROB_D 0
#define MEM_PROB_F 0
#define MEM_PROB_M 50
#define MEM_PROB_D 100

/* Le seguenti costanti sono state ottenute mediante simulazioni. */
#define BEST_MIN_PUNTI 2
#define BEST_COEFF_PUNTI 1.9
#define BEST_PESO_BRISC 0


/* Le definizioni di tipi enumerativi sostituiscono ripetuti usi della
   direttiva define. */
typedef enum {
      DENARI, COPPE, SPADE, BASTONI
} Seme;

typedef enum {
      FACILE, MEDIO, DIFFICILE
} Livello;

/* Carta, identificata da $seme e $valore.
   $punti contiene il punteggio della carta nella Briscola classica:
   2 punti per l'8, 3 per il 9, 4 per il 10, 10 per il 3 e 11 per
   l'asso.
   Tali valori consentono l'uso del tipo char per diminuire lo spazio
   occupato in memoria.
   $isBrisc indichera' se la carta sara' una briscola o meno.
   $pesoCost conterra' la componente del peso staticamente legata alla
   singola carta.
*/
typedef struct  {
      Seme seme;
      char valore;
      char punti;
      bool isBrisc;
      double pesoCost;
} Carta;

/* Informazioni su un giocatore.
   $isUtente indica se il giocatore corrisponda all'utente umano o
   meno.
   $mano e' un array di puntatori alle carte nel mazzo.
   $punti indichera' i punti correnti del giocatore.
   $minPunti indica il minimo punteggio della carta avversaria per
   raccoglierla se necessario con una briscola.
   $mem e' un array di booleani in cui, casualmente e con probabilita'
   uguale a $memProb, vengono memorizzate le carte giocate.
   $errProb indica la probabilita' con cui il giocatore giochera' una
   carta a caso.
*/
typedef struct {
      bool isUtente;
      Carta** mano;
      int punti;
      char minPunti;
      bool* mem;
      int memProb;
      int errProb;
} Giocatore;
```

```
/* Informazioni su una partita.
   $mazzo e' un array di puntatori alle carte del mazzo.
   $g1 e $g2 sono puntatori ai due giocatori.
   $puntPrimaCarta e' un puntatore all'elemento di $mazzo che
   rappresenta la prima carta non ancora distribuita.
   $ultimaCarta e' l'ultimo elemento di $mazzo.
   $manche indica la manche attuale.
   $stampa indica se deve essere prodotto output a video o meno.
   $campo scambiati indica se g1 e g2 sono scambiati rispetto
   all'inizializzazione.
*/
typedef struct {
      Carta* mazzo[N_CARTE_MAZZO];
      Giocatore *g1, *g2;
      Carta** puntPrimaCarta;
      Carta* ultimaCarta;
      int manche;
      bool stampa;
      bool scambiati;
} Partita;
```

BriscolaProt.h

```
/* Marco Gasparini */
/* INIZIALIZZAZIONE E GESTIONE DELLA MEMORIA */
void swap(void*, void*, int);
void inizPartitaUtente(Partita*);
void allocInizPartita(Partita*, double, double, bool);
void allocInizGiocatori(Partita*);
void inizCarta(Carta*, int);
void setBriscPesoCost(Carta*, Seme, double, double);
void setParamGiocatore(Giocatore*, bool, char, int, int);
void mescolaCarte(Carta**);
void freePartita(Partita*);
/* INPUT E OUTPUT */
Carta** leggiIndCartaMano(Carta**, int);
void stampaSeme(Seme);
void stampaGiocata(const Carta*, bool);
void stampaCarta(const Carta*, const char*);
void stampaEsito(char, const Partita*);
/* GESTIONE DEL GIOCATORE AUTOMATICO */
int contaMinoriSeme(const Carta*, const bool*);
int contaAltriSemi(const bool*, Seme);
double getPeso(const Carta*, const bool*);
Carta** minCarta(const Giocatore*, int);
Carta** migliorCartaSeme(Carta**, Carta**, int);
Carta** minBriscVincente(Carta**, const Carta**, int);
Carta** migliorVincente(const Giocatore*, Carta**, int);
void memCarteGiocatore(Giocatore*, const Carta*, const Carta*);
void memCarte(Partita*, const Carta*, const Carta*);
/* GESTIONE DELLA PARTITA */
bool isVincente(const Carta*, const Carta*);
Carta** giocaTurno(const Partita*, Carta**, int);
```

```
bool giocaManche(Partita*, int);

char giocaPartita(Partita*);
```

BriscolaImplem.c

```
/* Marco Gasparini */

#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>
#include "BriscolaTSCost.h"
#include "BriscolaProt.h"

/* Scambia il contenuto di due aree di memoria.
IOP p1 Puntatore alla prima area.
IOP p2 Puntatore alla seconda area.
IP nByte Numero di byte nelle due aree.
*/
void swap(void* p1, void* p2, int nByte) {
      char* temp = malloc(nByte);
      assert(temp != NULL);
      memcpy(temp, p2, nByte);
      memcpy(p2, p1, nByte);
      memcpy(p1, temp, nByte);
      free(temp);
} /* swap */

/*
Inizializza una partita tra utente umano e computer.
IOP + IK p Partita da inizializzare.
OV Informazioni sulla partita inizializzata.
*/
void inizPartitaUtente(Partita* p) {
      char liv; /* l'uso del tipo char garantisce una corretta
                   gestione degli errori di input. */
      int memProb, errProb;
      do {
            printf("\nInserisci il livello di difficolta':\n > 0:
            facile\n > 1: medio\n > 2: difficile\n");
            scanf("%c", &liv);
            fflush(stdin);
            if(liv > '2' || liv < '0')
                  printf("Inserisci uno dei livelli indicati!\n");
      } while(liv > '2' || liv < '0'); /* do-while */
      printf("\nPartita inizializzata: livello %c", liv);
      memProb = MEM_PROB_F * (liv == FACILE) + MEM_PROB_M * (liv ==
      MEDIO) + MEM_PROB_D * (liv == DIFFICILE);
      errProb = ERR_PROB_F * (liv == FACILE) + ERR_PROB_M * (liv ==
      MEDIO) + ERR_PROB_D * (liv == DIFFICILE);
      allocInizPartita(p, BEST_COEFF_PUNTI, BEST_PESO_BRISC, true);
      setParamGiocatore(p->g1, false, BEST_MIN_PUNTI, memProb,
      errProb);
      setParamGiocatore(p->g2, true, 0, 0, 0);
      printf("\n-- ULTIMA CARTA DEL MAZZO: ");
      stampaCarta(p->ultimaCarta, " --\n\n");
} /* inizPartitaUtente */
```

```
/*
Inizializza la partita e i suoi campi.
IOP p Partita da inizializzare.
IP coeffP Coefficiente per la parte del peso legata ai punti della
carta.
IP pesoBrisc Bonus al peso se la carta e' una briscola.
IP stampa Valore da assegnare al campo p->stampa.
*/
void allocInizPartita(Partita* p, double coeffP, double pesoBrisc,
bool stampa) {
      int i;
      p->puntPrimaCarta = p->mazzo;
      for(i = 0; i < N_CARTE_MAZZO; i++) {
            p->mazzo[i] = malloc(sizeof(Carta));
            assert(p->mazzo[i] != NULL);
            inizCarta(p->mazzo[i], i);
      } /* for */
      mescolaCarte(p->mazzo);
      p->ultimaCarta = p->mazzo[N_CARTE_MAZZO - 1];
      /* la briscola e' data dal seme dell'ultima carta del mazzo
         mescolato. */
      for(i = 0; i < N_CARTE_MAZZO; i++) {
            setBriscPesoCost(p->mazzo[i], p->ultimaCarta->seme, coeffP,
            pesoBrisc);
      } /* for */
      allocInizGiocatori(p);
      p->scambiati = false;
      p->stampa = stampa;
} /* allocInizPartita */

/*
Inizializza i campi isBrisc e pesoCost di $c.
IOP c Carta.
IP brisc Seme della briscola in questa partita.
IP coeffP, pesoBrisc Parametri per il calcolo del campo pesoCost di
$c.
*/
void setBriscPesoCost(Carta* c, Seme brisc, double coeffP, double
pesoBrisc) {
      c->isBrisc = (c->seme == brisc);
      c->pesoCost = coeffP * c->punti + pesoBrisc * c->isBrisc;
} /* setBriscPesoCost */
```

```
/*
Alloca lo spazio in memoria necessario per i giocatori e i relativi
campi in $p.
Inizializza i campi $mano dei giocatori.
IOP p Partita.
*/
void allocInizGiocatori(Partita* p) {
      int i;
      p->g1 = malloc(sizeof(Giocatore));
      assert(p->g1 != NULL);
      p->g2 = malloc(sizeof(Giocatore));
      assert(p->g2 != NULL);
      p->g1->punti = p->g2->punti = 0;
      p->g1->mano = malloc(MAX_CARTE_MANO * sizeof(Carta*));
      assert(p->g1->mano != NULL);
      p->g2->mano = malloc(MAX_CARTE_MANO * sizeof(Carta*));
      assert(p->g2->mano != NULL);
      for(i = 0; i < MAX_CARTE_MANO; i++) {
            p->g1->mano[i] = p->mazzo[i];
            p->g2->mano[i] = p->mazzo[i + MAX_CARTE_MANO];
      } /* for */
      p->puntPrimaCarta += 2 * MAX_CARTE_MANO;
} /* allocInitGiocatori */



/*
Imposta i parametri del giocatore $g.
Se il giocatore e' l'utente umano viene solo impostato il campo
isUtente.
IOP g Giocatore.
IP isUt Indica se il giocatore e' l'utente umano.
IP minPunti Minimo punteggio di una carta a cui $g deve rispondere con
una briscola, se la ha.
IP memProb Probabilita' con cui $g memorizzera' le carte giocate.
IP errProb Probabilita' con cui $g giochera' una carta casuale.
*/
void setParamGiocatore(Giocatore* g, bool isUt, char minPunti, int
memProb, int errProb) {
      if(!(g->isUtente = isUt)) {
            int i;
            g->minPunti = minPunti;
            g->memProb = memProb;
            g->mem = malloc(N_CARTE_MAZZO * sizeof(bool));
            assert(g->mem != NULL);
            for(i = 0; i < N_CARTE_MAZZO; i++)
                  g->mem[i] = false;
            g->errProb = errProb;
      } /* if */
} /* setParamGiocatore */
```

```
/*
Inizializza i campi di $c.
IOP c Carta.
IP ind Indice di $c nel mazzo non mescolato.
*/
void inizCarta(Carta* c, int ind) {
      int resto = ind % N_CARTE_SEME;
      c->valore = resto + 1;
      c->seme = ind / N_CARTE_SEME;
      if(resto == 0) c->punti = 11;
      else if(resto == 2) c->punti = 10;
      else if(resto == 7) c->punti = 2;
      else if(resto == 8) c->punti = 3;
      else if(resto == 9) c->punti = 4;
      else c->punti = 0;
} /* inizCarta */

/*
Mescola casualmente le carte di $mazzo:
ogni puntatore viene scambiato con un'altro scelto casualmente.
IOP mazzo Array di carte.
*/
void mescolaCarte(Carta* mazzo[]) {
      int i, ind;
      for(i = 0; i < N_CARTE_MAZZO; i++) {
            ind = rand() % N_CARTE_MAZZO;
            swap(mazzo + i, mazzo + ind, sizeof(Carta*));
      }
} /* mescolaCarte */

/*
Legge da tastiera la carta scelta tra quelle in mano.
IP arr Array di carte.
IP nCarteMano grandezza di $arr.
IK + OR Carta scelta.
*/
Carta** leggiCartaMano(Carta* arr[], int nCarteMano) {
      int i;
      char ind;
      for(i = 0; i < nCarteMano; i++) {
            printf("Carta %c: ", 'a' + i);
            stampaCarta(arr[i], "\n");
      } /* for */
      printf("Inserire l'indice della carta scelta: ");
      do {
            scanf("%c", &ind);
            fflush(stdin);
            if(ind < 'a' || ind >= 'a' + nCarteMano)
                  printf("Indice non presente! Riprova... ");
      } while(ind < 'a' || ind >= 'a' + nCarteMano); /* do-while */
      return &arr[ind - 'a'];
} /* leggiCartaMano */
```

```
/*
Stampa a video le informazioni sulla carta $c giocata.
IP c Carta.
IP isUtente Flag booleano che indica se $c e' stata giocata dall'uten-
te.
OV Informazioni su $c.
*/
void stampaGiocata(const Carta* c, bool isUtente) {
      if(isUtente)
            printf("\nHai giocato ");
      else
            printf("L'avversario ha giocato ");
      stampaCarta(c, "\n\n");
} /* stampaGiocata */

/*
Stampa a video valore e seme di una carta.
IP c Carta.
IP etich Stringa da aggiungere alla stampa.
OV Informazioni su $c.
*/
void stampaCarta(const Carta* c, const char* etich) {
      printf("%d di ", c->valore);
      stampaSeme(c->seme);
      printf("%s", etich);
} /* stampaCarta */

/*
Stampa a video per esteso il seme corrispondente a $seme.
IP seme Seme da stampare.
OV come specificato sopra.
*/
void stampaSeme(Seme seme) {
      if(seme == BASTONI) printf("Bastoni");
      if(seme == SPADE) printf("Spade");
      if(seme == COPPE) printf("Coppe");
      if(seme == DENARI) printf("Denari");
} /* stampaSeme */

/*
Stampa a video i dettagli di una partita terminata.
IP v Carattere che indica l'esito della partita.
IP p Partita terminata.
OV Dettagli.
*/
void stampaEsito(char v, const Partita* p) {
      printf("\n\nFINE PARTITA!");
      if(v == '1')
            printf("\n\nVince il computer!");
      else if(v == '2')
            printf("\n\n++ HAI VINTO! ++");
      else
            printf("\n\nPareggio!");
      printf("\n\nPunteggio:\n > computer %d punti\n > utente %d
      punti\n", p->g1->punti, p->g2->punti);
} /* stampaEsito */
```

```
/*
Stabilisce quale sia la vincente tra due carte $c1 e $c2.
IP c1, c2 Carte.
OR booleano che vale true se c1 e' vincente, false se lo e' c2.
*/
bool isVincente(const Carta* c1, const Carta* c2) {
      bool h = false;
      if(c1->seme == c2->seme) {
            if(c1->punti > c2->punti)
                  h = true;
            else if(c1->punti == c2->punti && c1->valore > c2->valore)
                  h = true;
            /* questa condizione si verifica solo se $c1 e $c2 hanno
               entrambe 0 punti. */
      }
      else if (c1->isBrisc && !c2->isBrisc)
            h = true;
      else if(!c2->isBrisc)
            h = true;
            /* giunti qui, se $c2 non e' una briscola, vince $c1
               giocata dal primo di mano. */
      return h;
} /* isVincente */

/*
Restituisce il numero di carte dello stesso seme di $c che la stessa
potrebbe battere, tra quelle non memorizzate come giocate in $memSeme.
IP c Carta rispetto a cui contare come specificato.
IP memSeme Array di booleani che indicano se le carte del seme di $c
siano state giocate.
OR Numero di carte come specificato.
*/
int contaMinoriSeme(const Carta* c, const bool memSeme[]) {
      int i;
      int cont = 0;
      Carta c2;
      for(i = 0; i < N_CARTE_SEME; i++) {
            inizCarta(&c2, i + c->seme * N_CARTE_SEME);
            /* operazione eseguita per poter utilizzare la funzione
               isVincente. */
            cont += !memSeme[i] * isVincente(c, &c2);
            /* incremento unitario se $c2 non e' stata giocata
               e perderebbe se le rispondesse $c in seconda mano. */
      } /* for */
      return cont;
} /* contaMinoriSeme */
```

```
/*
Restituisce il numero di carte di seme diverso da $s non ancora memo-
rizzate come giocate in $mem.
IP mem Array di booleani che indicano se le carte del mazzo siano sta-
te giocate.
IP s Seme da non considerare nel conto.
OR Numero di carte come specificato.
*/
int contaAltriSemi(const bool mem[], Seme s) {
      int i;
      int cont = 0;
      for(i = 0; i < N_CARTE_MAZZO; i++)
            cont += (!mem[i] && i / N_CARTE_SEME != s);
            /* i / N_CARTE_SEME permette di ottenere il seme della
               carta. */
      return cont;
} /* contaAltriSemi */

/*
Ritorna il peso di una carta.
IP c Carta di cui calcolare il peso.
IP mem Array di booleani che indicano se le carte del mazzo siano sta-
te giocate.
OR Peso di $c.
*/
double getPeso(const Carta* c, const bool mem[]) {
      double peso = c->pesoCost;
      peso += contaMinoriSeme(c, mem + N_CARTE_SEME * c->seme);
      peso += c->isBrisc * contaAltriSemi(mem, c->seme);
      /* Questo incremento avviene solo se $c e' una briscola. */
      return peso;
} /* getPeso */

/*
Restituisce la carta di peso minimo tra quelle in mano a $g.
IP g Giocatore.
IP nCarteMano Grandezza del campo $mano di $g.
OR Carta scelta come specificato.
*/
Carta** minCarta(const Giocatore* g, int nCarteMano) {
      int i, iMin = 0;
      double actPeso, minPeso = getPeso(g->mano[0], g->mem);
      for(i = 1; i < nCarteMano; i++)
            if((actPeso = getPeso(g->mano[i], g->mem)) < minPeso) {
                  minPeso = actPeso;
                  iMin = i;
            } /* if */
      return &(g->mano[iMin]);
} /* minCarta */
```

```
/*
Restituisce la carta in $arr di punti piu' alti, o con 0 punti e di
valore piu' alto, che batta $c1 e sia dello stesso suo seme.
Se non esiste restituisce NULL.
IP arr Array di carte.
IP c1 Carta da battere.
IP nCarteMano Grandezza di $arr.
OR Carta scelta come sopra.
*/
Carta** migliorCartaSeme(Carta* arr[], Carta** c1, int nCarteMano) {
      int i;
      Carta** cMax = c1;
      for(i = 0; i < nCarteMano; i++)
            if(arr[i]->seme == (*cMax)->seme && isVincente(arr[i],
            *cMax))
                  cMax = &arr[i];
      if(cMax == c1)
            return NULL;
            /* in tal caso non e' stata trovata nessuna carta che
               risponda a quanto richiesto. */
      return cMax;
} /* migliorCartaSeme */

/*
Restituisce la briscola di minori punti, o in subordine di minor valo-
re, in $arr per battere $c1.
Se non esiste restituisce NULL.
IP arr Array di carte.
IP c1 Carta da battere.
IP nCarteMano Grandezza di $arr.
OR Carta scelta come sopra.
*/
Carta** minBriscVincente(Carta* arr[], const Carta** c1, int nCarteMa-
no) {
      Carta** minB = NULL;
      char minPuntiB = 12; /* valore sentinella */
      int i;
      for(i = 0; i < nCarteMano; i++)
            if(arr[i]->isBrisc && arr[i]->punti < minPuntiB &&
            isVincente(arr[i], *c1)) {
                  minB = &arr[i];
                  minPuntiB = (*minB)->punti;
            } /* if */
      return minB;
} /* minBriscVincente */
```

```
/*
Restituisce la carta migliore in $arr per battere $c1, secondo la
strategia sviluppata.
Se non esiste restituisce NULL.
IP g2 Giocatore che deve rispondere a $c1.
IP c1 Carta da battere.
IP nCarteMano Grandezza del campo $mano di $g.
OR Carta scelta come sopra.
*/
Carta** migliorVincente(const Giocatore* g2, Carta** c1, int nCarteMa-
no) {
      Carta** c2 = NULL;
      if(!(*c1)->isBrisc)
            c2 = migliorCartaSeme(g2->mano, c1, nCarteMano);
      if(c2 == NULL && (*c1)->punti >= g2->minPunti)
            c2 = minBriscVincente(g2->mano, (const Carta**) c1,
            nCarteMano);
      return c2;
} /* migliorVincente */

/*
Gestisce l'eventuale memorizzazione (aleatoria) di $c1 e $c2 in $p->g1
e $p->g2.
IOP p Partita nei cui giocatori memorizzare eventualmente $c1 e $c2.
IP c1, c2 carte.
*/
void memCarte(Partita* p, const Carta* c1, const Carta* c2) {
      if(!p->g1->isUtente)
            memCarteGiocatore(p->g1, c1, c2);
      if(!p->g2->isUtente)
            memCarteGiocatore(p->g2, c1, c2);
} /* memCarte */

/*
Gestisce l'eventuale memorizzazione (aleatoria) di $c1 e $c2 in $g,
con probabilita' g->memProb.
IOP g Giocatore di cui aggiornare eventualmente il campo $g->mem.
IP c1, c2 carte.
*/
void memCarteGiocatore(Giocatore* g, const Carta* c1, const Carta* c2)
{
      if(rand() % 100 < g->memProb)
            g->mem[c1->valore - 1 + N_CARTE_SEME * c1->seme] = true;
            /* L'indice della carta viene calcolato ricordando che in
               $g->mem le carte sono ordinate per seme e valore. */
      if(rand() % 100 < g->memProb)
            g->mem[c2->valore - 1 + N_CARTE_SEME * c2->seme] = true;
} /* memCarteGiocatore */
```

```
/*
Gestisce lo svolgimento di uno dei due turni di una manche.
IP p Partita in svolgimento.
IP c1 Carta giocata dal primo di mano, o il valore NULL
se il turno attuale e' quello del primo di mano.
IP nCarteMano Grandezza del campo $mano dei giocatori.
OR Carta giocata.
*/
Carta** giocaTurno(const Partita* p, Carta** c1, int nCarteMano) {
      Carta** c = NULL;
      Giocatore* g = (c1 == NULL) ? p->g1 : p->g2;
      if(p->stampa)
            (c1 == NULL) ? printf("> PRIMO DI MANO\n") : printf("
            SECONDO DI MANO\n");
      if(g->isUtente)
            c = leggiCartaMano(g->mano, nCarteMano);
      else if(rand() % 100 < g->errProb)
            c = &(g->mano[rand() % nCarteMano]);
      else if(c1 != NULL)
            c = migliorVincente(g, c1, nCarteMano);
      if(c == NULL)
            c = minCarta(g, nCarteMano);
      if(p->stampa)
            stampaGiocata(*c, g->isUtente);
      return c;
} /* giocaTurno */

/*
Gestisce lo svolgimento di una singola manche della partita.
IOP p Partita in svolgimento.
IP nCarteMano Grandezza del campo $mano dei giocatori.
OV Informazioni sullo svolgimento.
OR true se la manche e' vinta da $p->g1, false se e' vinta da $p->g2.
*/
bool giocaManche(Partita* p, int nCarteMano) {
      Carta **c1, **c2;
      bool vinc1;
      c2 = giocaTurno(p, c1 = giocaTurno(p, NULL, nCarteMano),
      nCarteMano);
      memCarte(p, *c1, *c2);
      vinc1 = isVincente(*c1, *c2);
      if(!vinc1) {
            swap(&p->g1, &p->g2, sizeof(Giocatore*)); /* cosi' il
            giocatore vincente viene salvato in $p->g1. */
            swap(&c1, &c2, sizeof(Carta**)); /* scambio parallelo. */
      } /*if */
      p->g1->punti += (*c1)->punti + (*c2)->punti;
      if(p->manche < N_MANCHES - 3) {
            *c1 = *(p->puntPrimaCarta);
            *c2 = *(++(p->puntPrimaCarta));
            /* Distribuzione delle carte. */
            (p->puntPrimaCarta) ++;
      } /* if */
      else {
            *c1 = p->g1->mano[nCarteMano - 1];
            *c2 = p->g2->mano[nCarteMano - 1];
      } /* else */
      /* Sovrascrittura dell'ultima carta in mano ai giocatori. */
      return vinc1;
} /* giocaManche */
```

```
/*
Gestisce lo svolgimento di una partita.
IOP p Partita inizializzata.
OV Informazioni sullo svolgimento.
OR esito della partita:
   '1', '2', 'P' a seconda che vinca il Giocatore 1, il 2 o ci sia un
   pareggio.
*/
char giocaPartita(Partita* p) {
      int nCarteMano = MAX_CARTE_MANO;
      if(rand() % 2) {
            swap(&p->g1, &p->g2, sizeof(Giocatore*)); /* cosi' inizia
            un giocatore casuale. */
            p->scambiati = true;
      } /* if */
      for(p->manche = 0; p->manche < N_MANCHES; (p->manche) ++) {
            if(p->stampa) {
                  printf("\n\n-- MANCHE %d -- ULTIMA CARTA DEL MAZZO:
                  ", p->manche + 1);
                  stampaCarta(p->ultimaCarta, " --\n\n");
            } /* if */
            if(p->manche > N_MANCHES - 3)
                  nCarteMano--;
            if(!giocaManche(p, nCarteMano))
                  p->scambiati = !p->scambiati;
                  /* Lo scambio avviene se vince la manche $p->g2. */
      } /* for */
      if(p->scambiati)
            swap(&p->g1, &p->g2, sizeof(Giocatore*));
      /* I puntatori $p->g1 e $p->g2 tornano nell'ordine che avevano
      al momento dell'inizializzazione della partita. */
      if(p->g1->punti > p->g2->punti) return '1';
      if(p->g2->punti > p->g1->punti) return '2';
      return 'P';
} /* giocaPartita */

/*
Libera lo spazio allocato in maniera dinamica per $p.
IOP p Partita.
*/
void freePartita(Partita* p) {
      int i;
      for(i = 0; i < N_CARTE_MAZZO; i++)
            free(p->mazzo[i]);
      free(p->g1->mano);
      free(p->g2->mano);
      free(p->g1->mem);
      free(p->g2->mem);
      free(p->g1);
      free(p->g2);
      free(p);
} /* freePartita */
```

BriscolaParTest.c

```
/* Marco Gasparini */

#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <stdbool.h>
#include "BriscolaTSCost.h"
#include "BriscolaProt.h"

#define N_PARTITE 100000

#define DELTA_COEFF 0.1
#define DELTA_PESO 1
#define N_PESI 10
#define N_COEFF 25

/* Incapsula i parametri legati alla partita per migliorare la leggi-
bilita'.
La definizione e' inserita qui poiche' la struttura risulta utile
principalmente in fase di test.
*/
typedef struct {
      char minP1, minP2;
      int memP1, memP2;
      int errP1, errP2;
      double coeffP, pesoBrisc;
} Parametri;

/* Le funzioni seguenti compattano il codice e migliorano la leggibi-
lita'.
IOP par Parametri.
IP Parametri specifici da assegnare ai campi di $p.
*/
void setMinPunti(Parametri* par, char minP1, char minP2) {
      par->minP1 = minP1;
      par->minP2 = minP2;
} /* setMinPunti */

void setMemProb(Parametri* par, int memP1, int memP2) {
      par->memP1 = memP1;
      par->memP2 = memP2;
} /* setMemProb */

void setErrProb(Parametri* par, int errP1, int errP2) {
      par->errP1 = errP1;
      par->errP2 = errP2;
} /* setErrProb */

void setCoeffPeso(Parametri* par, double coeffP, double pesoBrisc) {
      par->coeffP = coeffP;
      par->pesoBrisc = pesoBrisc;
} /* setCoeffPeso */
```

```
/*
Simula N_PARTITE tra due giocatori virtuali, restituendo il numero di
vittorie
di ciascuno.
I parametri sono gli stessi per ogni partita, tranne le probabilità di
errore
e memorizzazione, ma solo nel caso in cui $genPar abbia valore true:
in tal caso
assumono a rotazione valori tra 0 e 100 uguali per i due giocatori.
IP par Parametri per la simulazione.
IP genPar Booleano che indica se le probabilita' debbano essere gene-
rate a rotazione o meno.
OP v1, v2 Vittorie per giocatore.
*/
void testNPartite(Parametri* par, int* v1, int* v2, bool genPar) {
      int i;
      Partita* p;
      char v;
      for(i = 0; i < N_PARTITE; i++) {
            p = malloc(sizeof(Partita));
            assert(p != NULL);
            if(genPar) {
                  setMemProb(par, i % 101, i % 101);
                  setErrProb(par, i % 101, i % 101);
            } /* if */
            allocInizPartita(p, par->coeffP, par->pesoBrisc, false);
            setParamGiocatore(p->g1, false, par->minP1, par->memP1,
            par->errP1);
            setParamGiocatore(p->g2, false, par->minP2, par->memP2,
            par->errP2);
            v = giocaPartita(p);
            freePartita(p);
            if(v ==  '1') (*v1)++;
            else if(v == '2') (*v2)++;
      } /* for */
} /* testNPartite */

int main(void) {
      int v1, v2, i, j;
      double bestCoeff, bestLocCoeff, bestLocWinDiff, bestWinDiff,
      actWinDiff, bestPeso;
      char arrMin[6] = {0, 2, 3, 4, 10, 11};
      int vittorie[6] = {0, 0, 0, 0, 0, 0};
      double bestVittorie[N_PESI];
      double bestCoeffs[N_PESI];
      Parametri par;
      srand(time(NULL));

/* scelta del valore migliore per minPunti */
      setCoeffPeso(&par, BEST_COEFF_PUNTI, BEST_PESO_BRISC);
      for(i = 0; i < 6; i++)
            for(j = i + 1; j < 6; j++) {
                  printf("%d vs %d\n", arrMin[i], arrMin[j]);
                  v1 = v2 = 0;
                  setMinPunti(&par, arrMin[i], arrMin[j]);
                  testNPartite(&par, &v1, &v2, true);
                  printf("minPunti %d vince %f\n", arrMin[i], ((double)
                  v1) / N_PARTITE);
                  printf("minPunti %d vince %f\n\n", arrMin[j],
                  ((double) v2) / N_PARTITE);
                  if(v1 > v2) vittorie[i]++;
                  if(v2 > v1) vittorie[j]++;
            }
```

```
      for(i = 0; i < 6; i++) printf("minPunti %d vince %d volte\n",
                              arrMin[i], vittorie[i]);

/* test: giocatore in mod. difficile contro gioc. in mod. media. */
      v1 = v2 = 0;
      setMinPunti(&par, BEST_MIN_PUNTI, BEST_MIN_PUNTI);
      setMemProb(&par, MEM_PROB_D, MEM_PROB_M);
      setErrProb(&par, ERR_PROB_D, ERR_PROB_M);
      setCoeffPeso(&par, BEST_COEFF_PUNTI, BEST_PESO_BRISC);
      testNPartite(&par, &v1, &v2, false);
      printf("\nMod. difficile contro media: %f a %f\n",
      ((double) v1) / N_PARTITE, ((double) v2) / N_PARTITE);

/* test: giocatore in mod. media contro gioc. in mod. facile */
      v1 = v2 = 0;
      setMinPunti(&par, BEST_MIN_PUNTI, BEST_MIN_PUNTI);
      setMemProb(&par, MEM_PROB_M, MEM_PROB_F);
      setErrProb(&par, ERR_PROB_M, ERR_PROB_F);
      setCoeffPeso(&par, BEST_COEFF_PUNTI, BEST_PESO_BRISC);
      testNPartite(&par, &v1, &v2, false);
      printf("Mod. media contro facile: %f a %f\n",
      ((double) v1) / N_PARTITE, ((double) v2) / N_PARTITE);

/* scelta dei valori migliori per coeffPunti e pesoBrisc */
      bestWinDiff = -1; /* valore sentinella */
      par.pesoBrisc = 0;
      setMemProb(&par, MEM_PROB_D, MEM_PROB_F);
      setErrProb(&par, ERR_PROB_D, ERR_PROB_D);
      for(i = 0; i < N_PESI; i++){
            par.coeffP = 0;
            bestLocWinDiff = -1;
            for(j = 0; j < N_COEFF; j++){
                  v1 = v2 = 0;
                  testNPartite(&par, &v1, &v2, false);
                  printf("\nParametri:\n > coeffPunti %f\n > pesoBrisc
                  %f\n", par.coeffP, par.pesoBrisc);
                  printf(" > vittorie 1: %f\n > vittorie 2: %f\n",
                  (double) v1 / N_PARTITE, (double) v2 / N_PARTITE);
                  actWinDiff = ((double) v1 - (double) v2) / N_PARTITE;
                  if(actWinDiff > bestLocWinDiff) {
                        bestLocWinDiff = actWinDiff;
                        bestLocCoeff = par.coeffP;
                  } /* if */
                  par.coeffP += DELTA_COEFF;
            } /* for */
            if(bestLocWinDiff > bestWinDiff) {
                  bestWinDiff = bestLocWinDiff;
                  bestCoeff = bestLocCoeff;
                  bestPeso = par.pesoBrisc;
            } /* if */
            bestVittorie[i] = bestLocWinDiff;
            bestCoeffs[i] = bestLocCoeff;
            par.pesoBrisc += DELTA_PESO;
      } /* for */
      for(i = 0; i < N_PESI; i++)
            printf("PesoBrisc: %f bestLocCoeff: %f Differenza vittorie
            perc.: %f\n",
            (double) DELTA_PESO * i, bestCoeffs[i], bestVittorie[i]);
            printf("\n\nCoeff. punti migliore: %f\npeso migliore:
            %f\nvittorie: %f\n", bestCoeff, bestPeso, bestWinDiff);

      return 0;
} /* main */
```