



UNIVERSITÀ DEGLI STUDI DI MILANO - BICOCCA

Scuola di Scienze

Dipartimento di Informatica, Sistemistica e Comunicazione

Corso di laurea in Informatica

Applicazione per dispositivi mobili basata sul principio della bisimulazione

Relatore: Prof.ssa Micucci Daniela

Correlatore: Dott. Ginelli Davide

Correlatore: Prof. Bernardinello Luca

Relazione della prova finale di:

Lanzi Mattia

Matricola 827502

Anno Accademico 2020-2021

Indice

1	Bisimulazione	7
2	Organizzazione applicazione	9
2.1	Backend	9
2.1.1	Interazione con Firebase	9
2.1.2	Registrazione	9
2.1.3	Autenticazione	10
2.2	Realtime Database	11
2.2.1	Scrittura dati	11
2.2.2	Lettura dati	12
2.3	Modulo logico	12
2.3.1	Mosse forti	13
2.3.2	Mosse deboli	13
2.3.3	Controllo mossa valida	14
2.3.4	Controlli fine partita	14
2.4	Frontend	15
2.4.1	Material Design	15
2.4.2	Componenti Sign Up Activity	15
2.4.3	Componenti Login Activity	15
2.4.4	Componenti Home Fragment	15
2.4.5	Componenti Matchmaking Room Activity	15
2.5	Canvas	15
2.5.1	Directed Graph	16
2.5.2	Node	17
2.5.3	Edge	17
3	Partita in tempo reale	19
3.1	Lettura file di configurazione	19
3.2	Interazione con Realtime Database	19
3.2.1	Preparazione	19
3.2.2	Svolgimento	20
4	Manuale utente	23
4.1	Manuale d'uso	23
4.2	Guida all'uso	23
4.2.1	Registrazione	23
4.2.2	Login	25
4.2.3	Home	25
4.2.4	Il mio profilo	27
4.2.5	Giocatori attivi	27
4.2.6	FAQ	27

4.2.7	Impostazioni	28
4.2.8	Segnala un problema	28
4.2.9	Stanza pre-partita	28
4.2.10	Gioco	29
4.3	Regole	30
5	Conclusioni	33
6	Sviluppi futuri	35
6.1	Scelta partita casuale	35
6.2	Avversario CPU	35
6.3	Animazioni	35
6.4	Impostazioni	35

Introduzione

Il progetto di tesi consiste nello sviluppo di un'applicazione per dispositivi mobili che utilizzano una versione del sistema operativo Android.

L'applicazione dal titolo **Bisimulazione** permette di sperimentare un gioco basato sul principio della bisimulazione.

L'obiettivo della tesi è la realizzazione di una app Android che sia da supporto didattico agli studenti che affrontano lo studio della bisimulazione; in particolare, l'app sviluppata supporta lo svolgimento di un gioco dove attaccante e difensore si affrontano navigando un grafo fino a che sono disponibili delle mosse valide.

L'applicazione Bisimulazione ha dunque uno scopo prettamente didattico in quanto il suo fine è quello di essere utilizzato per comprendere il meccanismo della bisimulazione all'interno del corso magistrale sui sistemi di metodi formali e sui modelli di sistemi concorrenti.

Il seguente elaborato propone come primo capitolo la definizione del principio di bisimulazione e dei suoi concetti derivati sia formalmente che informalmente.

Di seguito viene illustrata l'organizzazione dell'applicazione dividendo il capitolo in tre macro argomenti: il backend, il frontend e il modulo logico.

Successivamente nel terzo capitolo si mostra il meccanismo della partita in tempo reale dal punto di vista dell'interazione con il database.

Segue il manuale utente che spiega all'utente come utilizzare il gioco e le regole dello stesso.

Infine vengono tratte le conclusioni e i possibili sviluppi futuri che l'applicazione può subire per migliorare l'esperienza dell'utente.

Obiettivo dello stage

Come già anticipato, l'obiettivo dello stage è stato quello di implementare un'applicazione per dispositivi mobili che possa essere utilizzata come sostegno all'insegnamento e all'apprendimento del concetto della bisimulazione. Per lo sviluppo del progetto è stato scelto di programmare utilizzando l'**IDE** (*integrated development environment*) **Android Studio**, un ambiente di sviluppo integrato adibito per la creazione di applicazioni Android, basato sul linguaggio **Java** per il *backend* e sull'**XML** (*extensible Markup Language*) per il *frontend*. Android Studio offre inoltre un ampio supporto a servizi di terze parti, in particolare ai servizi di **Google**, tra cui **Firebase**, utilizzato principalmente come *Realtime Database*.

Dispositivi Android

Secondo l'articolo riportato da Ericsson Mobility Report [10] sono 5.9 miliardi le persone nel mondo che possiedono un telefonino ed il numero di abbonamenti LTE risulta essere in crescita costante.

Tra questi emergono i dispositivi con sistema operativo Android, che risulta essere il sistema operativo più diffuso al mondo per quanto riguarda i dispositivi mobili (Figura 1). I fattori che hanno determinato questo primato di Android sono i prezzi contenuti per i mercati emergenti e la disponibilità di poter usufruire di migliaia di app per la salute e per l'istruzione.

Android, più di qualunque altra piattaforma per dispositivi mobili, aiuta le persone ad accedere alle informazioni e a più opportunità.

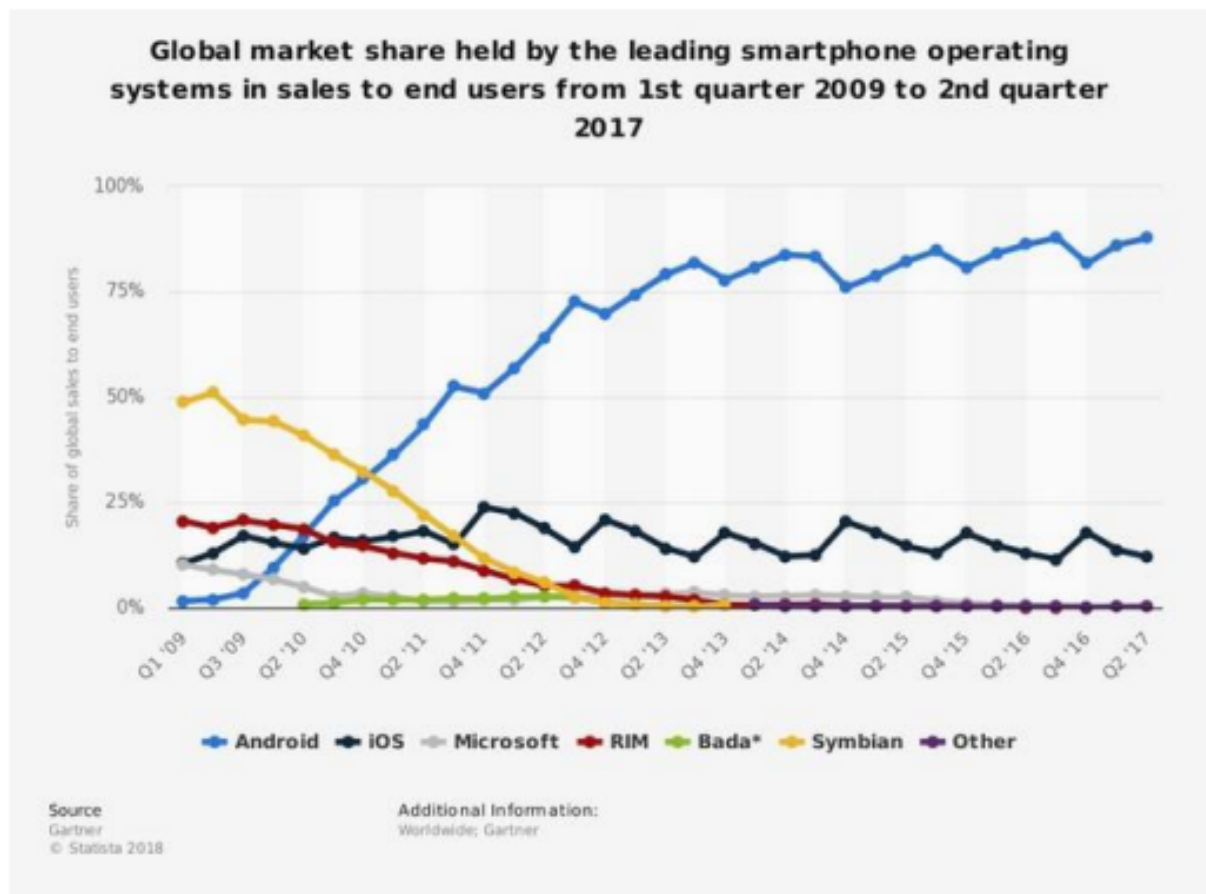


Figura 1: Grafico smartphone con sistema operativo Android.

Capitolo 1

Bisimulazione

In questa sezione vengono presentati dal punto di vista formale i concetti di sistema a transizione di stati, transizione debole e bisimulazione debole [13].

SISTEMA DI TRANSIZIONI ETICHETTATE

Un *sistema di transizioni etichettate* (dall'inglese *labelled transition system*), talvolta chiamato *transition graph*, è una tripla $(\mathbf{Proc}, \mathbf{Act}, \{ \xrightarrow{\alpha} \mid \alpha \in \mathbf{Act} \})$ dove:

- \mathbf{Proc} è un insieme di *stati* (o *processi*);
- \mathbf{Act} è un insieme di *azioni* (o *etichette*);
- $\xrightarrow{\alpha} \subseteq \mathbf{Proc} \times \mathbf{Proc}$ è una *relazione di transizione* per ogni $\alpha \in \mathbf{Act}$.

Un sistema a transizioni di stati è *finito* se i suoi insiemi di stati e azioni sono anch'essi finiti.

TRANSIZIONE DEBOLE

Assumiamo che P e Q siano stati in un LTS. Per ogni azione di α , diciamo che $P \xRightarrow{\alpha} Q$ se una delle due condizioni è verificata:

- $\alpha \neq \tau$ e ci sono stati P' e Q' tali che
$$P(\xrightarrow{\tau})^* P' \xrightarrow{\alpha} Q' (\xrightarrow{\tau})^* Q$$
- oppure $\alpha = \tau$ e $P(\xrightarrow{\tau})^* Q$,

dove $(\xrightarrow{\tau})^*$ indica la chiusura riflessiva e transitiva sulla relazione $\xrightarrow{\tau}$.

BISIMULAZIONE DEBOLE

Una relazione binaria R sull'insieme di stati di un LTS è una *weak bisimulation* (relazione di bisimulazione debole) se $s_1 R s_2$ e α è un'azione e:

- se $s_1 \xrightarrow{\alpha} s'_1$, allora c'è una transizione $s_2 \xRightarrow{\alpha} s'_2$ tale che $s'_1 R s'_2$.
- se $s_2 \xrightarrow{\alpha} s'_2$, allora c'è una transizione $s_1 \xRightarrow{\alpha} s'_1$ tale che $s'_1 R s'_2$.

Due stati s ed s' sono *weakly bisimilar* (*debolmente bisimili*), se e solo se c'è una relazione di bisimulazione che li unisce.

Capitolo 2

Organizzazione applicazione

2.1 Backend

2.1.1 Interazione con Firebase

Firebase, servizio offerto da Google per l'utilizzo di funzionalità a supporto dello sviluppo mobile, interagisce con l'applicazione sotto tre aspetti principali:

1. registrazione all'applicazione;
2. autenticazione all'applicazione;
3. lettura e scrittura dati sul database.

Nelle prossime sezioni verrà analizzato nel dettaglio ciascuno di questi tre aspetti.

Al fine di poter utilizzare il servizio di autenticazione e registrazione di Firebase è stato necessario dichiarare la dipendenza per la libreria Firebase Authentication nel modulo File Gradle a livello di app.

Allo stesso modo per poter usufruire delle funzionalità inerenti al database è stato necessario dichiarare la dipendenza per la libreria Firebase Database nel modulo File Gradle a livello di app.

2.1.2 Registrazione

La classe Java *SignUp* si occupa della registrazione di un utente alla piattaforma Firebase che permetterà poi di autenticarsi al fine di poterne usufruire dei servizi.

La modalità scelta per la registrazione è la classica modalità che utilizza email e password. Uno degli attributi con modificatore privato di questa classe è *auth*, un oggetto della classe *FirebaseAuth*; su questo oggetto, definito nel metodo *onCreate()*[4] (metodo che viene eseguito appena l'activity corrispondente alla classe viene lanciata), tramite il metodo *getInstance()* che ritorna un'istanza default della classe *FirebaseApp*, viene chiamato il metodo *createUserWithEmailAndPassword()*[3]. Quest'ultima funzione legge come argomento due stringhe, l'indirizzo mail inserito dall'utente e la password, anch'essa inserita dall'utente, e prova a creare un nuovo account utente con l'indirizzo mail e la password fornita. Questo metodo lancia tre eccezioni:

1. ***FirebaseAuthWeakPasswordException***;
2. ***FirebaseAuthInvalidCredentialsException***;
3. ***FirebaseAuthUserCollisionException***.

La prima eccezione viene lanciata nel momento in cui la password non è sufficientemente "forte", ovvero è inferiore ai 6 caratteri.

Per quanto riguarda l'eccezione *FirebaseAuthInvalidCredentialsException* essa viene lanciata se l'indirizzo email inserito non segue il pattern corretto.

Infine l'ultima eccezione viene lanciata se è già presente un account con l'indirizzo mail inserito dall'utente.

Ogni eccezione viene gestita da un blocco catch specifico che provvede a mostrare all'utente messaggi differenti in modo da spiegare le cause degli errori.

Se il processo va a buon fine, questa funzione autentica anche l'utente all'interno dell'applicazione; inoltre, le generalità inserite dall'utente vengono memorizzate all'interno del Realtime Database di Firebase: viene creato un oggetto *map* di tipo `HashMap<String, String>` a cui vengono aggiunti tramite metodo `put()` tutte le informazioni necessarie e poi salvate nel database. L'utente sarà aggiunto all'interno del nodo *users*, come rappresentato nella figura 2.1.

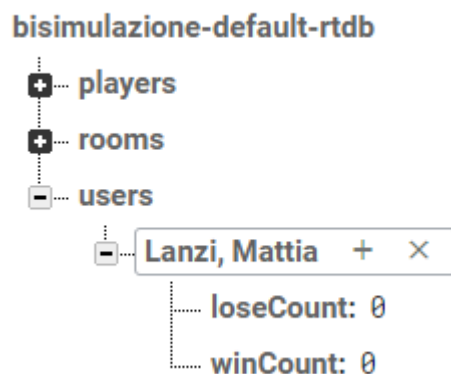


Figura 2.1: Creazione di un utente nel database.

2.1.3 Autenticazione

L'autenticazione dell'utente all'interno dell'applicazione per utilizzare i servizi offerti da Firebase viene gestita dalla classe *Login*. L'autenticazione avrà successo solo se l'utente in precedenza si è già registrato; in caso contrario non si potrà accedere e dunque non sarà reso possibile l'utilizzo dell'applicazione **Bisimulazione**.

A livello di codice l'autenticazione è gestita dal metodo *signInWithEmailAndPassword()*[6] che, così come per il metodo *createUserWithEmailAndPassword()* descritto nella sezione precedente, legge come argomento due stringhe, l'indirizzo mail e la password forniti dall'utente, e permette il login all'interno dell'applicazione. Questo metodo lancia due eccezioni:

- **FirebaseAuthInvalidUserException;**
- **FirebaseAuthInvalidCredentialsException.**

Le due eccezioni vengono lanciate rispettivamente se l'indirizzo mail dell'account utente non esiste oppure è stato disabilitato, o se la password inserita non è corretta.

Così come per le eccezioni descritte nella sezione precedente, anche in questo caso è presente un blocco catch per ognuna di esse.

La memorizzazione delle credenziali avviene tramite l'interfaccia *SharedPreferences*[11] che permette di accedere e modificare i dati a livello locale, non a livello di Realtime Database. I dati collocati nelle *SharedPreferences* vengono salvati in un file XML contenuto nello storage interno,

in particolare nella cartella *shared_prefs*. Il file in cui sono contenute le informazioni è chiamato "*sharedPreferencesLogin*". In questo file vengono salvati i dati di accesso (indirizzo mail e password) dell'utente che saranno poi memorizzati per il successivo utilizzo.

Il salvataggio dei dati in questo file locale avviene solamente nel caso in cui l'utente abbia contrassegnato con un flag la checkbox accompagnata dalla descrizione "*Ricordami*". In caso contrario, i dati non vengono salvati e se l'utente dovesse dimenticarsi della password utilizzata, è possibile effettuare un reset utilizzando sempre una funzionalità offerta da Firebase; nello specifico, per mezzo del metodo `sendPasswordResetEmail()` che prende come parametro l'indirizzo mail dell'utente e rende possibile effettuare questa operazione. Questo metodo attiva il backend di Firebase Authentication per inviare un'email di reimpostazione della password all'indirizzo email fornito che deve corrispondere ad un utente esistente già registrato a Bisimulazione.

2.2 Realtime Database

Realtime Database è un database ospitato nel cloud. I dati vengono archiviati come *JSON* e sincronizzati in tempo reale con ogni client connesso, rimanendo disponibili fino a che l'app diventa offline. Con questo database è possibile archiviare e sincronizzare i dati con il database dal modello NoSQL[7].

Un aspetto fondamentale che ha favorito l'utilizzo del Realtime Database come supporto all'applicazione è che, invece delle tipiche richieste HTTP, Firebase Realtime Database utilizza una sincronizzazione in tempo reale dei dati: ogni volta che i dati cambiano, qualsiasi dispositivo connesso riceve l'aggiornamento in pochi millisecondi. Questo aspetto è fondamentale per la gestione di una partita real time.

2.2.1 Scrittura dati

Per l'operazione di scrittura all'interno del database viene utilizzato il metodo `setValue()`[5] che permette di salvare il dato passato come argomento in un riferimento specifico, sostituendo qualsiasi dato esistente presente in quel percorso.

Questo metodo è stato utilizzato in maniera molto frequente e un esempio chiaro del suo funzionamento è possibile trovarlo nella classe Java **MatchmakingRoom**: qui quando viene creata una partita si passa al riferimento della partita stessa tutte le informazioni poi smistate nei vari nodi figlio.

Ogni operazione di scrittura genera un nodo figlio di uno dei tre nodi principali del database rappresentati in figura 2.2.

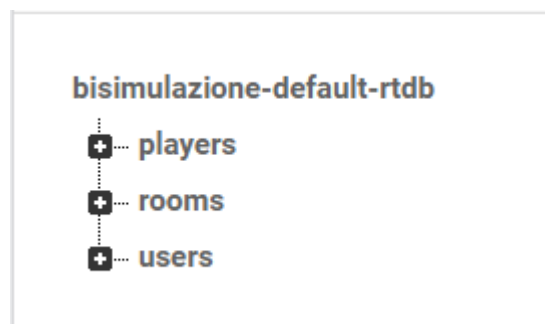


Figura 2.2: Nodi principali del database

2.2.2 Lettura dati

La lettura dei dati salvati all'interno del database è fondamentale per la creazione, lo sviluppo e il termine delle sfide in tempo reale.

Per leggere un valore presente nel database è stato utilizzato il meccanismo dei callback.

I dati memorizzati all'interno del Firebase Realtime Database sono recuperati collegando un ascoltatore asincrono (*asynchronous listener*) che punta ad un riferimento del database. Il listener è attivato appena il dato viene trasmesso al database e in seguito si attiva ogni qual volta il dato cambia (si aggiorna o si elimina).

Innanzitutto viene dichiarata un'interfaccia. Un'interfaccia è un componente di un programma che contiene le intestazioni di un certo numero di metodi. Un'interfaccia può anche definire delle costanti pubbliche.[14]

Di seguito la classe in cui si dovrà leggere il dato o i dati eredita l'interfaccia generata. A questo punto viene creato un metodo che legge come parametro il metodo callback definito nell'interfaccia; tramite un ascoltatore si legge il dato dal database che viene poi passato al callback. Una volta che il processo di lettura del dato termina, il callback rilascia il valore del dato letto nel metodo in cui è stato definito.

Per lo sviluppo di quest'applicazione mobile sono stati utilizzati esclusivamente due listener:

1. ascoltatore singolo (*ListenerForSingleValueEvent*);
2. ascoltatore persistente (*ValueEventListener*).

Un ascoltatore singolo permette di leggere i dati una volta sola; più nello specifico appena il callback viene chiamato e termina il suo processo, questo viene immediatamente rimosso.

Un esempio di utilizzo di un *single value listener* è all'interno della classe Java **Table**: una volta letto il nome dell'attaccante e definito all'interno dell'info box non è più necessario mantenere un ascoltare su questo valore, quindi viene letto una volta sola e poi rimosso.

In maniera differente un ascoltatore persistente serve per tenere traccia di aggiornamenti ad uno specifico riferimento.

Un esempio di utilizzo di un *value event listener* è possibile trovarlo sempre all'interno della classe Java **Table**: qui è necessario che venga letto il valore del riferimento *turnOf* ogni volta che esso viene aggiornato per far sì che i giocatori che si sfidano sappiano in tempo reale a chi tocca eseguire una mossa.

2.3 Modulo logico

Dal punto di vista logico, la maggior parte dei controlli partono dal metodo *setOnTouchGraph()* contenuto nella classe **Table** dove si svolge effettivamente la partita tra i due giocatori. Il metodo riceve come argomento un oggetto *event* della classe **MotionEvent** e un oggetto *directedGraph* della classe **DirectedGraph** (definita nella sezione 2.5.1).

Il metodo effettua un ciclo continuo sui nodi del grafo passato come argomento.

Questo metodo inizialmente verifica se un tocco effettuato dall'utente è stato eseguito all'interno di un nodo o meno passando il compito di tale controllo al metodo *touchIsInCircle()*; questa funzione riceve in ingresso le coordinate del punto dove l'utente ha effettuato il tocco e le coordinate del nodo insieme al valore del raggio, in seguito i dati vengono processati per mezzo del codice mostrato in figura 2.3 ritornando rispettivamente *true* o *false*.

Questa formula si basa sull'equazione del cerchio per le coordinate cartesiane.

Se il tocco non risulta all'interno del nodo, quindi il risultato della funzione chiamata è *false* si rimane in attesa di un ulteriore tocco da parte dell'utente.

Altrimenti si verifica che la mossa eseguita dall'utente sia una mossa valida. Questa parte viene illustrata nelle sezioni successive.

```
private boolean touchIsInCircle(float x, float y, float centreX, float centreY,
                                float radius) {
    double dx = Math.pow(x - centreX, 2);
    double dy = Math.pow(y - centreY, 2);

    return (dx + dy) < Math.pow(radius, 2);
}
```

Figura 2.3: Implementazione metodo *touchIsInCircle()*.

2.3.1 Mosse forti

Per determinare se una mossa è valida è necessario comprendere prima la differenza tra mosse forti e mosse deboli.

Una mossa forte viene eseguita esclusivamente dall'attaccante e la sua definizione è meglio illustrata nella sezione 4.3 dove vengono esplicate le regole.

Dal punto di vista implementativo per verificare se una mossa compiuta è una mossa forte si chiama il metodo *isStrongMove()* che legge come argomento il nodo del grafo toccato (grafo sinistro o grafo destro) selezionato prima di eseguire la mossa (*startNode*) e il nodo successivamente toccato.

A partire dallo *startNode* si esegue un ciclo su tutti gli archi uscenti di questo nodo e si controlla se il nodo destinazione di uno degli archi su cui si sta ciclando coincide con il nodo toccato. Più nello specifico questo controllo viene effettuato sugli id essendo essi univoci.

Il metodo restituisce un valore booleano che assume la forma di *true* se è stato trovato un id che combacia, *false* in tutti gli altri casi.

2.3.2 Mosse deboli

Una mossa debole può essere eseguita esclusivamente dal difensore e la sua definizione è meglio illustrata nella sezione 4.3.

Dal punto di vista implementativo è stata definita la funzione *isWeakMove()*. Questa funzione riceve in ingresso lo *startNode* (già definito nella sezione precedente), il nodo toccato dall'utente, una variabile booleana *atLeastOneArcOfLastMoveColour*, una mappa con coppia chiave valore $\langle \text{Node}, \text{Boolean} \rangle$ che assume il nome di *visitedNodes*. Un oggetto della classe **Map**[2] che associa una chiave ad un valore. Una mappa non può contenere chiavi duplicate; ogni chiave può essere associata al massimo ad un valore.

La variabile booleana *atLeastOneArcOfLastMoveColour* assume valore *true* quando è già stato visitato un arco con lo stesso colore del colore dell'ultima mossa; altrimenti, assume valore *false*. La mappa *visitedNodes* aggiunge i nodi visitati accostando ad essi il valore *true* se visitati, *false* se non ancora visitati.

Questo metodo ha un carattere ricorsivo.

Innanzitutto si verifica se il colore speciale coincide con il colore dell'ultima mossa effettuata; in questo caso, si cicla sui nodi uscenti dello *startNode* e se il colore di un arco coincide con il colore speciale (che è a sua volta uguale al colore dell'ultima mossa) la funzione ritorna *true*; altrimenti, si prosegue verso il prossimo blocco di codice. Questa parte di codice è necessaria a verificare il caso in cui colore speciale e colore dell'ultima mossa coincidono: in questo caso si può anche rimanere fermi.

Il blocco successivo verifica se la mappa contiene al suo interno lo *startNode*; se questo si verifica la funzione ritorna *true* dopo aver riabilitato l'utilizzo di entrambi i grafi per l'attaccante tramite la funzione *updateValidWeakMove()* (prima era stato disabilitato al tocco il grafo in cui aveva effettuato la mossa l'attaccante).

Altrimenti se la mappa non contiene il nodo toccato si passa al prossimo blocco di codice. Qui si controlla che lo *startNode* sia uguale al nodo toccato; se questo accade si ritorna il valore di *atLeastOneArcOfLastMoveColour* ovvero se è stato trovato almeno un arco del colore dell'ultima mossa ritornerà *true* altrimenti il suo valore di default, ovvero *false*. Se i due nodi non coincidono, si fa un ciclo sugli archi uscenti dello *startNode* e si verifica tramite la variabile *lastColorJustUsed* che viene inizializzata uguale al valore di *atLeastOneArcOfLastMoveColour* se il colore dell'arco ispezionato è uguale al colore dell'ultima mossa: in questo caso la variabile *lastColourJustUsed* assume il valore *true* altrimenti si continua a ciclare.

In seguito avviene la chiamata ricorsiva passando come argomenti il nodo destinazione dell'arco, il nodo toccato, la variabile *lastColourJustUsed* e la mappa *visitedNodes*; su questa chiamata si fa un controllo e se essa restituisce *true* si aggiunge alla mappa il nodo di destinazione dell'arco associato al valore *true* e si richiama il metodo *updateValidWeakMove()*, ritornando il valore *true*. Questo blocco di codice permette di percorrere tutti gli archi percorribili a partire dal nodo di partenza fino a che non si trova il percorso contenente il nodo toccato.

Se viene toccato un nodo che non rientra nei percorsi degli archi percorribili, il metodo ritornerà *false*.

2.3.3 Controllo mossa valida

Al fine di verificare che una mossa eseguita da un giocatore sia corretta, il primo step è verificare a chi appartiene attualmente il turno. Per fare ciò si legge una casella di testo contenente tale informazione che a sua volta prende l'informazione leggendola dal database.

Se è il turno dell'attaccante si chiama la funzione *isStrongMove()* e a seconda del valore ritornato da questa funzione si rende valida la mossa o meno.

Analogamente nel turno del difensore si procede alla stessa maniera richiamando il metodo *isWeakMove()*.

2.3.4 Controlli fine partita

Le condizioni di fine partita sono tre e sono illustrate nella sezione 4.3.

A livello di codice se una mossa è stata ritenuta valida si passa il controllo al metodo *controlConfiguration()*. Questo metodo controlla che una configurazione non si sia già verificata in precedenza; se così fosse, la partita terminerebbe.

Questo metodo utilizza un **HashSet**<Pair>. Un oggetto della classe **HashSet** non garantisce l'ordine di iterazione sull'insieme; in particolare, non garantisce che l'ordine rimanga costante nel tempo[1].

La classe *Pair* è una classe del folder *models* che sostanzialmente memorizza due nodi, uno del grafo di sinistra e uno del grafo di destra.

Dunque se al termine di una mossa la nuova configurazione, ovvero la nuova coppia di nodi, è già presente all'interno dell'insieme dunque è già stata visitata, la partita termina.

Questo metodo garantisce il controllo di una delle tre condizioni di termine della partita.

Le altre due condizioni sono analoghe tra di loro e dal punto di vista implementativo sono riassunte in un unico metodo chiamato *possibleMoves()*. Questo metodo legge come parametro lo *startNode* e al suo interno verifica se sono presenti ancora mosse possibili per l'attaccante simulando una mossa dell'attaccante stesso. Più nel dettaglio vengono selezionati il nodo corrente di ogni singolo grafo e si verifica se almeno uno dei due nodi tra i suoi archi uscenti ha un arco diverso da null; se questo non avviene significa che non ci sono più mosse disponibili e dunque dopo un aggiornamento del valore *gameInProgress* all'interno del database la partita termina.

Il discorso è analogo per il difensore.

2.4 Frontend

2.4.1 Material Design

Il **Material Design** [8] è una guida completa per il design visivo, del movimento e dell'interazione dei dispositivi.

Per rendere l'esperienza utente ottimale e per rispettare le indicazioni Android, nello sviluppo dell'applicazione è stato utilizzato il Material Design seguendo le linee guida definite, dalla tipografia alle diverse views utilizzate.

2.4.2 Componenti Sign Up Activity

Dal punto di vista dell'interfaccia grafica l'activity *SignUp* presenta una serie di **TextView** ed **EditText** con lo scopo rispettivamente di informare l'utente sull'informazione richiesta che esso dovrà inserire e lo spazio necessario per eseguire tale operazione.

2.4.3 Componenti Login Activity

Questa activity, oltre a presentare le solite **TextView** ed **EditText**, mostra una *checkbox* ovvero un componente che consente all'utente di selezionare un'opzione. In questo caso specifico questa checkbox permette all'utente di memorizzare le proprie credenziali di accesso.

All'interno dell'activity è presente anche una *bottom sheet*, ovvero un modello di interfaccia utente che viene utilizzato per visualizzare una *view* che si apre dal basso. Nell'applicazione ha funzione di reset della password in caso di dimenticanza.

2.4.4 Componenti Home Fragment

La schermata principale (*Home*) costituisce l'unico *Fragment* presente all'interno del progetto. Qui sono presenti due *cards* che forniscono all'utente informazioni sul proprio score personale e sulle partite attualmente attive.

Ciascuna card è una **MaterialCardView** che contiene al proprio interno **TextView** e **Button**. Un elemento presente all'interno di ciascuna card che poi ricorre anche in altri componenti all'interno delle varie interfacce è il **separator**, ovvero una semplice *view* dalla larghezza di un solo 1dp.

In questo fragment troviamo anche un **FloatingActionButton**[9], ossia un pulsante utilizzato per un tipo speciale di azione. Un **FAB** si distingue per un'icona circolare.

Questo pulsante circolare permette di condividere la propria esperienza con l'applicazione sui social.

2.4.5 Componenti Matchmaking Room Activity

Questa activity è composta semplicemente da un **button** e un **ListView** posizionati all'interno di un *Relative Layout*.

Una list view mostra un insieme di item che possono essere "scrollati" verticalmente e dove ogni item è posizionato immediatamente sotto l'item che lo precede nella lista.

Il componente **ListView** è stato implementato anche nell'activity *ActivePlayers* per mostrare i giocatori attivi in quel momento.

2.5 Canvas

Al fine di disegnare i due grafi che compongono il tavolo da gioco dove si sfidano attaccante e difensore è stata utilizzata la classe **Canvas**.

Questa classe contiene le chiamate alle funzioni che permettono di disegnare.

Il metodo *onDraw()* riceve come argomento un oggetto di tipo *Canvas*, che rappresenta la "tela" su cui disegnare gli oggetti, e al suo interno chiama due funzioni che hanno il compito di disegnare i nodi e gli archi dei singoli grafi.

Per quanto riguarda i nodi, questi vengono disegnati tramite il metodo *drawCircle()* che legge quattro parametri:

1. la coordinata **x** del centro del nodo che sta per essere disegnato (di tipo *float*);
2. la coordinata **y** del centro del nodo che sta per essere disegnato (di tipo *float*);
3. il **raggio** del nodo che sta per essere disegnato (di tipo *float*);
4. un oggetto di tipo **Paint**.

Quest'ultimo oggetto è il risultato della chiamata ad una funzione che riceve come parametro il colore del nodo e ritorna un oggetto con le informazioni sullo stile del nodo.

Gli archi invece sono disegnati tramite il metodo *drawLine()* che riceve come parametri 5 argomenti:

1. la coordinata **x** del punto di partenza della linea;
2. la coordinata **y** del punto di partenza della linea;
3. la coordinata **x** del punto di arrivo della linea;
4. la coordinata **y** del punto di arrivo della linea;
5. un oggetto di tipo **Paint** esattamente come nel metodo *drawCircle()*.

Come punto di arrivo e punto di fine sono stati utilizzati come riferimento i nodi, facendo dei calcoli utilizzando il centro e il raggio.

2.5.1 Directed Graph

I grafi rappresentati tramite la classe *Canvas* sono contenuti nella classe *astratta* **DirectedGraph**.

Questa classe definisce i metodi principali, quali getter, setter e costruttore, quest'ultimo utilizzato per la rappresentazione dei grafici.

Tra gli attributi di questa classe sono presenti un array di oggetti della classe *Node* (sezione 2.5.2) e un array di oggetti della classe **Edges** (sezione 2.5.3). All'interno di questa classe sono implementati 3 metodi che restituiscono tutti e 3 un oggetto di tipo **Paint** che viene utilizzato rispettivamente per la rappresentazione grafica rispettivamente dei nodi, degli archi e della punta della freccia che termina l'arco. In particolare, questi oggetti definiscono il colore, lo spessore e lo stile dei tre elementi indicati precedentemente.

La classe *DirectedGraph* è una classe astratta in quanto definisce senza implementazione i seguenti metodi:

- *drawGraph()*;
- *drawNodes()*;
- *drawEdges()*;

Tutti questi metodi ricevono come parametro un oggetto di tipo Canvas e sono implementati nelle classi figlie, **DirectedGraphLeft** e **DirectedGraphRight**.

Per disegnare i nodi nelle posizioni corrette, viene eseguito un ciclo sull'attributo di classe *nodes* a cui si ha accesso tramite il getter e con il metodo *drawCircle()* chiamato sull'oggetto canvas si disegna nodo per nodo. Nello specifico, questo metodo legge come parametro la posizione del nodo (coordinate *x* e *y*), il raggio e il colore del nodo.

2.5.2 Node

La classe **Node** rappresenta un componente insieme alla classe **Edge** necessaria per la rappresentazione grafica dei Canvas.

Oltre alla definizione dei getter e dei setter, questa classe è basata sul costruttore che legge come parametro tutti gli attributi definiti e li attribuisce tramite i setter.

Gli attributi *id*, *root* (che definisce se un nodo è la radice del grafo o meno), *toLeft* (che definisce se un nodo si trova a sinistra rispetto al nodo genitore o meno), *toRight* (stesso principio di *toLeft*), *leftTable* (che definisce se un nodo appartiene al grafo di sinistra o meno) e *color* vengono impostati tramite i setter senza alcuna manipolazione.

D'altro canto gli attributi *x* e *y* che rappresentano rispettivamente l'ascissa e l'ordinata del nodo sono manipolati: se il nodo è un nodo radice, allora il punto *x* dovrà essere messo al centro del layout cui appartiene utilizzando il metodo *getDisplayMetrics()* della classe Resources. La coordinata *y* non viene manipolata.

Altrimenti se il nodo non è un nodo radice bisogna distinguere due casi:

1. il nodo si trova alla sinistra rispetto al nodo genitore;
2. il nodo si trova alla destra rispetto al nodo genitore.

In entrambi i casi la coordinata *x* viene manipolata; nel primo caso viene spostata verso sinistra di un valore fisso che assume il nome di *shiftHorizontal*, nel secondo caso viene spostato dalla parte opposta sempre dello stesso valore precedentemente citato.

E' possibile che un nodo non rientri in nessuno dei due casi sopracitati: in questo caso la coordinata *x* non viene manipolata rispetto alla sua lettura come parametro del costruttore.

In tutti i casi la coordinata *y* subisce una variazione di un valore fisso che assume il nome di *shiftVertical* verso il basso.

2.5.3 Edge

La classe **Edge** definisce semplicemente i getter, i setter e il costruttore. All'interno di questi metodi le variabili non vengono manipolate.

Gli attributi di questa classe sono:

- *id* che rappresenta un numero univoco che identifica l'arco;
- *source* un oggetto della classe Node che rappresenta il nodo di partenza dell'arco;
- *destination* un oggetto della classe Node che rappresenta il nodo di arrivo dell'arco;
- *color* che rappresenta il colore dell'arco;
- *toBottom* che indica se l'arco ha direzione nord-sud;

Capitolo 3

Partita in tempo reale

3.1 Lettura file di configurazione

Al fine di poter creare nuove configurazioni di grafi da navigare durante la partita è stato utilizzato un meccanismo di lettura di un file di testo per generare i grafi.

Il file di testo, chiamato **config**, è posizionato nella cartella **assets**; tramite l'utilizzo della classe **InputStream** che permette di aprire il file di testo e della classe **BufferedReader** che dà la possibilità di leggere il contenuto del file si è reso possibile creare nuove configurazioni senza dover modificare il codice sorgente.

All'interno del file sono descritte le specifiche da seguire per poter creare una nuova configurazione. Questo file deve essere modificato seguendo le specifiche e i pattern proposti; una eventuale modifica non corretta può portare a danneggiare le configurazioni già presenti.

3.2 Interazione con Realtime Database

3.2.1 Preparazione

Per preparare la sfida tra i due avversari viene inviata al database di Firebase una struttura di base che viene aggiornata durante la partita in base alle mosse degli sfidanti.

Una volta creata la partita dall'utente viene creato all'interno del nodo **rooms** una stanza che prende il nome del giocatore che l'ha creata, come in figura 3.1.

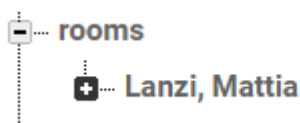


Figura 3.1: Nodo all'interno del database che rappresenta stanza creata da un giocatore.

I nodi figli di ogni singola stanza sono i seguenti:

- *Player 1* che contiene a sua volta ulteriori nodi figli;
- *gameInProgress* inizializzato a **true**;
- *lastMoveColour* inizializzato ad una stringa vuota;
- *leftGraph* che contiene nodi figli;
- *noMove* inizializzato a **false**;

- *rightGraph* con lo stessa logica del nodo *leftGraph*;
- *show* inizializzato a **true**;
- *specialColour*;
- *turnOf* inizializzato alla stringa "attacker".

Il nodo padre, che indica il nome della stanza, acquisisce un nodo figlio chiamato *Player 2* nel momento in cui un avversario si unisce alla stanza. Questo nodo segue la stessa logica del nodo *Player 1*. Questi nodi forniscono le informazioni sul nome dell'utente e il suo ruolo, attaccante o difensore.

Nel momento in cui il secondo giocatore si unisce alla partita, quindi nei momenti che ancora precedono l'inizio della stessa, il nodo *show* assume valore **false**. Il nodo *show* se ha valore *true* permette di mostrare la stanza della partita nella ListView, indicando che manca ancora un avversario. Quando questo nodo assume valore *false* la stanza della partita non sarà più visibile all'interno della ListView.

Il nodo *specialColour* assume il valore di una stringa corrispondente al colore di uno degli archi utilizzati all'interno del gioco (rosso, blu, verde, nero). Anch'esso non viene modificato durante lo svolgimento della partita. La figura 3.2 rappresenta un nodo completo prima dell'inizio della partita.

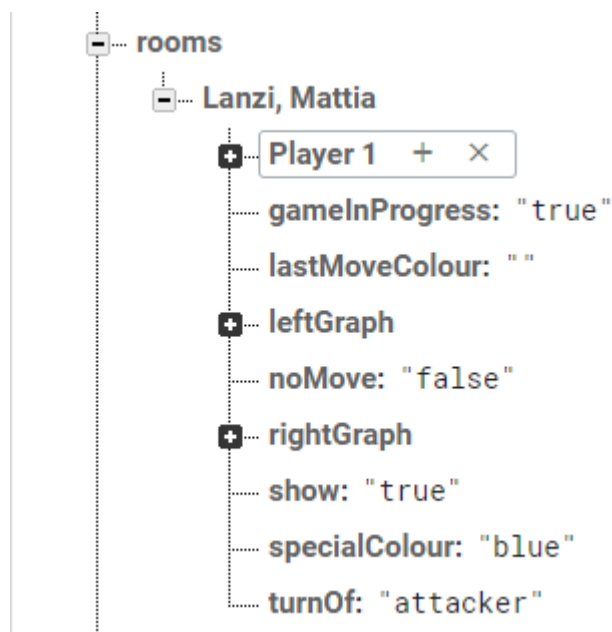


Figura 3.2: Nodo che contiene informazioni preparatorie per la partita.

3.2.2 Svolgimento

Durante la partita i nodi *leftGraph* e *rightGraph* vengono aggiornati costantemente.

Prendendo come esempio il nodo *leftGraph* (il meccanismo per il grafo di destra è identico), esso contiene nodi figli che rappresentano tutti i nodi che compongono il grafo. A loro volta ogni nodo contiene due attributi: *colour* e *selected*. Questi due nodi vengono aggiornati ad ogni mossa dei giocatori, cambiando il loro valore se selezionati (*colour* assume valore blue e *selected* assume valore true).

Il nodo *lastMoveColour* assume il valore del colore dell'arco percorso dal giocatore in seguito

all'ultima mossa compiuta.

Il nodo *turnOf* cambia ad ogni turno: inizialmente prende valore di "attacker" e ad ogni mossa eseguita, se valida, cambia il proprio valore passando a "defender" e viceversa.

Tutte queste informazioni sono contenute nella casella sottostante i tavoli di gioco.

In aggiunta è presente anche un attributo *enabled* che, quando assume valore **false** rende il grafo inutilizzabile, come mostrato in figura 3.3.

Infine il nodo *gameInProgress* ha lo scopo di terminare la partita non appena esso assume valore **false**.

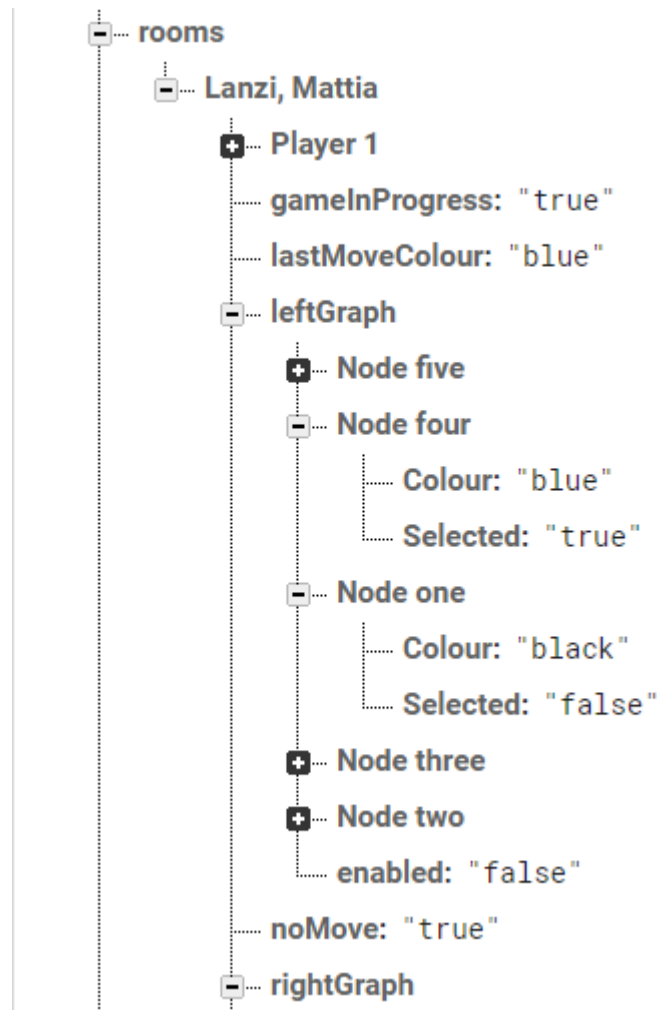


Figura 3.3: Nodo riguardante partita che ha subito variazioni.

Capitolo 4

Manuale utente

4.1 Manuale d'uso

L'applicazione sviluppata è costituita dalle seguenti schermate elencate con questi titoli:

- Login;
- Registrazione;
- Home;
- Il mio profilo;
- Gioca ora!;
- FAQ;
- Impostazioni;
- Segnala un problema.

L'insieme di queste schermate forma l'applicazione che risulta navigabile tramite un menù laterale in tutte le sue schermate, fatta eccezione per le schermate di Login e Registrazione. La funzione principale di questa applicazione è affrontare un avversario ad un gioco basato sul **principio della bisimulazione**.

4.2 Guida all'uso

4.2.1 Registrazione

La schermata iniziale dell'applicazione **Bisimulazione** è rappresentata dalla schermata di **Login** (Figura 4.1). Questa schermata permette di autenticarsi e in seguito di utilizzare l'applicazione. L'autenticazione è possibile solo se già registrati. Se ancora non si è registrati, è necessario effettuare la registrazione.

Per poter effettuare la registrazione bisogna premere nel link a piè pagina contrassegnato dalla scritta *Clicca qui*, immediatamente successivo alla scritta *Non sei ancora registrato?*.

A questo punto comparirà la schermata di **Registrazione**; qui basterà inserire i propri dati nelle caselle di testo dedicate e infine cliccare su *Registrati*. Una volta rilasciato il tasto *Registrati*, i dati saranno inviati alla piattaforma Firebase e verrà aperta la schermata di **Login**.

A mobile application sign-up screen with a dark blue header bar containing a back arrow and the text "Sign up". The form consists of several text input fields with placeholder text, followed by a "SIGN UP" button.

← Sign up

Insert your name
Your first name

Insert last name
Your last name

Insert your email address
example@mail.com

Insert a username
Username

Insert password

Confirm your password

SIGN UP

Figura 4.1: Schermata di registrazione.

4.2.2 Login

Di default quando si esegue l'applicazione viene mostrata la schermata di **Login** (Figura (a)). Questa schermata ha come funzione principale quella di permettere agli utenti di autenticarsi in maniera semplice e veloce al fine di poter usufruire dell'applicazione. Per autenticarsi è necessario inserire il proprio indirizzo mail utilizzato in fase di registrazione e la password scelta sempre durante la registrazione. A questo punto dovrà essere premuto il pulsante *Login* e, una volta rilasciato, si potranno verificare due casi:

1. l'autenticazione avviene **correttamente**;
2. l'autenticazione **fallisce**.

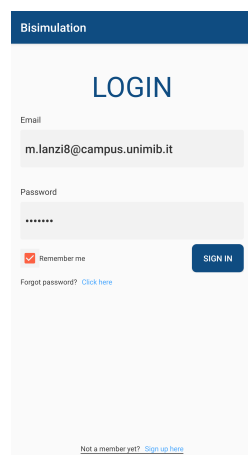
Nel primo caso viene mostrato un messaggio di avvenuta autenticazione e successivamente comparirà la schermata principale con titolo **Home** (o Bisimulazione).

Nel secondo caso viene stampato a video un messaggio di errore che non permette il passaggio alla schermata successiva, mantenendo l'utente sulla schermata corrente. I motivi della mancata autenticazione possono essere molteplici a partire da un input scorretto fornito dall'utente.

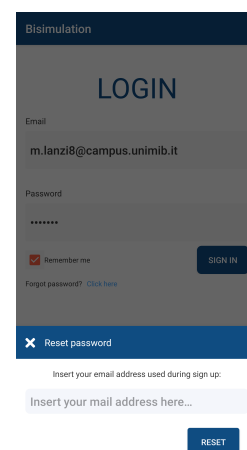
Nel caso in cui si volessero memorizzare i propri dati di autenticazione è possibile spuntare con un flag la casella posizionata alla sinistra della scritta *Ricordami*: in questo modo ogni volta che verrà eseguita l'applicazione non bisognerà inserire di nuovo le proprie credenziali di accesso, bensì saranno già memorizzate e basterà semplicemente cliccare sul pulsante *Login*.

Inoltre se è stata dimenticata la propria password si può toccare il link descritto dal testo *Clicca qui* immediatamente successivo alla scritta *Password dimenticata?*. A questo punto dal basso verrà mostrata una **bottom sheet** che invita l'utente ad inserire il proprio indirizzo mail utilizzato durante la registrazione; una volta digitato l'indirizzo mail e premuto il tasto *Reset*, verrà inviata una mail all'indirizzo mail appena inserito con una nuova password da utilizzare al seguente accesso.

Con il termine bottom sheet [12] (rappresentato in figura (b)) si intende una schermata aggiuntiva utilizzata principalmente nel mondo mobile. Una bottom sheet standard, come quella implementata, mostra un contenuto integrativo rispetto a quanto mostrato nella schermata principale. Mentre l'utente utilizza la schermata principale, la bottom sheet non è visibile.



(a) Schermata di Login.



(b) Bottom sheet utilizzata per reimpostare la password.

4.2.3 Home

La schermata principale dal titolo **Home** (o **Bisimulazione**) (Figura 4.3) presenta due cards, tre pulsanti e il menù di navigazione.

Le due cards contengono informazioni riguardo l'utente, i giocatori e le partite in corso. In particolare la prima card dal titolo *Statistiche personali* mostra le vittorie ottenute dall'utente sotto la scritta *Numero di vittorie* e le sconfitte accumulate sotto la scritta *Numero di sconfitte*. Inoltre questa card contiene un pulsante che presenta la dicitura *Scopri di più* che se premuto apre la schermata **Il mio profilo**.

La seconda card dal titolo *Giocatori attivi* mostra il numero di giocatori attivi e le stanze da gioco attive al momento attuale.

Al centro sono presenti due pulsanti dal titolo: *Gioca ora!* e *Giocatori attivi*. Il pulsante *Gioca ora!* se cliccato porta alla pagina **Stanza pre-partita**, mentre l'altro pulsante porta alla pagina **Giocatori attivi**.

In basso a destra è presente anche un bottone circolare che permette di condividere un messaggio con i propri contatti su differenti social networks: una volta cliccato apparirà un messaggio che invita a condividere l'utilizzo dell'applicazione con i propri contatti. Se si clicca sul tasto *Condividi*, si potrà proseguire condividendo il messaggio, altrimenti se si tocca su *Chiudi* la finestra verrà chiusa.

Infine questa schermata presenta lateralmente un menù di navigazione; per aprirlo basterà schiacciare sull'apposita icona e saranno mostrati le diverse schermate presenti all'interno dell'applicazione che possono essere raggiunte con un semplice tocco sull'item desiderato.

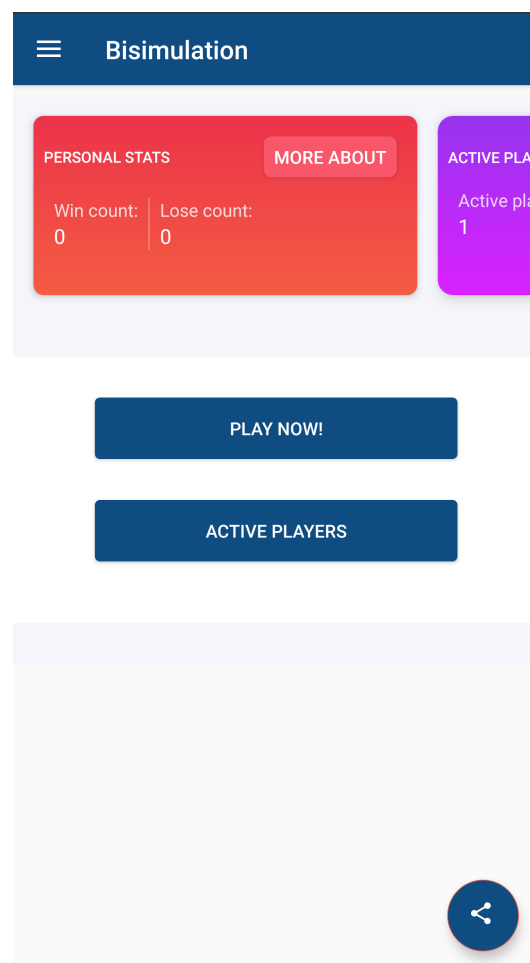


Figura 4.3: Schermata principale.

4.2.4 Il mio profilo

La schermata **Il mio profilo** presenta due sezioni: la prima sezione riguardante le informazioni personali dell'utente dove è possibile trovare le generalità dell'utente che ha effettuato il login all'interno dell'applicazione quali nome, cognome e indirizzo mail utilizzato al momento della registrazione; nella seconda sezione intitolata *Statistiche* è mostrato il numero di vittorie e il numero di sconfitte accumulate dal giocatore.

4.2.5 Giocatori attivi

Questa pagina mostra semplicemente una lista con tutti i giocatori attualmente attivi (ovvero che sono impegnati in una partita oppure sono in attesa di iniziarne una nuova).

4.2.6 FAQ

All'interno dell'applicazione è presente anche una pagina dedicata alle domande più frequenti (Frequently Asked Question) accessibile tramite menù di navigazione. Questa pagina presenta alcune domande da cui si può ricavare la risposta semplicemente toccando sulla domanda a cui si è interessati. A questo punto si aprirà verso il basso sotto la domanda un testo contenente la risposta alla domanda selezionata. La figura 4.4 mostra la pagina appena descritta.

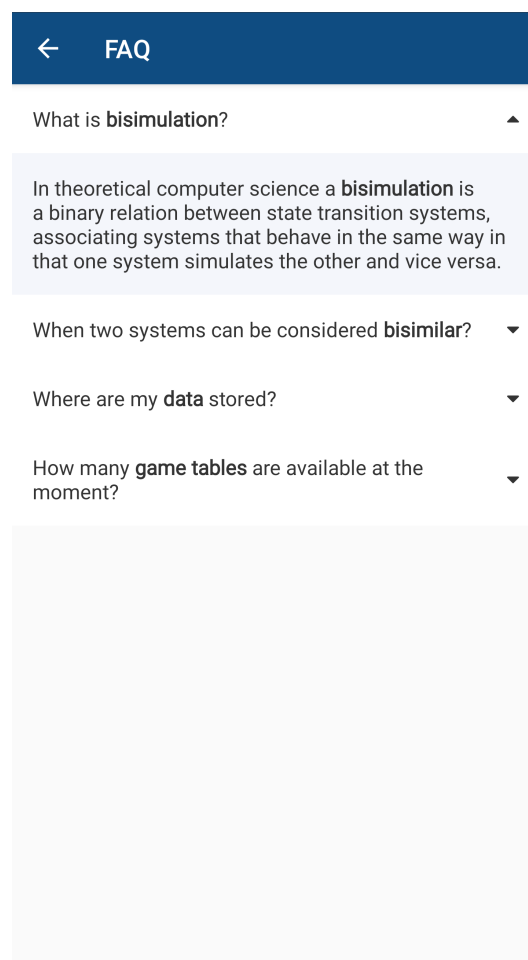


Figura 4.4: Schermata FAQ.

4.2.7 Impostazioni

La schermata **Impostazioni** permette esclusivamente di cambiare la lingua dell'applicazione da italiano a inglese o viceversa. Per eseguire questa operazione è necessario cliccare sulla scritta **Seleziona lingua...**; in seguito verrà mostrata la possibilità di scegliere tra *Italiano* o *Inglese*: per effettuare la propria scelta basterà toccare sulla lingua desiderata e subito si tornerà alla schermata principale con la nuova lingua selezionata.

4.2.8 Segnala un problema

Dal menù di navigazione è possibile raggiungere anche la pagina **Segnala un problema** (Figura 4.5). Qui è possibile segnalare eventuali problemi riscontrati durante l'utilizzo dell'applicazione o fornire dei suggerimenti per migliorare l'applicazione; per fare ciò è necessario toccare all'interno della casella di testo dove è presente la scritta *Inserisci qui il testo* e scrivere il messaggio che si intende recapitare. Una volta terminato di scrivere il messaggio si deve cliccare sul tasto *Invia* che, una volta rilasciato, mostrerà un messaggio di avvenuta spedizione del messaggio di testo.

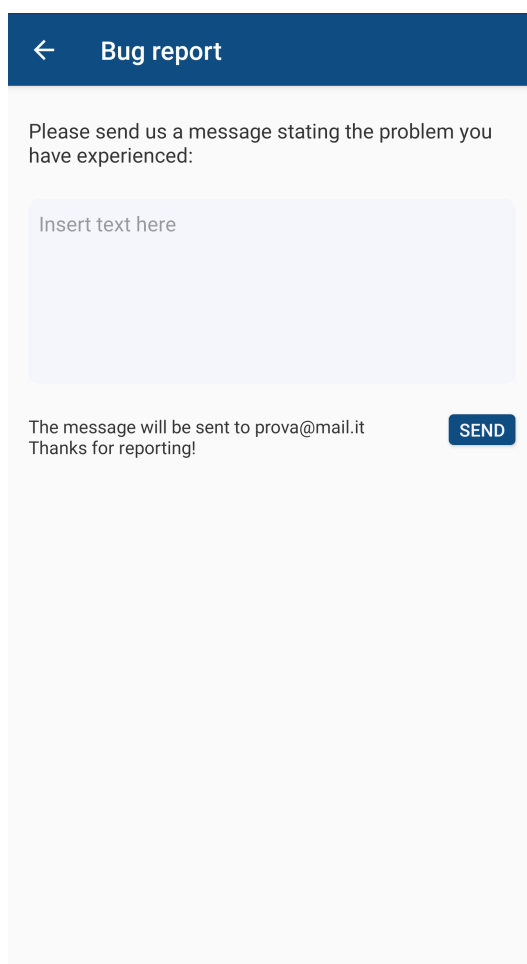


Figura 4.5: Schermata segnala un problema.

4.2.9 Stanza pre-partita

La pagina **Stanza pre-partita** (Figura 4.6) mostra tutte le partite create da altri giocatori che ancora stanno aspettando il proprio avversario. Per poter partecipare alla partita creata da un altro giocatore si deve schiacciare su un elemento della lista con scritto *Stanza di "Cognome", "Nome"*; a questo punto inizierà la partita in una nuova schermata contro l'avversario selezionato.

In questa schermata è possibile anche creare una stanza propria cliccando sul pulsante *Crea stanza*; appena viene cliccato questo pulsante si apre una nuova schermata con la partita.

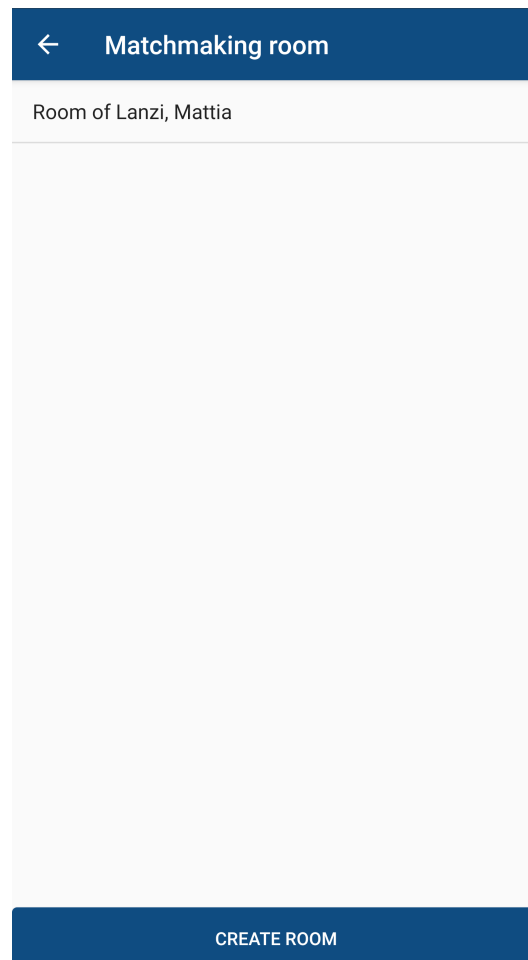


Figura 4.6: Schermata creazione stanza da gioco.

4.2.10 Gioco

La partita tra i due avversari avviene nella schermata **Gioco** (Figura 4.7). Questa schermata si compone di due tavoli: *Tavolo sinistro* e *Tavolo destro*.

All'interno di ogni tavolo è presente un **grafo**. Il gioco consente di eseguire due azioni: toccare un nodo di uno dei due grafi oppure toccare il pulsante *Stai fermo* che si trova in fondo alla schermata. Se viene toccato un nodo e la mossa eseguita dal giocatore è valida allora il nodo toccato si colorerà di blu, così come era colorato precedentemente il nodo selezionato.

Se il giocatore decide di non muoversi e questa mossa è consentita dalle regole del gioco, allora cambierà il turno senza che sui grafi avvenga alcun cambiamento.

In entrambi i casi, se la mossa eseguita rispetta le regole del gioco, viene aggiornato il turno di gioco e il colore dell'ultima mossa. In caso contrario, verrà mostrato un messaggio che suggerisce al giocatore che ci sono ancora mosse disponibili; se questo non fosse vero, allora la partita terminerebbe mostrando un messaggio di *Fine partita* che riporta l'utente alla schermata principale. Sotto i due tavoli è presente una tabella che contiene informazioni utili riguardanti lo svolgimento del gioco; in particolare la casella fornisce informazioni su:

- attaccante;
- difensore;

- turno del giocatore;
- colore dell'ultima mossa;
- colore speciale.

Le regole del gioco possono essere trovate nella sezione 4.3 "Regole del gioco".

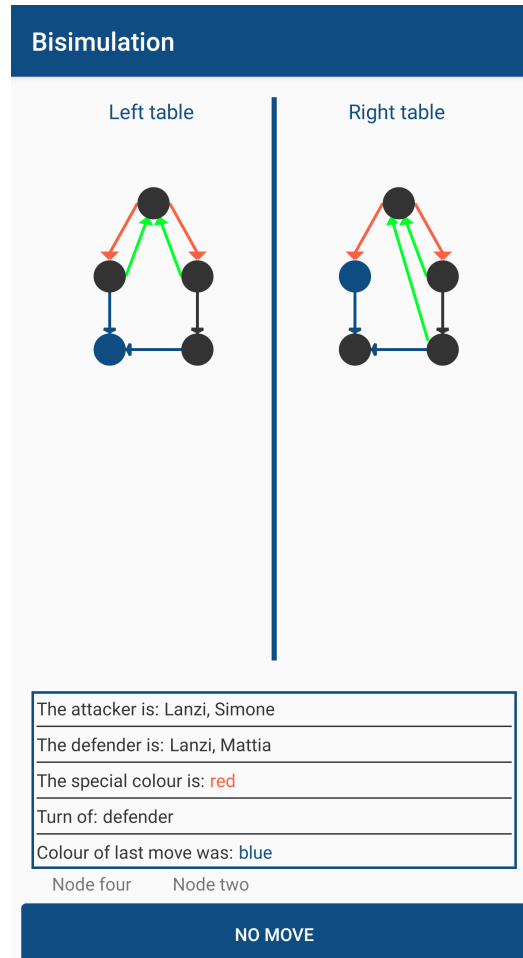


Figura 4.7: Schermata di gioco.

4.3 Regole

Il gioco si svolge tra due giocatori, che chiameremo Attaccante (A) e Difensore (D).

Il terreno di gioco è formato da due grafi orientati, detti rispettivamente *tavolo sinistro* e **tavolo destro**. Gli archi dei due grafi sono colorati di rosso, verde, blu e nero. All'inizio della partita è proposto un colore che definiremo *colore speciale*.

Una partita è composta da una sequenza di turni. Ad ogni turno muove l'attaccante che sceglie su quale tavolo compiere la mossa; e il difensore deve rispondere muovendo sul tavolo opposto. Ci sono due tipi di mosse: mosse forti e mosse deboli. L'attaccante compie sempre mosse forti, il difensore mosse deboli. Una mossa forte consiste nello spostarsi dallo stato in cui si trova a uno stato raggiungibile percorrendo uno ed un solo arco del grafo. Diremo che il colore della mossa è il colore dell'arco percorso. Una mossa debole di colore x consiste nello spostarsi lungo una sequenza di archi, uno dei quali di colore x , mentre gli altri devono essere tutti del colore speciale. C'è un'eccezione: una mossa debole del colore speciale consiste in una sequenza di zero

o più archi, tutti del colore speciale.

Prendendo come esempio la figura 4.8, se il difensore si trova sullo stato 3, una mossa debole di colore azzurro consiste nello spostare la pedina sullo stato 4, passando per lo stato 5; a partire dallo stato 1, una mossa debole rossa può consistere nello spostare la pedina sullo stato 2, oppure sullo stato 3, oppure sullo stato 5 (passando per lo stato 3).

A partire dallo stato 3 del tavolo di sinistra, una mossa forte grigia consiste nello spostare la pedina sullo stato 5; una mossa debole grigia può consistere nello spostare la pedina sullo stato 5, oppure nel lasciare la pedina sullo stato 3.

La partita può terminare in tre modi:

1. Dopo una mossa dell'attaccante, il difensore non può rispondere con una mossa dello stesso colore. L'attaccante vince.
2. Dopo una mossa del difensore, l'attaccante non ha mosse a disposizione su nessuno dei due tavoli. Il difensore vince.
3. Dopo una mossa del difensore, si torna a una configurazione già visitata nella stessa partita al termine di un turno. Il difensore vince.

Sempre in riferimento alla figura 4.8, data la coppia $(1, 1)$ come configurazione iniziale, l'attaccante può scegliere fra quattro mosse iniziali, tutte rosse: può muovere la pedina su 2 o su 3 sul tavolo di sinistra, oppure su 2 o su 3 sul tavolo di destra. Supponiamo che scelga di muovere su 3 sul tavolo di destra; la nuova configurazione sarà $(1, 3)$. Il difensore può scegliere fra tre mosse deboli sul tavolo di sinistra: da 1 a 2, da 1 a 3, da 1 a 5. Le nuove configurazioni nei tre casi saranno, rispettivamente, $(2, 3)$, $(3, 3)$, e $(5, 3)$.

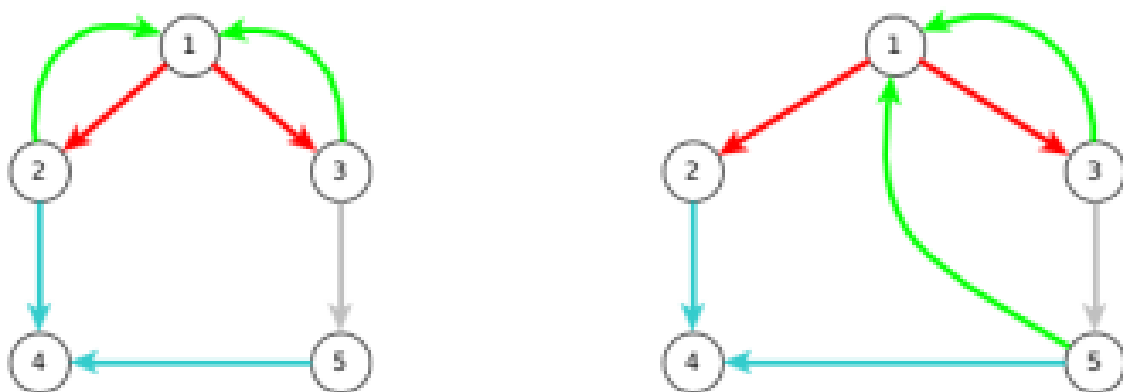


Figura 4.8: Esempio terreno di gioco.

Capitolo 5

Conclusioni

In questa relazione di stage viene affrontato lo sviluppo di un'applicazione che mira ad essere di supporto per l'insegnamento didattico del principio della bisimulazione.

La soluzione offerta è dunque un gioco che permette di mettere in pratica le conoscenze acquisite a livello teorico su tale argomento contro un avversario che può essere più o meno competente in materia. Con questa soluzione si cerca di offrire una possibilità di testare le proprie competenze possibilmente divertendosi.

L'esperienza di stage interna all'università si è rivelata un'importante occasione per accrescere le mie competenze in ambito di sviluppo Android, imparando ad utilizzare nuove librerie come le Canvas, aggiungendo al mio bagaglio personale nuovi meccanismi implementativi e perfezionando le abilità pregresse.

Capitolo 6

Sviluppi futuri

In questo capitolo vengono proposti dei possibili sviluppi futuri che possono rendere l'applicazione più completa e più fruibile.

6.1 Scelta partita casuale

Un primo aspetto che può rendere l'applicazione più facilmente utilizzabile è la possibilità di giocare una partita contro un avversario casuale. L'aggiunta di questa componente non rende più obbligatoria la scelta da parte dell'utente dell'avversario contro cui confrontarsi ma apre la possibilità a giocare una partita contro un avversario scelto in maniera randomica.

6.2 Avversario CPU

Attualmente è possibile affrontare esclusivamente un avversario reale in una partita online tramite l'interazione e le continue richieste con il database. In futuro si può sviluppare un'intelligenza artificiale da affrontare in mancanza di connessione Internet o semplicemente per testarsi in modalità singola.

6.3 Animazioni

Uno sviluppo futuro può riguardare la **UI** (*user interface*) ovvero l'interfaccia grafica. Si possono aggiungere animazioni più intrattenenti quando un utente compie una mossa e dunque c'è un passaggio di stato all'interno del grafo scelto.

6.4 Impostazioni

L'activity riguardante le impostazioni attualmente consta della sola funzionalità di cambio lingua. In futuro è programmata l'implementazione di nuove funzionalità in questa pagina quali la possibilità di cambiare tema, aggiungere un collegamento sulle info del progetto e sui servizi di terze parti.

Bibliografia

- [1] Documentazione classe hashset. URL: <https://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html>.
- [2] Documentazione classe map. URL: <https://docs.oracle.com/javase/8/docs/api/java/util/Map.html>.
- [3] Documentazione metodo `createuserwithemailandpassword()`. URL: [https://firebase.google.com/docs/reference/android/com/google/firebase/auth/FirebaseAuth#createUserWithEmailAndPassword\(java.lang.String,%20java.lang.String\)](https://firebase.google.com/docs/reference/android/com/google/firebase/auth/FirebaseAuth#createUserWithEmailAndPassword(java.lang.String,%20java.lang.String)).
- [4] Documentazione metodo `oncreate()`. URL: [https://developer.android.com/reference/android/app/Activity#onCreate\(android.os.Bundle,%20android.os.PersistableBundle\)](https://developer.android.com/reference/android/app/Activity#onCreate(android.os.Bundle,%20android.os.PersistableBundle)).
- [5] Documentazione metodo `setvalue()`. URL: <https://firebase.google.com/docs/database/admin/save-data>.
- [6] Documentazione metodo `signinwithemailandpassword()`. URL: [https://firebase.google.com/docs/reference/android/com/google/firebase/auth/FirebaseAuth#signInWithEmailAndPassword\(java.lang.String,%20java.lang.String\)](https://firebase.google.com/docs/reference/android/com/google/firebase/auth/FirebaseAuth#signInWithEmailAndPassword(java.lang.String,%20java.lang.String)).
- [7] Firebase realtime database. URL: <https://firebase.google.com/docs/database>.
- [8] Material design for android. URL: <https://developer.android.com/guide/topics/ui/look-and-feel>.
- [9] Regular floating action button. URL: <https://material.io/components/buttons-floating-action-button/android#regular-fabs>.
- [10] Report ericsson utilizzo smartphone android. URL: <https://www.ericsson.com/en/mobility-report/reports/june-20211>.
- [11] Shared preferences. URL: <https://developer.android.com/reference/android/content/SharedPreferences>.
- [12] Standard bottom sheet. URL: <https://material.io/components/sheets-bottom>.
- [13] Luca Aceto, Anna Ingolfsdottir, Kim G.Larsen, and Jiri Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, 2007.
- [14] Walter Savitch and Daniela Micucci. *Programmazione di base e avanzata con Java*. Pearson, 2014.