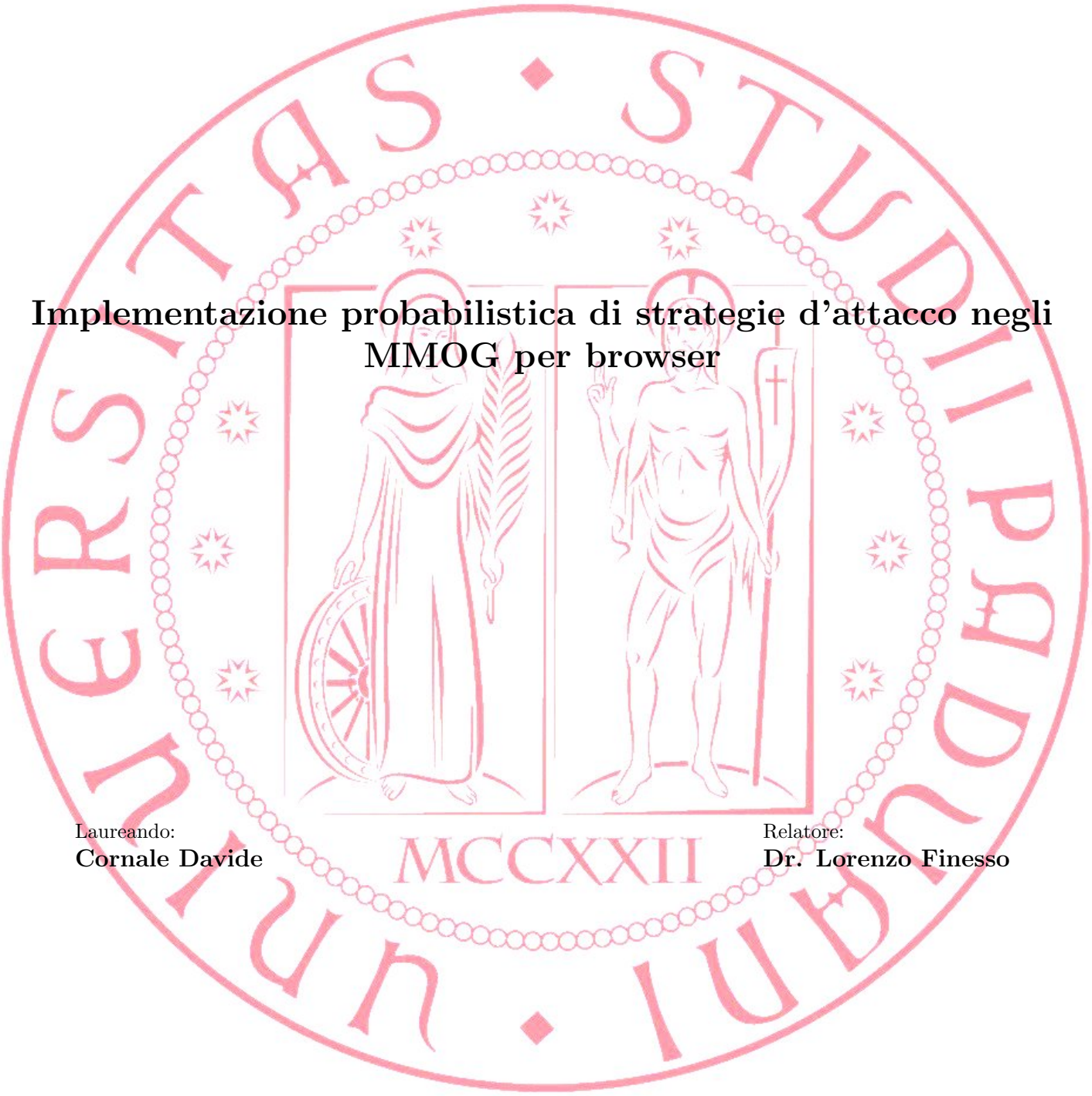
Università degli Studi di Padova

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Corso di Laurea in Ingegneria dell'Informazione

Implementazione probabilistica di strategie d'attacco negli MMOG per browser



Laureando:
Cornale Davide

Relatore:
Dr. Lorenzo Finesso

Anno Accademico 2011-2012

Un'idea che non sia pericolosa, è indegna di chiamarsi idea.
Oscar Wilde

Sommario

In questa tesi si intende descrivere una categoria di videogiochi, noti come *massively multiplayer online game* (MMOG), che in questi ultimi anni si sta espandendo in modo considerevole. In particolare concentreremo l'attenzione sulla sottocategoria dei *browser based massively multiplayer online game* (BBMMOG), analizzando la piattaforma *Noirblack* ed i relativi problemi implementativi.

Noirblack è un gioco BBMMOG di strategia, in tempo reale, ambientato nello spazio, in cui i giocatori possono effettuare attacchi fra varie flotte di navi spaziali. L'algoritmo che deve compiere le elaborazioni di queste battaglie spaziali è però soggetto ad alcuni problemi computazionali che lo rendono inutilizzabile con numeri elevati di navi.

Scopo della tesi è la descrizione dettagliata dell'algoritmo, evidenziandone i problemi implementativi, e la successiva modellazione probabilistica finalizzata alla riduzione della complessità computazionale.

Per fare ciò, si eseguiranno alcuni passaggi probabilistici tratti dalle regole di funzionamento dell'algoritmo che culmineranno nell'utilizzo di una versione non-standard del teorema del limite centrale al fine di ottenere un'approssimazione accettabile dell'algoritmo iniziale.

Finita la fase dell'elaborazione del modello, si proporrà una soluzione implementativa basata sul linguaggio di programmazione PHP e si concluderà effettuando una simulazione di gioco. I risultati della simulazione saranno analizzati cercando di valutarne criticamente la qualità e la coerenza.

Indice

Acronimi **IX**

1 Introduzione: cosa sono gli MMOG **1**
1.1 Problemi implementativi di un MMOG 2
1.1.1 Sistema distribuito 2
1.1.2 Sistema client-server 3
1.2 Client-server: problemi implementativi 4
1.2.1 Ottimalizzazione della banda 5
1.2.2 Ottimalizzazione dei dati 5
1.2.3 Re-sincronizzazione dei giocatori 6
1.2.4 Sicurezza 6
1.3 BBMMOG: Browser based MMOG 7
1.3.1 Problemi comuni nei BBMMOG 8
1.3.2 Noirblack un BBMMOG integrato 9
1.3.3 Noirblack: Problemi e scelte Implementative 11

2 L'Algoritmo d'attacco di Noirblack **13**
2.1 Descrizione dell'algoritmo 13
2.1.1 I parametri 14
2.1.2 Le notazioni 15
2.1.3 La fase d'attacco 16
2.1.4 Conteggio dei danni 16
2.1.5 Conclusione del match 16
2.2 Problemi Implementativi 17
2.3 Modellizzazione dell'algoritmo 17
2.3.1 Definizioni e notazioni: cosa è cambiato 18
2.3.2 Calcolo dei colpi medi 19
2.3.3 Elaborazione di un modello per l'attenuazione fornita dallo scudo 20

2.3.4 Elaborazione di un modello per la valutazione del danno alle corazze 21
2.3.5 Elaborazione del Modello di probabilità di esplosione . 22
2.3.6 Elaborazione del modello di calcolo per le navi esplose 22
2.4 Implementazione dell'algoritmo 24
2.4.1 Calcolo dei colpi medi 24
2.4.2 L'attacco: calcolo degli scudi 25
2.4.3 L'attacco: calcolo delle corazze 26
2.4.4 Calcolo dei danni 27
2.4.5 Implementazione del round 28

3 Conclusioni **31**
3.1 Simulazioni 31
3.1.1 Flotte comparabili 32
3.1.2 Flotte non comparabili 34
3.2 Conclusioni 35

Acronimi

MMOG Massive Multiplayer Online Game

BBMMOG Browser-Based Massively Multiplayer Online Game

MMOFPS Massively Multiplayer Online First Person Shooter

MMORPG Massively Multiplayer Online Role Player Game

MMORTS Massively Multiplayer Online RealTime Strategy

ARQ Automatic Repeat reQuest

UDP User Datagram Protocol

TCP Transmission Control Protocol

IP Internet Protocol

AJAX Asynchronous Javascript And Xml

W3C World Wide Web Consortium

HTML HyperText Markup Language

CSS Cascade StyleSheet

ORM Object-relational mapping

iid Indipendenti Identicamente Distribuite

Capitolo 1

Introduzione: cosa sono gli MMOG

L'incremento della potenza di calcolo dei PC, lo sviluppo di ambienti grafici sempre più performanti e l'aumento della capacità di immagazzinare e scambiare dati fra PC, ha permesso lo sviluppo di molte categorie di videogiochi sempre più raffinate. Una di queste categorie di videogiochi, diffusasi specialmente grazie alla penetrazione di internet nelle case, prende il nome di MMOG ovvero Massively Multiplayer Online Game.

Un videogioco viene classificato come MMOG quando il suo scopo principale è fornire un ambiente atto all'interazione di molti utenti in un unico e complesso ambiente virtuale. Questa funzione viene raggiunta sfruttando le reti accessibili al calcolatore e spesso questo coincide con l'utilizzo di una rete TCP/IP di cui internet è il massimo esponente[1]. Gli MMOG si dividono ulteriormente in sotto categorie basate sul genere di gioco che implementano di cui possiamo ricordarne tre in particolare: MMORPG, MMOFPS, MMORTS.

Si parla di MMORPG quando il gioco è comparabile ad un classico gioco di ruolo dove solitamente il giocatore interpreta un personaggio del mondo virtuale con abilità e caratteristiche uniche e migliorabili. Fra tutti ricordiamo World of Warcraft, Metin2 e Ragnarok Online.

L'acronimo MMOFPS viene assegnato ai giochi sparatutto in prima persona. In questo genere di giochi, il giocatore rappresenta un personaggio il cui fine ultimo è quello di scontrarsi in duelli spesso all'ultimo sangue con altri giocatori con l'obbiettivo di conquistare punti e primeggiare su tutti i partecipanti. Di questo genere citiamo Unreal Tournament,Quake Arena, Call of Duty Online.

Infine con l'acronimo MMORTS ricordiamo i giochi che si ispirano ai giochi di strategia. In questi videogiochi il giocatore deve organizzare il suo micro-mondo al fine di renderlo il più competitivo possibile con i micromondi degli avversati. In questa categoria rientrano sicuramente Stronghold Kingdoms,Empire Earth Online,Ogame e Travian.

(a) World of Warcraft (MMORPG)

(b) Unreal Tournament 3 (MMOFPS)

(c) Stronghold Kingdoms (MMORTS)

Figura 1.1: Screenshot di alcuni MMOG

1.1 Problemi implementativi di un MMOG

La necessità di comunicare in tempo reale con altri giocatori e di utilizzare reti pubbliche come internet per trasmettere i dati ai vari giocatori, ha reso necessario lo sviluppo di un sistema che permetta l'interazione dei giocatori nelle rete.

Questa comunicazione può essere raggiunta utilizzando un sistema distribuito oppure un sistema client-server. Una delle differenze sostanziali fra queste due architetture è sicuramente la presenza o meno di un nodo all'interno della rete con il compito di coordinare l'accesso e lo scambio di dati fra la stessa.

1.1.1 Sistema distribuito

Nel sistema distribuito tutti i nodi sono uguali, è quindi assente un nodo con il ruolo di coordinare e gestire l'accesso alle risorse della rete. Questo comporta uno sforzo maggiore per i calcolatori che, oltre ad elaborare le informazioni da inviare all'utente, devono anche elaborare un insieme di informazioni atte a mantenere il collegamento con gli altri calcolatori e a garantire l'accesso di ulteriori macchine alla rete. All'atto pratico, utilizzare un sistema distribuito, implica l'utilizzo di tecniche atte alla conservazione

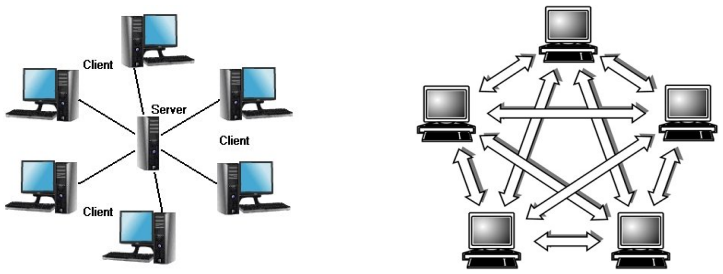
delle informazioni all'interno della rete e questo risulta particolarmente complesso e poco utile al fine di implementare MMOG. Un ulteriore problema del sistema distribuito risiede nella gestione dei dati sensibili degli utenti e relativi allo stato del gioco i quali sono contenuti interamente nei computer dei giocatori stessi rendendo quindi più complessa la fase di verifica dell'integrità dei dati a causa di una potenziale manipolazione degli stessi da parte dell'utente.

1.1.2 Sistema client-server

Nel sistema client-server, possiamo invece notare la presenza di un nodo principale che ha il compito di garantire la connessione fra tutti i client presenti e di vigilare sull'integrità dei dati inviati e ricevuti dai giocatori, riuscendo quindi a bloccare azioni atte alla violazione delle regole strutturali del gioco. Altra caratteristica intrinseca del sistema client-server è il controllo centralizzato di tutti gli utenti con la possibilità di creare delle strutture automatiche atte a vigilare sul comportamento dei giocatori e in grado di garantirne la possibile esclusione degli stessi se non rispettassero i parametri imposti dal proprietario (e.s. pagamento di un abbonamento).

La soluzione client-server comporta anche una riduzione delle attività del client con conseguente riduzione del carico di lavoro che gli stessi devono gestire permettendo una semplificazione del software con conseguenti richieste hardware meno stringenti per le macchine client. Gli svantaggi del sistema client-server sono la necessità di sviluppare un software aggiuntivo e specializzato a garantire l'iterazione degli utenti. Questo software deve inoltre essere eseguito su macchine opportune dotate di sufficienti capacità computazionali e di canali di comunicazioni proporzionali all'utenza e alla mole di dati che esse dovranno gestire. Quest'ultimo svantaggio risulta un limite non indifferente per l'implementazione privata di una struttura simile frenata sia dal costo di mantenimento che dal costo d'avviamento della stessa. Parallelamente diventa un vantaggio per le software house che possono permettersi di sostenere il costo per avviare queste strutture confidando nel futuro rientro economico derivato dal porre un costo fisso per l'accesso alla stessa o dall'imposizione di un prezzo adeguato per l'acquisto del client di gioco.

Concludendo, risulta chiaro che per fini economici la scelta più attrattiva risulti essere la soluzione client-server e in effetti, la quasi totalità degli MMOG è creata utilizzando questo sistema.



(a) Sistema client-server (b) Sistema Distribuito

Figura 1.2: Schema delle tipologie di comunicazione fra calcolatori

1.2 Client-server: problemi implementativi

Dopo aver valutato le differenze di superficie fra queste due strutture, analizzeremo in dettaglio i problemi d'affrontare nell'implementazione di un MMOG su sistema client-server. I problemi che sicuramente in ogni sistema client-server bisogna affrontare sono sostanzialmente legati alla sicurezza e alle performance. L'utilizzo di macchine performanti, non è infatti assolutamente sufficiente per garantire una comunicazione fluida fra i vari client.

Ogni MMOG richiede una quantità di dati da gestire dipendente dal genere di gioco che si vuole implementare e dalle regole che si vogliono adottare. La tipologia più onerosa in questi termini, è sicuramente quella dei giochi sparatutto dove è necessario un invio continuo e in tempo reale della posizione di ogni singolo giocatore e di ogni singolo oggetto capace di interagire con il personaggio (es. proiettili). Questa necessità rende il gioco estremamente pesante nei momenti più adrenalinici dove c'è una maggiore concentrazione di personaggi e quindi di dati da inviare ai relativi client. In queste situazioni infatti possono verificarsi potenziali rallentamenti e anomalie dovuti alla banda insufficiente o a latenza della rete.

Un altro problema che bisogna affrontare nella progettazione di un buon MMOG è sicuramente la quantità di informazioni da salvare al fine di poter identificare univocamente e continuativamente nel tempo il giocatore. La categoria che risulta più sensibile a questo problema è sicuramente la categoria degli MMORPG, in quanto necessitano di memorizzare in modo dettagliato lo status di ogni personaggio incluse le caratteristiche degli oggetti che lo stesso ha con se (se presenti nel gioco). Questa situazione risulta ulteriormente aggravata se, oltre alle caratteristiche degli avatar dei giocatori, si decide di salvare anche i messaggi inviati e le statistiche degli stessi per poterli analizzare in un secondo momento al fine di garantire il rispetto delle regole imposte dal gestore del server. In presenza di un gran numero di giocatori, il salvataggio di questi dati può diventare molto oneroso, è quindi comune salvare i dati sopra citati per periodi temporali limitati.

L'ultimo problema che possiamo attribuire ad una particolare categoria di MMOG è sicuramente il mantenimento e l'aggiornamento delle informazioni del giocatore anche quando esso non è più effettivamente connesso alla struttura. Questo problema è tipico degli MMORTS dove normalmente il server ha il compito di mantenere aggiornato un mondo virtuale più o meno ampio, ove i giocatori interagiscono. Solitamente, questo tipo di giochi procedono anche se il giocatore non è più online obbligando il server a compiere uno sforzo proporzionale al numero di giocatori registrati per mantenere aggiornato e coerente il mondo virtuale.

Concludendo, è doveroso ricordare che l'autenticazione e l'invio sicuro delle informazioni tramite la rete è la cosa più critica e problematica da implementare in qualsiasi scenario e risulta spesso il problema da considerare con maggiore attenzione al fine di mantenere la struttura sicura e stabile.

1.2.1 Ottimalizzazione della banda

L'ottimalizzazione della banda disponibile, è un problema che sicuramente bisogna curare al fine di rendere l'esperienza di gioco più accattivante e gradevole. Un buon progettista di MMOG deve essere conscio che ogni persona dispone di connessioni internet più o meno performanti e deve garantire una soglia minima di giocabilità con un utilizzo di banda il più irrisorio possibile. Per fare questo si possono usare svariate tecniche che comprendono la compressione del flusso dati per migliorare l'entropia media [1] oppure l'utilizzo di strategie adatte a ridurre gli effetti causati dal ritardo della connessione fra server e client (lag) mantenendo comunque alta la qualità di gioco sacrificando magari l'idealità del canale di comunicazione in favore di un ritardo minore (es uso del UDP al posto del TCP).

1.2.2 Ottimalizzazione dei dati

Sebbene i dati salvati siano spesso informazioni molto piccole (una data, un indirizzo ecc) possono comunque essere un problema sopratutto quando il numero di utenti è molto elevato e i periodi di esercizio sono dilatati. In questi casi i dati salvati possono occupare uno spazio non più trascurabile e possono inoltre insorgere problemi legati al recupero degli stessi. Per organizzare in modo efficiente i dati spesso si utilizzano database relazione (es Mysql,Postgres) che riducono il compito del progettista all'organizzazione e correlazione logica dei dati fra di loro.

Per velocizzare il recupero dei dati, infatti, i database relazionali mettono a disposizione del progettista un insieme di strumenti molto efficaci quali indici e relazioni esterne al fine di velocizzare le ricerche all'interno dell'archivio di dati e mantenerne la coerenza e la correlazione fra di essi. All'atto della progettazione del gioco, il progettista, deve quindi dedicare un tempo

adeguato ad analizzare i dati che vuole salvare e alle relative correlazioni che essi offrono cercando il giusto equilibrio tra velocità e memoria occupata.

1.2.3 Re-sincronizzazione dei giocatori

La disconnessione di un client è un evento che può causare dei problemi, a tale fine il server deve imporre una strategia atta a permettere il rientro nella rete dello stesso in un secondo momento.

Si possono adottare fondamentalmente due strategie per compiere questa azione: cancellare l'utente dalla sessione di gioco attuale oppure lasciarne un avatar rappresentativo in situazione d'attesa fino al ritorno dell'utente sconnesso.

La prima risulta inattuabile o fortemente demotivante con i giochi di strategia in tempo reale mentre risulta apprezzabile e consigliabile in giochi sparatutto o comunque organizzati in round di modesta durata.

La seconda risulta, invece, preferibile in ambienti strategici e con durate lunghe o addirittura senza durata. In questa tesi ci limiteremo ad analizzare la seconda di queste due alternative.

Compiere questa scelta, obbliga a prevedere un sistema che aggiorni in modo autonomo e continuativo le caratteristiche dei giocatori allo scopo di mantenere l'ambiente di gioco aggiornato e coerente. Questo modo operativo, seppur logicamente corretto, risulta computazionalmente oneroso in quanto costringe il server ad aggiornare ogni utente ad intervalli prefissati che devono essere ravvicinati per garantire la fluidità del gioco. Questo lavoro di background può portare un sovraccarico di lavoro al server con potenziali rallentamenti e malfunzionamenti generali. Un altro approccio al problema, suggerisce di aggiornare le informazioni dell'utente e del mondo, solo quando effettivamente esse sono richieste. In questo caso bisogna attuare delle regole atte a identificare quando un aggiornamento è necessario e un insieme di procedure atte a ridurre al minimo i dati da processare in suddetto aggiornamento. Utilizzando questo secondo approccio il server risulta meno stressato e quindi è possibile garantire una maggior giocabilità all'utente finale.

1.2.4 Sicurezza

Il problema più delicato da analizzare nel progetto di un MMOG è sicuramente, come già detto, quello della sicurezza.

Internet risulta spesso un luogo privo di controllo ove una possibile minaccia informatica si cela ad ogni angolo buio e a causa di ciò bisogna prestare particolarmente attenzione ad attuare strategie atte a rendere le comunicazioni sicure e non mutabili. A tale fine, lo sviluppatore può utilizzare alcune accortezze a seconda del protocollo che intende utilizzare.

Limitandoci alle reti TCP/IP su cui si basa internet, i protocolli sicuramente più usati sono il TCP e l'UDP.

Analizzando il protocollo TCP possiamo facilmente convincerci che la trasmissione dei dati avviene senza errori ma con dei ritardi proporzionali alla congestione della linea. Possiamo considerare quindi il canale di comunicazione ideale ricordandoci che lo stesso avrà delle latenze proporzionali alla congestione della linea[1]. I problemi del protocollo TCP sono quindi ridotti alla sola interferenza da parte di un possibile utente esterno che si inserisca nella comunicazione intercettando i pacchetti con lo scopo recuperare dati sensibili oppure per alterarli eludendo i controlli previsti dal protocollo TCP. Per proteggersi da questi problemi, si possono adottare sistemi di codificazione con opportuni codici di ridondanza al fine di identificare le manomissioni. Questo approccio richiede però un aumento della banda necessaria per la trasmissione.

In alternativa si può utilizzare un sistema di cifratura fra client-server per impedire la comprensione dei dati utilizzando algoritmi di codifica asimmetrici come RSA che riescono a garantire una buona sicurezza dei dati inviati. Il secondo protocollo a disposizione dello sviluppatore è il protocollo UDP. Questo protocollo è obbligatorio quando vogliamo ridurre le latenze e i ritardi fra client e server ma a differenza del canale TCP il canale UDP risulta non ideale e quindi oltre ai problemi sovraesposti nel canale TCP, dobbiamo preoccuparci anche di rilevare eventuali errori di comunicazione. I possibili errori di comunicazione nel canale UDP sono:

- perdita di pacchetti
- perdità dell'ordine di trasmissione dei pacchetti

Quando si sviluppa un gioco utilizzando il protocollo UDP è necessario ricordarsi di queste problematiche e, nel caso nessuno le abbia affrontate al posto nostro, bisogna ricordarsi della loro presenza ed attuare strategie atte a controllarle e correggerle se si sono verificate.

Il vantaggio che può sopraggiungere nel utilizzo del protocollo UDP rispetto all TCP è sicuramente riscontrabile nella velocità della comunicazione che, a patto di tollerare alcuni errori, risulta molto più performante e ottimizzata non dovendo aspettare le continue risposte ai controlli di errore presenti nel sistema di correzione d'errore basato su ARQ tipiche del protocollo TCP[1].

1.3 BBMMOG: Browser based MMOG

Una sezione molto particolare degli MMOG risiede sicuramente negli MMOG browser based. Essi si presentano come dei semplici siti web in cui, una volta iscritti, si accede ad un mondo virtuale dove il giocatore interagisce in modo

più o meno raffinato con gli altri utenti. Per poter usufruire di questi videogiochi, è necessario utilizzare un browser web e se necessario installare o abilitare alcune features aggiuntive dei browser stessi atte a fornire il supporto per alcune tecnologie particolari indispensabili al corretto funzionamento del MMOG (e.s. Flash,Javascript, AJAX...).

Questa categoria di giochi, è accessibile completamente ed esclusivamente online, essa non richiede l'installazione di nessun client da parte dell'utente e l'interazione con esso è quindi limitata al solo utilizzo del sito web. In questa categoria troviamo giochi fra loro molto differenti ma quelli che quelli di maggior successo appartengo alla categoria degli strategici in tempo reale oppure in quella dei giochi di ruolo nei quali l'utente deve migliorare le caratteristiche del suo personaggio per diventare più forte degli altri giocatori oppure deve governare il suo micro-sistema al fine di renderlo più potente e prestigioso.

Fra tutti i browser game, si vuole ricordarne alcuni di successo che sono semplici da reperire sui più comuni motori di ricerca: Forsakia, Ogame, Crystal Saga, Travian, BiteFight, Urban Rivals, Dark Orbit...

(a) DarkOrbit (b) Travian (c) Ogame

Figura 1.3: gameplay di alcuni BBMMOG

1.3.1 Problemi comuni nei BBMMOG

A differenza dei comuni MMOG, nel caso degli BBMMOG la tecnologia Server-Client è una scelta obbligatoria.

I problemi aggiuntivi che si presentano in questo gruppo, oltre a quelli citati in precedenza, sono sicuramente legati alla compatibilità. Infatti, a differenza degli MMOG dove si sviluppa un client ad hoc, essi necessitano solo di un browser per poter funzionare ma, agli albori di internet e nel suo immediato futuro, i browser erano fra i più vari e ognuno di essi implementava una propria versione personalizzata degli standard html redatti dal W3C senza però rispettarli integralmente. Questa situazione è stata aggravata dall'atteggiamento iniziale della Microsoft, che ha cercato di imporre, installandolo in modo predefinito, Internet Explorer in tutti i sui sistemi operativi non curandosi di sviluppare questo software in conformità degli standard redatti dal W3C e creando grandi difficoltà a tutti gli sviluppatori web che cercavano di mantenere una buona compatibilità anche con gli altri

browser. A causa di questo, rendere un sito e quindi un BBMOG, compatibile con Internet Explorer risulta spesso molto complesso, fortunatamente negli ultimi anni Microsoft ha cambiato linea d'azione e le nuove versioni di Internet Explorer risultano più semplici da gestire, ottimizzare e rendere compatibili con gli altri browser.

Se da un lato la Microsoft si sta avvicinando agli standard, dall'altro un nuovo mondo non trascurabile e in aumento vertiginoso si sta affacciando prepotentemente al mondo del web: gli smartphone e i tablet.

Questi nuovi dispositivi, si stanno diffondendo sempre più in profondità nelle case delle persone imponendo agli sviluppatori web di adattarsi anche alle loro necessità: pagine leggere, risoluzioni ridotte, accessibilità con sistemi di input fra i più vari (tastiere telefoniche, tastiere qwerty,touch input) e utilizzo minimale o nullo di Javascript e in generale di linguaggi client-side. A causa di queste differenze molto marcate con i normali PC, i tablet e gli smartphone necessitano di un ambiente grafico ad hoc.

A questo fine i linguaggi di programmazione e i server web si stanno adattando per fornire supporti ai progettisti atti a risolvere il problema che però hanno comunque un carico di lavoro ulteriore da dover gestire.

Ultimo problema particolare che necessario affrontare nella creazione di un BBMMOG è dato dalla necessità di scegliere un buon compromesso fra grafica e velocità: rispetto ad un comune MMOG l'aspetto grafico del gioco grava pesantemente sulla velocità del gioco e quindi sui ritardi del gioco. Se da un lato i giocatori amano grafiche belle ed esteticamente attraenti, dall'altro lato amano potersi muovere con velocità e fluidità nel gioco oltre che avere tempi di attesa limitati. Queste due caratteristiche sono discordi fra loro ed è quindi compito del progettista trovare il giusto equilibrio fra esse. Una possibile soluzione a questo problema è lasciar scegliere all'utente che caratteristiche grafiche usare a seconda delle sue preferenze e della sua linea di comunicazione.

1.3.2 Noirblack un BBMMOG integrato

Dopo questa attenta analisi sui giochi MMOG e in particolare sui BBMMOG ci concentreremo su uno di questi conosciuto come NoirBlack, in particolare, focalizzeremo la nostra attenzione su un problema che riguarda questa struttura, lo risolveremo e svilupperemo una soluzione algoritmica del problema.

Noirblack è un BBMMOG di strategia in tempo reale ambientato nello spazio. Originariamente è nato col fine di rispecchiare un altro BBMMOG chiamato Ogame da cui si differenziava solo per le velocità di gioco estremamente più veloci. Col passare del tempo, sono state aggiunte nuove funzionalità, strutture e caratteristiche atte a personalizzarlo al fine di dare ai giocatori un esperienza di gioco più accattivante e coinvolgente.

All'iscrizione è possibile scegliere un universo di gioco, ogni universo rappresenta un micro-mondo dove il giocatore interagisce, Questo micro-mondo è isolato da tutti gli altri e permette l'iterazione solo fra gli utenti giocanti in esso.

Ogni giocatore inizia con un singolo pianeta che deve migliorare costruendo strutture finalizzate ad aumentarne la sua micro-economia.

Successivamente si acquisisce la capacità di compiere ricerche di nuove tecnologie le quali permettono a loro volta di sbloccare nuove funzionalità e diventare quindi più competitivi nel gioco.

Ogni ricerca sbloccata permette a sua volta di acquisire nuove abilità o di sbloccare nuove strutture e navi che permettono di accedere ad altre funzionalità del gioco come la colonizzazione di nuovi pianeti oppure l'attacco di pianeti avversari.

Oltre al puro BBMOG, Noirblack offre un insieme di pacchetti software atti ad estendere il supporto al gioco e renderlo più immediato e accogliente possibile. L'obbiettivo finale che si vuole raggiungere è creare una community virtuale che ruota attorno al gioco permettendo di raggiungere un autonomia economica.

Attualmente ci sono all'incirca 200 persone fisse, ma c'è un continuo cambio di utenza causato dai bug che affliggono l'attuale versione di gioco.

La risoluzione di questi bug risulta molto onerosa a causa di una programmazione in stile "spaghetti" generata dai continui interventi da parte di programmatori per lo più volontari poco esperti e con idee spesso discordi. Per questo motivo, a dicembre 2010 è stato scelto di intraprendere una strada diversa: si è scelto di abbandonare lo sviluppo del vecchio codice spaghetti e avviare un nuovo progetto più coerente e solido che permettesse una maggiore integrazione del gioco nella struttura dove tutti i pacchetti comunicassero e interagissero fra loro in modo autonomo e armonioso.

Analizzando in dettaglio la piattaforma Noirblack, si può notare che tutto gravità attorno ad un server Linux, il quale gestisce l'intero sistema che vede il suo cuore nel software Apache2 configurato per gestire attualmente 5 sottodomini: www,board,wiki,skin,mail atti a fornire le funzionalità ausiliarie del game ovvero la pagina principale, il forum, il mail server, un portale wikipedia per inserire le informazioni e le guide relative al gioco e alla piattaforma e infine un sotto dominio atto ad ospitare le skins di gioco.

Oltre ai domini sopracitati, il gioco ha un ulteriore numero di sotto domini che contengono gli universi di gioco. Ogni universo è basato su un database relazione Mysql e su un insieme di script lato server finalizzati a mantenere sincronizzate le iterazioni fra i vari giocatori (attacchi, trasporti di risorse...)

Si è infine deciso di integrare il sistema di login del forum Phpbb3 con quello del game imponendo un unica registrazione per tutta la struttura e utilizzando le informazioni presenti nel forum per autenticare gli utenti anche nelle altre strutture.

(a) vecchio universo

(b) universo in sviluppo

(c) homepage del sistema

Figura 1.4: Alcuni screenshot del sistema Noirblack

1.3.3 Noirblack: Problemi e scelte Implementative

Il primo universo e attualmente unico universo di gioco, si basa su un progetto chiamato Ugamela, che in seguito ad un refactoring, è stato ribattezzato Xnova. Xnova è stato successivamente modificato dal team Noirblack al fine di correggere i bug presenti e adattarlo alle caratteristiche funzionali ricercate in questo gioco.

Nel corso della rielaborazione del codice, si è notato che lo stesso risulta mal progettato ed inefficiente. La progettazione del database è praticamente assente, risulta poco chiara sono mancanti molte correlazioni logiche e il database risulta spesso privo di molti indici fondamentali. Il codice non è testabile: sono presenti parecchie variabili globali e le funzioni non sono documentate e di difficile mantenimento. Per queste ragioni, nel dicembre 2010 è stato scelto di abbandonare completamente il codice e di riscriverlo utilizzando l'approccio di programmazione indicato dall'Extreme Programming.

A tale fine si è deciso di abbandonare il sistema di programmazione a funzioni in favore del sistema Object Based. Si è deciso inoltre di sfruttare il più possibile le funzionalità messe a disposizione dal Database e quindi si è optato per l'utilizzo del sistema relazionale fornito da InnoDb che permette di utilizzare le transazioni e le relazioni di chiave esterna.

Infine si è scelto di adottare il Framework Symfony al fine di velocizzare e semplificare la scrittura e il testing del codice.

Si è scelto di utilizzare il framework symfony anche perchè risulta abbastanza intuitivo e veloce, inoltre permette di isolare in modo molto semplice la parte operativa del codice (Azioni) dalla parte grafica e rappresentativa delle informazioni (Template). Symfony cura anche l'aspetto di comunicazione e interfacciamento con il database mettendo a disposizione dello sviluppatore un ORM per poter utilizzare le potenzialità del Database Relazione mantenendo l'ottica della programmazione ad oggetti. Tuttavia, utilizzare un ORM al posto di compiere iterazioni direttamente con il Database si è dimostrato inefficiente in alcune situazioni e in quel caso il framework mette a disposizione altri strumenti utili a ottimizzare comunque la situazione a discapito dell'immediata comprensione del codice.

Il tempo di sviluppo del codice e la sua mantenibilità risultano notevolmente migliorati grazie alla divisione in moduli del sistema e grazie alla struttura a classi prevista per ogni modulo dove i metodi rappresentano le azioni che i moduli possono compiere. Il framework Symfony mette infine a disposizione dello sviluppatore un insieme di tool atti a fare debug e a controllare i comportamenti anomali della struttura. Symfony da inoltre supporto nativo per la libreria PHPUnit indispensabile per fare il testing automatico del codice. Per queste ragioni si è quindi dimostrato il framework più completo e funzionale nell'ottica di sviluppo definita dall'extreme progamming che sta avendo parecchio successo in questi anni grazie allo sviluppo molto veloce del codice che permette.

Figura 1.5: schema logico di funzionamento dell'Extreme Programming

Capitolo 2

L'Algoritmo d'attacco di Noirblack

Dopo una breve panoramica sul sistema Noirblack e sulle scelte implementative utilizzate, si vuole ora analizzare in dettaglio un problema che ha creato particolari perplessità e dubbi agli sviluppatori della piattaforma.

Il problema che si vuole analizzare riguarda l'algoritmo finalizzato ad elaborare gli scontri spaziali fra i giocatori. Questo algoritmo si basa sulle regole di scontro definite nel BBMOG da cui la piattaforma originariamente si ispirava (Ogame) tuttavia, queste regole, come verrà in seguito chiarito risultano ardue da implementare alla lettera a causa degli elevati numeri di navi presenti su Noirblack e per questo motivo si vuole procedere con l'analizzare e risolvere il problema da un'ottica probabilistica. Come si può facilmente intuire, questo algoritmo implementa una delle parti più delicate del sistema Noirblack. Ha il compito di calcolare l'esito dello scontro fra due o più giocatori che scagliano le loro navi in un duello all'ultimo sparo e deve necessariamente restituire un risultato coerente con le regole imposte per permettere al giocatore di analizzare preventivamente la situazione ed attuare una strategia atta a trarre il maggior profitto dalla situazione.

2.1 Descrizione dell'algoritmo

Un attacco completo è divisibile in sotto parti chiamate round. In ognuno di essi l'attaccante spara contro il difensore mentre questo si protegge, successivamente il difensore spara contro l'attaccante che si difende a sua volta e concludendo vengono calcolate le navi esplose da ambo i lati.

L'attacco è formato da un numero R di round che solitamente è pari a 6 L'algoritmo inizia con una quantità di navi scelte dai giocatori le quali hanno dei valore predefiniti di scudi, di attacco e di corazze che verranno esaminati successivamente. Gli scudi e le corazze vengono danneggiate in ogni round, i primi vengono rigenerati all'inizio di ogni round mentre i secondi vengono rigenerate solo alla fine dell'intero attacco.

Limiteremo l'analisi dell'algoritmo ad un solo round, va comunque ricordando, che questa procedura si deve ripetere R volte variando le condizioni iniziali in accordo ai risultati ottenuti.

Un round è diviso in 4 fasi:

1. l'attaccante spara al difensore
2. il difensore spara all'attaccante
3. si verificano i danni e le eventuali navi esplose nella flotta in attacco
4. si verificano i danni e le eventuali navi esplose nella flotta in difesa

Valutando l'ordine in cui vengono effettuate le fasi si può verificare che le fasi 2 e 4 sono analoghe alle fasi 1 e 3 con i ruoli del difensore e dell'attaccante invertiti. Per questo considereremo solo le fasi 1 e 3 poiché le restanti fasi dell'algoritmo una mera ripetizione adattata di queste.

2.1.1 I parametri

Prima di procedere con la descrizione formale e dettagliata dell'algoritmo, è necessario definire in modo puntuale le variabili che utilizzeremo, il loro ruolo e il loro comportamento all'interno del sistema. L'algoritmo è formato da 2 strutture: la flotta dell'attaccante e quella del difensore. Ogni flotta è formata rispettivamente da un numero di navi predeterminato scelto dal giocatore e ogni nave di queste flotte è caratterizzata da 5 parametri:

- **Tipo**: parametro che indica la categoria a cui appartiene la nave (e.s. Caccia leggero , Incrociatore) da essa dipendono i valori base di attacco,scudo,corazze e rapid fire sotto citati.
- **Attacco**: parametro che rappresenta la capacità offensiva della nave, costante in ogni fase dell'algoritmo e definito all'inizio dello scontro.
- **Scudo**: parametro che rappresenta il valore massimo dello scudo. Si rigenera ad ogni round ed è formato da 100 particelle di valore $\lfloor \frac{1}{100} \rfloor$ del valore totale dello scudo. Ogni particella non è danneggiabile: o viene distrutta completamente oppure assorbe interamente l'attacco nemico senza essere danneggiata.
- **Corazze**: parametro che rappresenta il danno massimo che la nave riesce ad assorbire prima di esplodere con certezza
- **RapidFire** indica la capacità che ha una nave di sparare nuovamente dopo dell'aver colpito una nave di una determinata categoria. Il parametro Rapid fire è immutevole e possiamo riassumerlo in una matrice $M \times M$ dove M rappresenta il numero di tipi di navi presenti nel gioco.

2.1.2 Le notazioni

Definiamo ora le notazione che si vogliono usare per descrivere l'algoritmo,supposti h, k due indici, definiamo con:

- n numero di navi dell'attaccante $n \in \mathbb{N}^+$
- m numero di navi del difensore $m \in \mathbb{N}^+$
- α_k valore d'attacco della k-esima nave attaccante, $k = 1, 2, 3, ...n$
- γ_k tipo di appartenenza della nave k del attaccante
- β_h tipo di appartenenza della nave h del difensore
- ρ_{β_k,γ_h} valore di fuoco rapido della k-esima nave attaccante rispetto la h-esima nave del difensore, dove $k = 1, 2, 3, ...n$ e $h = 1, 2, 3, ...m$
- δ_h valore di resistenza di ogni singola particella di scudo della h-esima nave del difensore dove $h = 1, 2, 3, ...m$
- θ_h numero di particelle attive dello scudo della h-esima nave del difensore, inizialmente pari a 100 dove $h = 1, 2, 3, ...m$
- ϕ_h valore di integrità iniziale della h esima nave del difensore dove $h = 1, 2, 3, ...m$
- ω_h valore di integrità residua della h esima nave del difensore, inizialmente $\omega_h = \phi_h$ dove $h = 1, 2, 3, ...m$

Definite i parametri di nostro interesse, possiamo organizzarli in due matrici.

Il difensore è stato organizzato in una matrice $4 \times m$ cosi definita:

$$\begin{bmatrix} \delta_1 & \theta_1 & \phi_1 & \omega_1 \\ \vdots & \vdots & \vdots & \vdots \\ \delta_m & \theta_m & \phi_m & \omega_m \end{bmatrix}$$

L'attaccante invece è stato organizzato in una matrice $(h+1) \times n$:

$$\begin{bmatrix} \alpha_1 & \rho_{1,1} & \cdots & \rho_{1,\beta_h} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_n & \rho_{\gamma_n,1} & \cdots & \rho_{\gamma_n,\beta_h} \end{bmatrix}$$

2.1.3 La fase d'attacco

In questa fase avviene lo scontro fra la flotta dell'attaccante e la flotta del difensore:

1. $\forall k \mid 1 \leq k \leq n+1$
2. scelgo a caso un h t.c. $1 \leq h \leq m$
3. $\begin{cases} \omega_h = 0 & se\ \alpha_k > \omega_h + \delta_j \theta_h \\ \omega_h = \alpha_k - \delta_h * \theta_h & se\ \delta_h \theta_h < \alpha_k < \omega_h + \delta_j \theta_h \\ \theta_h = \theta_h - \lfloor \frac{\alpha_k}{\delta_h} \rfloor & se\ \alpha_k < \theta_h \delta_h \end{cases}$
4. scelgo r a caso tale che $0 \leq r \leq 1$
5. $\begin{cases} \text{torno al punto 2} & se\ r \geq \frac{\rho_{\gamma_k,\beta_h}}{\rho_{\gamma_k,\beta_h}+1} \\ \text{procedo con il prossimo valore di } k & se\ r < \frac{\rho_{\gamma_k,\beta_h}}{\rho_{\gamma_k,\beta_h}+1} \end{cases}$

2.1.4 Conteggio dei danni

Concluso l'attacco sia da parte dell'attaccante che del difensore, non ci resta che valutare i danni e calcolare nel modo sotto riportato le navi esplose, qui calcoleremo, come sopra, solo le navi esplose del difensore ma in modo analogo si procede con l'attaccante.

1. $\forall h \mid 0 < h \leq m$
2. Scelgo p casualmente t.c. $0 \leq p \leq 1$
3. $\begin{cases} \text{Nave distrutta} & se\ (0,5 \geq \frac{\omega_h}{\phi_h}) \wedge (p \leq 1 - \frac{\omega_h}{\phi_h}) \\ \text{Nave ancora in gioco} & altrimenti \end{cases}$

2.1.5 Conclusione del match

Alla fine di ogni round si verifica se ci sono le condizioni per terminare il combattimento, se tali condizioni si verificano il combattimento termina altrimenti si procede con il round successivo.

La condizioni necessarie per decretare la fine dell'attacco sono l'annientamento di una delle due flotte: se viene annientato il difensore, vince l'attaccante mentre si viene annientato quest'ultimo vince il difensore.

In caso di annientamento simultaneo delle due flotte oppure al termine di tutti i round regolamentari con la presenza di navi operative in ambo gli schieramenti, si decreta la fine in condizioni di pareggio.

2.2 Problemi Implementativi

Esaminiamo ora l'algoritmo appena descritto in termini di velocità computazionale e di complessità temporale. Si può subito notare che la complessità temporale di un singolo round è pari a $\mathcal{O}((m \cdot n)^2)$ infatti per compiere una simulazione di un round bisogna scorrere per ogni nave tutte le antagoniste presenti nello schieramento opposto. Si può pensare di ottimizzare l'algoritmo con alcuni artifici cercando di compiere la valutazione dei danni nella fase di attacco portando l'algoritmo ad una complessità temporale $\mathcal{O}(m \cdot n)$. Risulta però impossibile scendere al disotto di tale limite essendo necessario scorrere entrambe le matrici almeno una volta. In questo calcolo abbiamo volutamente trascurato la complessità temporale aggiuntiva dovuta al rapid fire. Analizzandola possiamo notare che essa rallenta l'algoritmo in modo drammatico, portandolo alla complessità $\mathcal{O}(\infty)$ infatti nella peggiore delle ipotesi quella in cui una nave continui a sparare indefinitamente, l'algoritmo rimanere bloccato.

Analizzando l'occupazione di memoria, si può evidenziare una complessità $\Theta(k+h)$ dovuta alla necessità di allocare in memoria una struttura dati atta a mantenere lo status della nave durante l'attacco.

Da questa veloce analisi si può dedurre che l'algoritmo si comporta in un modo assolutamente ingovernabile con un numero di navi elevato. Considerando il numero medio di navi di Noirblack (10^9 unità) e limitandoci ad analizzare solamente l'occupazione in memoria dell'algoritmo possiamo stimare che esso necessiterebbe di una memoria pari a:

$$2(flotte) \cdot 4(variabili) \cdot 2(\tfrac{bytes}{variabile}) \cdot 10 \cdot 10^9(\#navi) = 8 \cdot 10^{11} bytes \simeq\ 0.72Tb$$

Una tale dimensione di memoria volatile (RAM) è assolutamente impensabile anche perché abbiamo scelto un numero di navi nella norma del gioco e non un caso limite. Si potrebbe pensare di utilizzare lo spazio fisico (Hard Disk) per compiere la computazione ma risulterebbe di alcuni ordini di grandezza più lento, inoltre il problema diventerebbe comunque ingestibile aumentando il numero di navi da 10 Miliardi ai 1000 Miliardi (circa $72TB$ di memoria).

Dopo tale analisi si è cercato una via alternativa per implementare l'algoritmo. Si è scelto di compiere un analisi probabilistica appropriata al fine di garantire un esito confrontabile ma con una complessità temporale e di occupazione della memoria indipendenti dal numero di navi presenti ma legato al tipo delle stesse.

2.3 Modellizzazione dell'algoritmo

L'idea cardine di questo nuovo approccio si basa sulla seguente osservazione: l'algoritmo precedentemente descritto si concentra sulla singola nave e il suo

modo di interagire con il resto del sistema. Inoltre, si nota, che molte navi hanno le medesime caratteristiche e si comportano nello stesso modo: in particolare si osserva che questa situazione si verifica quando le navi sono dello stesso tipo.

Si è quindi deciso di modellare il sistema analizzando il comportamento dei tipi di nave invece che delle singole navi.

Osservando la quantità di tipi di navi presenti, si nota che esse sono in numero inferiore rispetto al numero di navi e in questa situazione l'algoritmo risulta più gestibile rispetto alla precedente.

2.3.1 Definizioni e notazioni: cosa è cambiato

Seguendo questa intuizione, è necessario compiere delle modifiche alle notazioni precedentemente introdotte per riadattarle al nuovo modello.

I nuovi parametri sono:

- h indice del tipo di nave del difensore
- k indice del tipo di nave dell'attaccante
- i indice di nave del difensore
- j indice di nave dell'attaccante
- n navi del attaccante
- m navi del difensore
- x tipi di nave del attaccante
- y tipi di nave del difensore
- β_i tipo di appartenenza della nave i del difensore
- γ_j tipo di appartenenza della nave j dell'attaccante
- ξ_h numero di navi del tipo h del difensore dove $\xi_h \in \mathbb{N}^+$
- ε_k numero di navi del tipo k dell'attaccante dove $k = 1, 2, ...x$
- α_k valore d'attacco del k-esima tipo di navi dell'attaccante, dove $k = 1, 2...x$
- $\rho_{k,h}$ valore di fuoco rapido del k-esima tipo di navi contro h-esimo tipo di navi del difensore, con $k =, 1, 2, ...x$ e $h = 1, 2, ...y$
- δ_h valore di resistenza di ogni singola particella di scudo del h-esimo tipo di navi del difensore $h = 1, 2, ...y$

- θ_h numero medio di particelle attive dello scudo del h-esimo tipo di nave del difensore inizialmente pari a 100 $h = 1, 2, ...y$
- ϕ_h valore di integrità iniziale del h-esimo tipo di navi del difensore $h = 1, 2, ...y$
- ω_h matrice $(xR)\times 2$ contenente i danni puntuali alle navi della categoria h-esima
- $\omega_h(s)$ valore della matrice ω_h di indice $i, 2$
- $\omega_h(s,t)$ valore della matrice ω_h di indice s, t
- ζ_h valore di integrità media residua del h-esimo tipo di navi del difensore inizialmente $\zeta_h = \phi_h$ $h = 1, 2, ...y$

2.3.2 Calcolo dei colpi medi

Ora si vuole stimare il numero di colpi che una nave in attacco può effettuare in un round.

L'algoritmo prevede che ogni singola nave spari contro una nave scelta casualmente nella flotta del difensore e successivamente possa ripetere l'operazione con probabilità definita dal parametro rapid fire.

In seguito stimeremo il numero medio di spari che una nave effettua in un round. Analizzando le regole algoritmiche, si verifica che ogni nave sceglie in modo casuale il suo bersaglio e quindi si può affermare che la probabilità che la nave i-esima sia colpita dalla nave attaccante j-esima è $P(i)$ cioè:

$$P(i) = \frac{1}{m}$$

Da questo risultato possiamo trovare la probabilità condizionata $P(r_j|i)$ che la nave j spari nuovamente dopo aver colpito la nave di indice i:

$$P(r_j|i) = \frac{\rho_{\gamma_j,\beta_i}}{\rho_{\gamma_j,\beta_i}+1}$$

E quindi, grazie alla formula della probabilità totale, possiamo ricavare la probabilità che una nave ha di effettuare un altro sparo, $Pr(j)$ che risulta:

$$P(r_j) = \sum_{i=1}^{m} P(r_j|i)P(i)$$

Le formule trovate, ci mostrano che la probabilità $P(r_j)$ dipende solo dai tipi di nave γ_j e non dalla particolare nave j; a seguito di queste considerazioni si riscrive il risultato appena trovato nel seguente modo:

$$P(h) = \frac{\xi_h}{m}\ P(r_k|h) = \frac{\rho_{k,h}}{\rho_{k,h}+1}\ P(r_j) = P(r_k) = \sum_{h=1}^{y} P(r_k|h)P(h)$$

Questo risultato, ci permette di stimare il numero medio di spari effettuati da una singola nave e analogamente il numero medio degli spari effettuati da un tipo di navi. Analizzando la struttura, si può infatti verificare che il risultato che vogliamo trovare, numero di medio di spari effettuati prima che la nave smetta di sparare, risulta congruo al comportamento di una variabile aleatoria geometrica[3] di parametro $1 - P(r_j)$ e quindi, ricordando che il valore medio di una variabile aleatoria geometrica è $1/p$ il numero medio di spari χ_j è:

$$\chi_j = \frac{1}{1-Pr(j)}$$

mentre il numero medio di spari effettuato dal tipo attaccante k risulta:

$$\chi_k = \frac{\varepsilon_k}{1-Pr(k)}$$

Ottenuto il numero medio di spari, siamo ora interessati a stimare il numero di spari che si concentra contro una tipologia specifica di navi. Per questo passaggio, abbiamo a disposizione il numero medio di spari e la probabilità che uno sparo colpisca una determinata tipologia di navi, grazie a questi due valori e utilizzando una variabile binomiale[3], possiamo stimare il numero medio di spari provenienti dagli attaccanti di tipo k che colpiscono le navi del difensore di tipo h:

$$S(k,h) = \chi_k \cdot P(h)$$

2.3.3 Elaborazione di un modello per l'attenuazione fornita dallo scudo

Trovati i colpi medi, l'algoritmo deve considerare un attenuazione della potenza in attacco coerente con lo stato degli scudi delle navi in difesa.

Questa elaborazione, analogamente a come si procederà per le corazze in seguito, viene compiuta in due modi differenti a seconda che il numero di spari sia maggiore o minore al numero di navi del difensore.

Inizialmente si vuole calcolare il numero di particelle D_{k_h} che l'attacco riesce a superare con certezza:

$$D_{k_h} = \lfloor \frac{\alpha_k}{\delta_h} \rfloor$$

Se $D_k = 0$ allora l'attacco viene deflesso altrimenti si procede in due modi distinti a seconda del valore $S(k,h)$ dove con α'_k e con $S(k,h)'$ si indicano il valore dell'attacco e il numero di colpi residui a seguito dell'attenuazione dovuta agli scudi.

Se $S(k,h) > \xi_h$ si procede con un analisi media:

$$\begin{cases} \text{pongo } \alpha'_h = \alpha_h & \\ S(k,h)' = S(k,h) - \frac{\xi_h \omega h}{D_k} & \text{se } S(k,h)D_k > \xi_h \omega_h \\ S(k,h)' = \alpha'_h = 0 & \text{altrimenti} \end{cases}$$

Se invece $\xi_h \geq S(k,h)$ si procede con un'analisi puntuale:

$$\begin{cases} \text{pongo } S(k,h)' = S(k,h) & \\ \alpha'_k = \alpha_k - \theta_h \delta_h & \text{se } D_k > \theta_h \\ S(k,h)' = \alpha'_h = 0 & \text{altrimenti} \end{cases}$$

Infine, per quanto concerne l'aggiornamento degli scudi residui per il round, si è scelto di operare un aggiornamento medio:

$$\begin{cases} \theta_h = \lfloor \frac{\theta_h \xi_h}{D_k S(k,h)} \rfloor & \text{se } \frac{\theta_h \xi_h}{D_k S(k,h)} > 1 \\ \theta_h = 0 & \text{altrimenti} \end{cases}$$

2.3.4 Elaborazione di un modello per la valutazione del danno alle corazze

Elaborata una strategia per considerare l'attenuazione dell'attacco prodotta dagli scudi, modelliamo ora il danno che subiscono le corazze delle navi per poter successivamente elaborare con precisione il numero di navi che esplodono. Si è scelto di dividere ancora una volta il problema in due parti al fine di stimare dettagliatamente i danni inflitti alle navi.

Se $S(k,h)' > \xi_h$ si effettua un analisi media dei colpi:

$$\zeta_h = a'_k \lfloor \frac{S(k,h)'}{\xi_h} \rfloor$$

Si può notare che in questa interazione sono stati trascurati $S(k,h)' \; MOD \; \xi_h$ colpi che vengono persi a causa dell'arrotondamento. Questi colpi persi verranno recuperati e analizzati utilizzando il procedimento sotto riportato che verrà usato anche per analizzare il caso $S(k,h)' \leq \xi_h$.

Per quanto concerne il caso puntuale, si è osservato che tutte le navi attaccate dal attaccante k vengono danneggiate allo stesso modo. Grazie a questa osservazione, si può pensare di tenere traccia dei danni puntuali che subiscono le navi. Si vuole quindi nella prima colonna indicare l'offset della nave danneggiata e nella seconda i danni che queste hanno subito. Si può quindi presupporre che l'attacco danneggi ξ' navi differenti, per semplicità pensiamo che esse siano consecutive e quando saranno tutte danneggiate riprenderemo dalla prima nave. Gli offset creati in questo modo, sono direttamente proporzionali ai tipi di navi attaccanti presenti e al numero di round. Si può infatti osservare che viene generato un offset per ogni tipologia di navi attaccante e questo si ripete per ogni round. Tutti questi spari vengono salvati nella matrice ω_h e in seguito questi valori verranno utilizzati, assieme al danno medio ζ_h per calcolare le navi esplose.

2.3.5 Elaborazione del Modello di probabilità di esplosione

Finita la fase di attacco, per completare l'algoritmo, dobbiamo calcolare le navi esplose.

Analizzando il comportamento di una singola nave, come abbiamo già detto, sappiamo che una nave può esplodere solo quando il danno è superiore al 50% della sua corazza. Da ciò si ricava che la funzione $P_e(\psi, \phi)$, finalizzata al calcolo puntualmente della probabilità di esplosione di una nave sapendo il suo valore iniziale di corazza ϕ e il danno da essa subito ψ, è:

$$P_e(\psi, \phi) = \begin{cases} 0 & se\ \frac{\psi}{\phi} < 0.5 \\ \frac{\psi}{\phi} & altrimenti \end{cases}$$

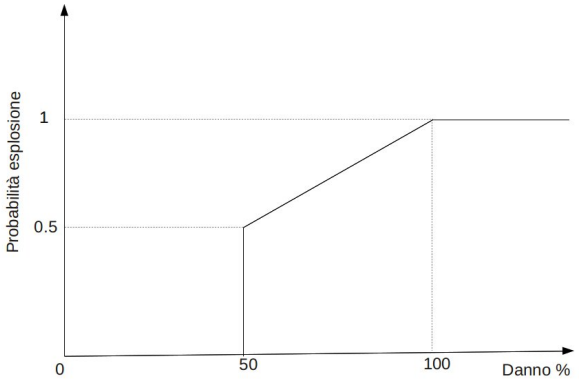


Figura 2.1: Grafico correlante la probabilità di esplosione con il danno %

2.3.6 Elaborazione del modello di calcolo per le navi esplose

Usando la funzione $P_e(\psi, \phi)$ possiamo cercare di stimare il numero di navi che esplodono. Dalla matrice precedentemente creata, possiamo ricavare il danno puntuale che riceve ogni nave il quale dovrà essere aggiunto a quello medio per poter stimare il danno effettivo che ogni nave subisce. Con entrambi questi valori possiamo calcolare il danno subito da ogni nave e da esso possiamo ricavare la seguente somma di variabili aleatorie:

$$\sum_{i=1}^{\xi_h} \mathcal{B}(P_e(\zeta_h + \omega_h(i), \phi_h))$$

Per costruzione, si può notare che le variabili aleatorie sono tutte indipendenti ma non identicamente distribuite. A causa di questo non possiamo applicare il teorema del limite centrale[3] "classico" perché esso prevede di utilizzare variabili aleatorie iid. Fortunatamente esiste una espressione più generale del sopracitato teorema che qui riportiamo:

Siano $\mathcal{X}_1, \mathcal{X}_2, ..., \mathcal{X}_i$ t.c. $\mathcal{X}_i \sim \mathcal{B}(p_i)$ indipendenti

allora per n sufficientemente grande è approssimativamente vero che:

$$\sum_{i=1}^{n} \mathcal{X}_i \sim \mathcal{N}(\sum_{i=1}^{n} p_i, \sum_{i=1}^{n} p_i(1-p_i))$$

Grazie a questo teorema, possiamo comunque stimare in modo sufficientemente preciso la situazione.

Possiamo notare inoltre, che anche se le variabili non sono uniformemente distribuite, c'è comunque una certa regolarità in esse che ci permette di riscrivere la formula qui sopra come:

$$\sum_{i=1}^{m} n_i\mathcal{X}_i \sim \mathcal{N}(\sum_{i=1}^{m} n_i p_i, \sum_{i=1}^{m} n_i p_i(1-p_i))$$

dove abbiamo indicato con n_i il numero di variabili aleatorie identicamente distribuite.

Osservando la scrittura sopra esposta, e ricordando la regola sulla combinazione lineare di variabili aleatorie normali indipendenti[3]:

Siano $\mathcal{X}_1, \mathcal{X}_2, ..., \mathcal{X}_i$ v.a. $\mathcal{X}_i \sim \mathcal{N}(\mu_i, \sigma_i^2)$ indipendenti allora

$$\mathcal{Y} = \sum_{i=1}^{n} \mathcal{X}_i = \mathcal{N}(\sum_{i=1}^{n} \mu_i, \sum_{i=1}^{n} \sigma_i^2)$$

Giunti a questo risultato, concludiamo, che è possibile calcolare il numero di navi esplose applicando più volte il teorema del limite centrale sui singoli blocchi logici presenti nella matrice ω_h inoltre, potendo controllare anche la varianza, si può considerare di imporre una percentuale di esplosione compresa esempio fra 0.4 e 0.6 e quindi valutare il numero di navi esplose in modo da generare un comportamento più incerto.

Dopo queste considerazioni, siamo ora in grado di stimare il valore medio $E_d(h)$ e la varianza delle navi esplose $var_d(h)$:

$$E_d(h) = \sum_{i=1}^{xR} \omega_h(i,1) P_e(\zeta_h + \omega_h(i), \phi_h)$$

$$var_d(h) = \sqrt{\sum_{i=1}^{xR} \omega_h(i,1) P_e(\zeta_h + \omega_h(i), \phi_h)[1 - P_e(\zeta_h + \omega_h(i), \phi_h)]}$$

Trovati questi valori, si può procedere con l'aggiornamento delle navi ancora attive per il prossimo round e si può calcolare il numero di navi esplose con il procedimento sopra descritto.

2.4 Implementazione dell'algoritmo

A seguito dell'analisi probabilistica dell'algoritmo, si propone una soluzione software del problema. Il linguaggio usato risulta il PHP5 che pur sacrificando la velocità, è molto indicato nello sviluppo di un BBMOG.

Inizialmente si riporta l'interfaccia dati atta a garantire la compatibilità con il motore d'attacco di tutte le possibili navi e o strutture che si vogliono far partecipare al combattimento.

Codice 2.1: Interfaccia Principale

```
interface CombatStructure {
    public function getType();
    public function getAttack();
    public function getIntegrity();
    public function getShield();
    public function getQuantity();
    public function calculateMeansShots($fleet);
    public function getMeansShots();
    public function inflictDamage($damage,$shots,$totalAlly);
    public function finalizeRound($calculateDamage=true);
    public function getAgainstCount();
    public function getShieldReflect();
}
```

In questa interfaccia possiamo notare tutti i metodi essenziali per gestire la simulazione. Si fa presente inoltre la presenza di 3 metodi particolari: calculateMeansShots, inflictDamage,finalizeRound che si vuole analizzare in dettaglio rappresentando l'implementazione del modello sopra discusso.

2.4.1 Calcolo dei colpi medi

Il calcolo dei colpi medi è svolto dalla funzione calculateMeansShots($fleet) che ci si presta ad analizzare.

Codice 2.2: algoritmo di calcolo dei colpi medi

```
public function calculateMeansShots($fleet){
  $this->totalShip = 0;
  foreach ($fleet as $value) {
    $this->totalShip+=$value->getQuantity();
  }
  $porcent=0;
  foreach ($fleet as $value){
    $rf = ShipList::getRf($this->getType(),$value->getType());
    $porcent+=($rf -1)/$rf*$value->getQuantity()/$this->totalShip;
  }
  $porcent=1-$porcent;
  $this->meansShot = round(1 / $porcent ) * $this->getQuantity();
}
```

Con il primo ciclo andiamo a valutare la quantità di navi nemiche presenti. Successivamente, si recupera il valore di rapid fire correlato a questa nave attaccante e si calcola, come sopra scritto, il valore di probabilità ad esso associato. Nelle ultime 2 righe viene calcolata la probabilità di non risparare e successivamente invertendo quest'ultima si ottengono i colpi medi sparati da una singola nave che moltiplicati per il numero di navi della stessa categoria ci fornisce il numero totale dei colpi effettuati.

2.4.2 L'attacco: calcolo degli scudi

Si riporta parte del metodo inflictDamage finalizzato ad elaborare l'attenuazione dell'attacco fornita dagli scudi.

Codice 2.3: algoritmo di calcolo dell'effetto attenuante degli scudi

```
public function inflictDamage($damage, $shots, $totalAlly){
  $qnt=$this->getQuantity();
  $shots=round($shots * $qnt / $totalAlly);
  if($this->particle > 0 && $this->getShield() > 0)}
    if ($damage>($this->getShield() / 100)){
      $killp = floor($damage * 100 / $this->getShield());
      if ($shots > $qnt){
        if (($shots*$killp) > ($this->Particle*$qnt)){
          $shots-=ceil($this->particle/$killp*$qnt);
          $this->particle = 0;
        }
        else{
          $this->particle=ceil(($this->particle*$qnt-$killp*$shots)/$qnt);
          $this->shieldReflect+=$damage*$shots;
          $damage=$shots=0;
        }
      }
      else{
        if($killp>$this->particle){
          $this->shieldp=ceil(($this->particle*$qnt-$killp*$shots)/$qnt);
          $damage=floor($damage-($this->getShield()*$this->particle/100));
        }
        else{
          $this->particle=ceil(($this->particle*$qnt-$killp*$shots)/$qnt);
          $damage=$shots=0;
        }
      }
    }
    else{
      $damage = $shots=0;
    }
  }
[...]
```

2.4.3 L'attacco: calcolo delle corazze

In questa parte finale del metodo inflictDamage, vengono calcolati i danni medi dell'attacco e salvati nella variabile meanDamage

Codice 2.4: algoritmo di calcolo dei danni alle corazze nel caso medio

```
public function inflictDamage($damage, $shots, $totalAlly){
[...]
if($damage>0 && $shots){
  $shotr = $shots % $qnt;
  $shots = floor($shots/$qnt);
  if(isset($this->meanDamage))
    $this->meanDamage=$this->meanDamage+$damage*$shots;
  else
    $this->meanDamage=$damage*$shots;
  if($shotr)
    $this->inflictSpecificDamage($shotr,$damage);
}
```

Successivamente, gli spari residui vengono affidati al metodo inflictSpecificDamage che inserisce lo specifico danno nell'array contenente i danni puntuali delle navi

Codice 2.5: algoritmo di calcolo delle corazze nel caso puntuale

```
private function inflictSpecificDamage($shotr,$damage){
  while($shotr){
    foreach ($this->damage as $index => $power) {
      if($index > $this->pointer) {
        $size = $index - $this->pointer;
        if($size > $shotr) {
          $this->damage[$this->pointer+$shotr]=$damage+$power;
          $this->pointer = $shotr + $this->pointer;
          $shotr = 0;
        }else{
          $this->damage[$index]+=$damage;
          $this->pointer = $index;
          $shotr-=$size;
        }
      }
      if(!$shotr)
        break;
    }
    ksort($this->damage);
    if($shotr)
      $this->pointer = 0;
  }
}
```

Inizialmente l'array damage contiene solamente una cella avete come indice il numero massimo delle navi e come valore il numero di danni iniziali ossia 0.

Man mano che si infliggono danni, si inseriscono altri indici intermedi nel vettore fino a quando il puntatore non supera il numero di navi. Arrivati a questa situazione il puntatore viene azzerato e si procede a sommare i successivi danni nuovamente dall'inizio dell'array.

Si riporta sotto, un possibile stato parziale dell'array damage dove ci sono 1000 navi di cui 500 con danno 10, e 175 con danno 95:

$$\begin{bmatrix} 500 & 10 \\ 675 & 95 \\ 1000 & 0 \end{bmatrix}$$

2.4.4 Calcolo dei danni

In quest'ultimo metodo vengono calcolati i danni e le navi esplose durante il round. La funzione calcola le navi esplose medie, e successivamente aggiorna il vettore dei danni. Il prossimo miglioramento prevede di aggiungere in questa funzione anche il calcolo della varianza per far oscillare la media di un valore predeterminato.

Codice 2.6: algoritmo di calcolo delle navi esplose

```
public function finalizeRound(){
  $death = 0;
  $newdamage = array();
  $prequantity = 0;
  foreach ($this->damage as $quantity => $value) {
    $realquantity = $quantity - $prequantity;
    $reld = ($value + $this->meanDamage) / $this->integrity;
    $reld = $reld > 1 ? 1 : $reld;
    if($reld>0.5)
      $prob = $reld;
    else
      $prob=0;
    $death+=round($realquantity * $prob);
    if($death != $quantity)
      $newdamage[$quantity - $death] = $value;
    $prequantity = $quantity;
  }
  $this->damage=$newdamage;
  $this->round++;
  $this->pointer = 0;
  $this->quantity[$this->round]=$this->quantity[$this->round-1]-$death;
  $this->particle = 100;
}
```

2.4.5 Implementazione del round

In quest'ultimo spezzone di codice viene riportato l'algoritmo che unisce tutte le funzioni e le fa operare tra loro. Questo codice ha infatti il compito di richiamare i metodi sopra descritti nel giusto ordine al fine di poter svolgere concretamente l'intera simulazione dell'algoritmo di attacco

Codice 2.7: algoritmo di calcolo di un round

```
public function executeRound() {
  if(!$this->result) {
    try{
      foreach ($this->attacker as $type) {
        $type->calculateMeansShots($this->defender);
        $this->atkShots+=$type->getMeansShots();
        $this->atkAttack+=$type->getAttack() * $type->getMeansShots();
        $def["ships"] = $type->getAgainstCount();
      }
    }catch(Exception $e){
      $defDied = true;
    }
    try{
      foreach($this->defender as $type){
        $type->calculateMeansShots($this->attacker);
        $this->defAttack+=$type->getAttack()*$type->getMeansShots();
        $this->defShots+=$type->getMeansShots();
        $atk["ships"]=$type->getAgainstCount();
      }
    }catch(Exception $e){
      $atkDied = true;
    }
    if($atkDied && $defDied){
      $this->result = 3;
    }elseif($defDied){
      $this->result = 1;
    }elseif($atkDied){
      $this->result = 2;
    }
    if($this->result){
      foreach ($this->defender as $defType)
        $defType->finalizeRound(false);
      foreach ($this->attacker as $atkType)
        $atkType->finalizeRound(false);
      return;
    }
```

In questa prima parte del codice, possiamo notare il calcolo dei colpi medi per tutte le navi e la valutazione della sconfitta di una delle due parti.

Durante il calcolo dei colpi medi, infatti, si è colta l'occasione per calcolare il numero di navi ancora attive, se questo numero scende a zero,

l'algoritmo lancia un'eccezione la quale viene catturata e avvisa che si sono verificate le condizioni per decretare un vincitore.

Codice 2.8: parte dell'algoritmo atto a elaborare i danni

```
    foreach($this->attacker as $atkType){
      if($atkType->getQuantity() > 0) {
        foreach ($this->defender as $type) {
          if ($type->getQuantity() > 0) {
            $atkType->inflictDamage($type->getAttack(),
                $type->getMeansShots(),$type->getAgainstCount());
          }else{
            break;
          }
        }
      }
    }
    foreach ($this->defender as $defType){
      if ($defType->getQuantity() > 0){
        foreach ($this->attacker as $type){
          if ($type->getQuantity() > 0){
            $defType->inflictDamage($type->getAttack(),
              $type->getMeansShots(),$type->getAgainstCount());
          }else{
            break;
          }
        }
      }
    }
    foreach($this->defender as $defType) {
      if($defType->getQuantity() > 0) {
        $this->defShield += $defType->getShieldReflect();
        $defType->finalizeRound();
      }
    }
    foreach ($this->attacker as $atkType) {
      if ($atkType->getQuantity() > 0) {
        $this->atkShield += $atkType->getShieldReflect();
        $atkType->finalizeRound();
      }
    }
  }
}
```

In quest'ultima parte invece vengono richiamate inizialmente le funzioni atte a infliggere il danno sulle varie flotte e successivamente vengono calcolati i danni e terminato il round. Queste funzioni lavorano in un ciclo che scorre tutti i vari tipi di navi e richiama su di essi i vari metodi se la quantità di navi presenti è diversa da zero.

Capitolo 3

Conclusioni

In quest'ultimo capitolo riportiamo alcuni dati sull'algoritmo al fine di valutarne l'efficienza e le performance.

Dal punto di vista computazionale, l'algoritmo presenta una complessità lineare rispetto al tipo di navi.

Dalle simulazioni svolte si è osservato che nel caso peggiore il tempo di esecuzione si attesta sull'ordine dei $200mS$ un tempo assolutamente ragionevole considerando che il php è un linguaggio interpretato e che quindi non è ottimale per questo fine. Nel caso medio-complesso invece, il tempo di esecuzione si attesta sui $50mS$ con un occupazione di memoria totale (comprensiva anche del framework symfony) di $25Mb$.

Possiamo affermare che le prestazioni dell'algoritmo è accettabile anche se si possono inserire alcune ulteriori migliorie ma già allo stato attuale è utilizzabile in contesti intensivi.

3.1 Simulazioni

In questa sezione vogliamo simulare alcuni scontri significativi ed analizzare i risultati al fine di valutare se il sistema si comporta come voluto. Utilizzeremo delle composizioni di navi particolari al fine di valutare obbiettivamente i risultati dell'algoritmo, infatti, essendo consapevoli delle caratteristiche delle navi ipotizzeremo il comportamento dell'algoritmo e poi verificheremo se le nostre ipotesi sono in linea con i risultati e in caso contrario cercheremo di capire perché i risultati sono diversi da quelli attesi.

Qui sotto si riporta la tabella contenente i valori di rapid fire per le tipologie di navi che intendiamo utilizzare.

Tabella 3.1: Valori di rapidfire

Tipo	Caccino	Caccione	Incro	BS.	BC.	Cozza	RIP
Caccino	1	1	1	1	1	1	1
Caccione	1	1	1	1	1	1	1
Incro	6	1	1	1	1	1	1
BS	1	1	1	1	1	1	1
BC	1	4	4	7	1	1	1
Cozza	1	1	2	2	2	1	1
RIP	2500	200	100	33	30	5	1

3.1.1 Flotte comparabili

In questa prima simulazione, analizzeremo uno scontro fra due flotte il cui numero totale di navi è comparabile. Le flotte sotto esposte sono state selezionate al fine di creare uno scontro tendenzialmente equilibrato. Considerando il numero di navi in gioco e i relativi rapid fire ad essi associati, ci si aspetta che lo scontro termini in pareggio o con la vittoria del difensore.

Tabella 3.2: Flotta dell'attaccante

Tipo	Quantità	Armi	Scudi	Corazze
Caccino	1 000 000	50	10	4 000
Caccione	500 000	150	25	10 000
Incro	250 000	400	50	27 000
BS	125 000	1 000	200	60 000
BC	62 500	400	700	70 000
Cozza	31 250	2 000	500	110 000

Tabella 3.3: Flotta del difensore

Tipo	Quantità	Armi	Scudi	Corazze
Caccione	500 000	150	25	10 000
Incro	250 000	400	50	27 000
BS	125 000	1 000	200	60 000
BC	62 500	400	700	70 000
Cozza	31 250	2 000	500	110 000
RIP	2500	200 000	50 000	9 000 000

Analizzando il numero di navi, infatti, notiamo che la flotta attaccante ne ha un gran numero di leggere che creeranno sicuramente uno scudo verso quelle più potenti. Dall'altro lato, però, l'attaccante dispone di un numero

di navi assolutamente uguale salvo quelle di tipo Caccini che sono contrapposte a quelle tipo RIP del difensore le quali, con il loro rapid fire, riescono sicuramente a porta una cadenza di fuoco e una potenza paragonabili se non superiori a quella dei Caccini.

Passando alla simulazione, ora riportiamo i dati restituiti al termine del primo round relativi alle statistiche del round ovvero: colpi sparati, potenza totale e colpi riflessi dagli scudi:

> La flotta in attacco spara 2 093 750 volte con una potenza di 543 750 000. Gli scudi del difensore assorbono 67 702 430 danni.
>
> La flotta in difesa spara 1 561 250 volte con una potenza di 56 550 000 000. Gli scudi dell'attaccante assorbono 37 483 626 danni.

Notiamo subito, che, nonostante il numero di navi assolutamente maggiore, la flotta in attacco spara meno della flotta in difesa, questo è coerente in quanto sia la nave di tipo Incro sia la nave di tipo RIP hanno un rapid fire verso la nave di tipo Caccino e questo incrementa di molto il numero di spari totali della flotta. Analogamente, la nave di tipo RIP non subisce fuoco rapido da nessuna delle navi attaccanti. Queste considerazioni sono confermate dai dati: il difensore spara proporzionalmente di più dell'attaccante infatti l'attaccante spara poco più di $2 \cdot 10^6$ volte con un numero di navi circa di $2 \cdot 10^6$ mentre il difensore spara $1.5 \cdot 10^6$ volte con un numero di navi di $1 \cdot 10^6$ quindi ogni nave spara mediamente 1 volta e mezza a differenza del singolo colpo dell'attaccante.

Altra cosa interessante si può notare guardando gli scudi. La tipologia di nave RIP ha scudi particolarmente elevati 50 000 e quindi ogni sua particella elementare ha valore 500 e conseguentemente esplode solo con un danno diretto di almeno 500 punti. In base alle regole sopracitate, essa riflette interamente qualsiasi danno proveniente dalle navi di tipo Caccino, Caccione, Incro, BC le quali hanno un danno singolo inferiore a 500. Questa osservazione si manifesta sui danni assorbiti infatti, il difensore assorbe il doppio dei danni dell'attaccante e questo è associabile a questo fenomeno. Per concludere analizziamo le navi residue alla fine del round. Esse sono riportate nella tabella sotto esposta.

Tipo	Quantità	Tipo	Quantità
Caccino	460 953	Caccione	340 861
Caccione	366 984	Incro	244 041
Incro	211 505	BS	125 000
BS	107 222	BC	62 500
BC	53 611	Cozza	31 250
Cozza	26 806	RIP	2 500

Tabella 3.4: Quantià residue della flotta dell'attaccante e del difensore

Come possiamo vedere, l'attaccante ha perso molte più navi del difensore. Il difensore, esce dal round in una situazione particolare: le navi più pericolose e in quantità minore non sono state ridotte nemmeno di una singola unità perché perché le navi più deboli e in numero maggiore hanno ricevuto molti più colpi facendo da scudo per le altre. Anche l'attaccante ha subito la stessa sorte solo che, in questo caso, le navi più corazzate hanno subito dei colpi diretti da parte delle RIP e non sono riuscite a difendersi, infatti queste navi con la loro immensa potenza di fuoco riescono a distruggere praticamente ogni ostacolo sulla loro strada e grazie a questa loro peculiarità hanno permesso di fare la differenza riducendo in modo visibile anche le navi pesanti dando valorizzazione alle nostre ipotesi.

In questa simulazione si è potuto constatare che l'algoritmo si è comportato in modo ragionevole e quindi fa pensare che sia una soluzione accettabile per risolvere il problema.

3.1.2 Flotte non comparabili

In questa sezione analizziamo il comportamento dell'algoritmo con flotte non paragonabili, ossia si valuterà il comportamento e il risultato dell'algoritmo in un caso limite con particolare interesse soprattutto quando gli attacchi vengono deflessi dagli scudi. Le flotte che andremo ad analizzare sono così composte:

Tabella 3.5: Flotta dell'attaccante

Tipo	Quantità	Armi	Scudi	Corazze
Caccino	1 000 000 000 000	50	10	4 000

Tabella 3.6: Flotta del difensore

Tipo	Quantità	Armi	Scudi	Corazze
Incro	300 000 000 000	400	50	27 000

Notiamo che le flotte in gioco sono completamente sbilanciate, i Caccini sono tre volte più numerose del numero di Incro ma, questi ultimi hanno un notevole fuoco rapido contro i Caccini oltre ad un numero di armi, scudi e corazze decisamente superiori.

Si ritiene sensato aspettarsi che l'incontro finisca in equilibrio o con la vittoria del difensore. Spostandoci ad analizzare le statistiche di fine round notiamo questi valori:

> La flotta in attacco spara 1 000 000 000 000 volte con una potenza di 50 000 000 000 000 Gli scudi del difensore assorbono 15 000 000 000 000 danni.

> La flotta in difesa spara 1 800 000 000 000 volte con una potenza di 7 200 000 000 000 000 Gli scudi dell'attaccante assorbono 10 000 000 000 000 danni.

Notiamo, che l'attaccante spara con una cadenza di fuoco e con una potenza decisamente inferiori al difensore. Un'ulteriore informazione che è stata fornita dal rapporto e che qui semplicemente citiamo, riguarda il numero di navi esplose. Le navi esplose al primo round sono zero, successivamente notiamo che anche nel round due e tre non abbiamo nessuna vittima e poi, nel round 3 notiamo che la flotta viene attaccante viene dimezzata e viene annientata nel giro di 2 round.

Questo comportamento che a prima vista sembra anomalo, è spiegabile analizzando il modo in cui le flotte subiscono i danni. Fino a metà della loro integrità le navi restano completamente intatte, appena viene superata questa soglia hanno una probabilità pari a 0.5 di esplodere e infatti al 4 round la flotta in attacco viene dimezza in quanto ha superato suddetta soglia. Il round successivo porta all'annientamento della flotta dovuto all'aumento dei danni inflitti mentre il difensore non perde nessuna unità in quanto i Caccini non riesco a danneggiare le navi in difesa al punto tale da comprometterne la sicura salvezza.

Dopo questa veloce analisi possiamo concludere che anche questa simulazione si è sviluppata secondo i criteri da noi imposti e che quindi possiamo ritenere con buona approssimazione che l'algoritmo si comporti effettivamente nel modo voluto.

3.2 Conclusioni

Lo scopo che si era prefissa questa tesi era di riuscire a modellare e realizzare un algoritmo per il gioco Noirblack che fosse il più possibile conforme al browser game da cui prende spunto ma che allo stesso tempo garantisse degli standard prestazionali adeguati. Dopo un'opportuna rielaborazione, si è potuto ricavare un algoritmo sufficientemente leggero, veloce e contemporaneamente si è dimostrato sufficientemente concorde agli standard richiesti. Concludendo, si pensa che questa sia una buona base di partenza per garantire alla piattaforma in questione un buon algoritmo di attacco, il prossimo passo che si deve compiere è cercare dei valori opportuni che caratterizzino le navi in modo da rendere la struttura il più possibile bilanciata al fine di evitare che emergano navicelle più forti di altre facendo risaltare le doti strategiche di ogni giocatore e rendendo ogni singola unità essenziale e vitale.

Un possibile miglioramento dell'algoritmo è introdurre delle stime basate su media e varianza in accordo con una variabile aleatoria $\mathcal{N}(0, 1)$ come detto nella costruzione del modello. Si ritiene comunque che l'algoritmo allo

stato attuale sia una valida soluzione al problema che si voleva analizzare e risolvere.

Bibliografia

[1] Nevio Benvenuto and Michele Zorzi University of Padova, Principle of Communications Networks and Systems

[2] M.T. Goodrich, R. Tamassia (2010) Data Structures and Algorithms in Java, 5th ehdition, New York, John Wiley & Sons.

[3] Sheldon M. Ross - Edizione italiana a cura di Carlo Mariconda e Marco Ferrante, Calcolo delle Probabilità.