**POLITECNICO DI MILANO**
**Corso di Laurea Magistrale in Ingegneria Informatica**
**Dipartimento di Elettronica e Informazione**


# Evoluzioni di mappe bilanciate per FPS con algoritmi evolutivi

**Relatore: Prof. Pier Luca Lanzi**
**Correlatore: Ing. Daniele Loiacono**

**Tesi di Laurea di:**
**Riccardo Stucchi, matricola: 783044**

**Anno Accademico 2012-2013**

*Ai miei genitori, a Jessica, l'amore della mia vita, e a mia nonna, che è sempre con me.*

# Prefazione

Il bilanciamento delle componenti dei videogiochi è sempre stato uno dei temi principali che ne interessano la fase di sviluppo, soprattutto per le componenti utilizzate nelle modalità a più giocatori. Nei videogiochi multigiocatore sparatutto in prima persona una delle componenti che più influenza il bilanciamento della partita è la mappa nella quale quest'ultima si svolge.

L'obiettivo di questo lavoro è generare mappe per un videogioco sparatutto in prima persona che, data una situazione di svantaggio di un giocatore rispetto ad un altro, rendano bilanciata la partita tra due giocatori. Utilizzando la generazione procedurale di contenuti con algoritmi evolutivi generiamo mappe per il videogioco *Cube 2*, partendo da diverse configurazioni iniziali e mirando a massimizzare il bilanciamento della partita. Impostiamo, per ogni esperimento, la configurazione iniziale tramite due parametri per ogni giocatore, il livello di abilità e l'arma utilizzata. Descriviamo i risultati dei nostri esperimenti condotti sull'evoluzione delle mappe per varie configurazioni, mostrando ed analizzando le mappe ottenute.

# *Ringraziamenti*

*Vorrei cominciare ringraziando il prof. Pier Luca Lanzi per la sua disponibilità negli orari e nei giorni più improbabili, per avermi consigliato in questi ultimi due anni di carriera scolastica e per avermi insegnato tanto.*

*Ringrazio l'Ing. Daniele Loiacono per avermi aiutato nella fase iniziale della tesi e per avermi messo a disposizione il server per gli esperimenti.*

*Ringrazio mia mamma per essermi sempre stata vicina, nonostante il mio nervosismo ed i repentini cambi di umore, per avermi sempre aiutato e per avermi fatto sorridere ogni giorno.*

*Ringrazio mio papà perché mi ha sempre appoggiato nelle mie decisioni, per aver sempre trovato il tempo di ascoltarmi nonostante i suoi impegni e per avermi dato un modello da seguire.*

*Infine ringrazio Jessica, perché se non ci fosse stata lei tutto questo non sarebbe stato possibile. La ringrazio per essermi sempre stata vicina, per avermi sopportato sempre, e per darmi ogni giorno la forza per superare ogni ostacolo.*

# Elenco delle figure


3.1 Mappa generata con rappresentazione *All-White* . . . . . 36
3.2 Mappa generata con rappresentazione *All-Black* . . . . . 36
3.3 Mappa generata con rappresentazione *Grind* . . . . . 38
3.4 Mappa generata con rappresentazione *Random-Digger* . . . . . 38

4.1 Heatmap del *Kill Ratio* per l'arma *Pistola* . . . . . 42
4.2 Heatmap della *Diff* per l'arma *Pistola* . . . . . 42
4.3 Heatmap del *Kill Ratio* per l'arma *Bazooka* . . . . . 43
4.4 Heatmap della *Diff* per l'arma *Bazooka* . . . . . 43
4.5 Heatmap del *Kill Ratio* per l'arma *Rifle* . . . . . 44
4.6 Heatmap della *Diff* per l'arma *Rifle* . . . . . 44
4.7 Heatmap del *Kill Ratio* per l'arma *Motosega* . . . . . 45
4.8 Heatmap della *Diff* per l'arma *Motosega* . . . . . 45
4.9 Heatmap del *Kill Ratio* per l'arma *Lanciagranate* . . . . . 46
4.10 Heatmap della *Diff* per l'arma *Lanciagranate* . . . . . 46
4.11 Heatmap del *Kill Ratio* per l'arma *Mitragliatore* . . . . . 47
4.12 Heatmap della *Diff* per l'arma *Mitragliatore* . . . . . 47
4.13 Mappa vincitrice dell'evoluzione con divario di abilità bot 45 . . . 52
4.14 *heatmap* del *Kill Ratio* per ogni livello dei due bot. Dati raccolti sulla mappa vincitrice dell'evoluzione con *DiffLvBot*=45 . . . . . 53
4.15 *heatmap* dell'entropia delle uccisioni per ogni livello dei due bot. Dati raccolti sulla mappa vincitrice dell'evoluzione con *DiffLvBot*=45 . . . . . 53
4.16 Mappa vincitrice dell'evoluzione con divario di abilità bot 80 . . . 54
4.17 *heatmap* del *Kill Ratio* per ogni livello dei due bot. Dati raccolti sulla mappa vincitrice dell'evoluzione con *DiffLvBot*=80 . . . . . 55
4.18 *heatmap* dell'entropia delle uccisioni per ogni livello dei due bot. Dati raccolti sulla mappa vincitrice dell'evoluzione con *DiffLvBot*=80 . . . . . 55

4.19 *heatmap* dell'entropia delle uccisioni per ogni livello dei due bot. Dati raccolti sulla mappa vuota. . . . . 56
4.20 Esempi di entropia delle uccisioni per *DivLvBot*=70 nelle 3 analisi 57
4.21 Esempi di entropia delle uccisioni per *DivLvBot*=80 nelle 3 analisi 57
4.22 Esempi di entropia delle uccisioni per *DivLvBot*=90 nelle 3 analisi 58
4.23 Etropia delle uccisioni media per ogni divario di bot nelle 3 analisi 58

5.1 *Matrix crossover* tra due genomi . . . . . 63
5.2 Fitness massima e media con rappresentazione *All-Black* nelle generazioni, Arma *Rifle*, Abilità 50-50 . . . . . 64
5.3 Fitness massima e media con rappresentazione *Grind* nelle generazioni, Arma *Rifle*, Abilità 50-50 . . . . . 64
5.4 Mappe vincitrici dell'evoluzione con rappresentazione *All-Black*, arma *Rifle*, abilità 50-50 . . . . . 65
5.5 Fitness massima e media con rappresentazione *All-Black* nelle generazioni, Arma *Motosega*, Abilità 50-50 . . . . . 66
5.6 Fitness massima e media con rappresentazione *Grind* nelle generazioni, Arma *Motosega*, Abilità 50-50 . . . . . 66
5.7 Mappe vincitrici dell'evoluzione con arma *Motosega*, abilità 50-50 67
5.8 Fitness massima e media con rappresentazione *All-Black* nelle generazioni, Arma *Rifle*, Abilità 35-80 . . . . . 68
5.9 Fitness massima e media con rappresentazione *Grind* nelle generazioni, Arma *Rifle*, Abilità 35-80 . . . . . 68
5.10 Mappe vincitrici dell'evoluzione con arma *Rifle*, abilità 35-80 . . 69
5.11 Heatmap delle uccisioni e delle morti della mappa migliore con rappresentazione *All-Black*, arma *Rifle*, abilità 35-80 . . . . . 69
5.12 Fitness massima e media con rappresentazione *All-Black* nelle generazioni, Arma *Motosega*, Abilità 15-95 . . . . . 71
5.13 Fitness massima e media con rappresentazione *Grind* nelle generazioni, Arma *Motosega*, Abilità 15-95 . . . . . 71
5.14 Mappe vincitrici dell'evoluzione con arma *Motosega*, abilità 15-95 72
5.15 Fitness massima e media con rappresentazione *All-Black* nelle generazioni, Armi *Rifle-Motosega*, Abilità 50-50 . . . . . 74
5.16 Fitness massima e media con rappresentazione *Grind* nelle generazioni, Armi *Rifle-Motosega*, Abilità 50-50 . . . . . 74
5.17 Mappe vincitrici dell'evoluzione con armi *Rifle-Motosega*, abilità 50-50 . . . . . 75
5.18 Fitness massima e media con rappresentazione *All-Black* nelle generazioni, Armi *Rifle-Lanciagranate*, Abilità 50-50 . . . . . 76

5.19 Fitness massima e media con rappresentazione *Grind* nelle generazioni, Armi *Rifle-Lanciagranate*, Abilità 50-50 . . . . . 76
5.20 Mappe vincitrici dell'evoluzione con armi *Rifle-Lanciagranate*, Abilità 50-50 . . . . . 77

6.1 Fitness massima e media con rappresentazione *All-Black* nelle generazioni, armi *Rifle-Motosega*, Abilità 80-20 . . . . . 83
6.2 Mappa vincitrice dell'evoluzione con rappresentazione *All-Black*, Armi *Rifle-Motosega*, Abilità 80-20 . . . . . 83
6.3 Fitness massima e media con rappresentazione *Grind* nelle generazioni, Armi *Rifle-Motosega*, Abilità 80-20 . . . . . 84
6.4 Mappa vincitrice dell'evoluzione con rappresentazione *Grind*, Armi *Rifle-Motosega*, abilità 80-20 . . . . . 84
6.5 Fitness massima e media con rappresentazione *All-Black* nelle generazioni, Armi *Rifle-Motosega*, Abilità 20-80 . . . . . 86
6.6 Fitness massima e media con rappresentazione *Grind* nelle generazioni, Armi *Rifle-Motosega*, Abilità 20-80 . . . . . 86
6.7 Mappe vincitrici dell'evoluzione con armi *Rifle-Motosega*, Abilità 20-80 . . . . . 87
6.8 Fitness massima e media con rappresentazione *All-Black* nelle generazioni, Armi *Rifle-Lanciagranate*, Abilità 80-20 . . . . . 89
6.9 Fitness massima e media con rappresentazione *Grind* nelle generazioni, Armi *Rifle-Lanciagranate*, Abilità 80-20 . . . . . 89
6.10 Mappe vincitrici dell'evoluzione con armi *Rifle-Lanciagranate*, Abilità 80-20 . . . . . 90
6.11 Heatmap delle uccisioni per le mappe vincitrici dell'evoluzione con armi *Rifle-Lanciagranate*, Abilità 80-20 . . . . . 90
6.12 Fitness massima e media con rappresentazione *All-Black* nelle generazioni, Armi *Rifle-Lanciagranate*, Abilità 20-80 . . . . . 92
6.13 Fitness massima e media con rappresentazione *Grind* nelle generazioni, Armi *Rifle-Lanciagranate*, Abilità 20-80 . . . . . 92
6.14 Mappe vincitrici dell'evoluzione con armi *Rifle-Lanciagranate*, Abilità 20-80 . . . . . 93

# Elenco delle tabelle

4.1 *DiffLvBot* massima (minima) sotto (sopra) la quale il *Kill Ratio* è compreso nella soglia specificata . . . . . 49
4.2 Valori massimi di *Kill Ratio* e *Diff* per ogni arma . . . . . 50

5.1 Risultati ottenuti sul problema del bilanciamento negli esperimenti effettuati nel capitolo 5 . . . . . 79

6.1 Risultati ottenuti sul problema del bilanciamento negli esperimenti effettuati nel capitolo 6 . . . . . 94

# Indice

**Elenco delle figure** **11**

**Elenco delle tabelle** **13**

**1 Introduzione** **17**

**2 Stato dell'arte** **19**

2.1 Generazione Procedurale di Contenuti . . . . . 19

2.2 Generazione procedurale di contenuti con algoritmi di ricerca . . 21

2.2.1 Funzione di Fitness . . . . . 23

2.2.2 Generazione procedurale di mappe con algoritmi di ricerca 25

2.3 Bilanciamento della difficoltà nei videogiochi . . . . . 28

2.3.1 Bilanciamento di videogiochi a più giocatori . . . . . 28

2.3.2 Bilanciamento delle mappe nei videogiochi . . . . . 30

2.4 Sommario . . . . . 31

**3 Evoluzione di mappe bilanciate per FPS** **33**

3.1 I nostri obiettivi . . . . . 33

3.2 Cube 2 . . . . . 34

3.3 Le rappresentazioni delle mappe per l'evoluzione . . . . . 36

3.4 Sommario . . . . . 38

**4 Analisi dei bot in Cube 2** **39**

4.1 I bot di Cube 2 . . . . . 39

4.2 Analisi delle differenze di livello tra bot . . . . . 40

4.3 Analisi della soglia per il *Rifle* . . . . . 50

4.3.1 Evoluzione con *Rifle* e configurazione 35-80 . . . . . 52

4.3.2 Evoluzione con *Rifle* e configurazione 15-95 . . . . . 52

4.3.3 Confronto delle evoluzioni con arma *Rifle* . . . . . 54

4.4 Sommario . . . . . 59

**5 Esperimenti su singolo parametro** **61**
5.1 Introduzione . . . . . 61
5.2 Bilanciamento per bot con arma uguale . . . . . 63
5.2.1 Arma *Rifle*, Abilità bot 50-50 . . . . . 63
5.2.2 Arma *Motosega*, Abilità bot 50-50 . . . . . 65
5.2.3 Arma *Rifle*, Abilità bot 35-80 . . . . . 67
5.2.4 Arma *Motosega*, Abilità bot 15-95 . . . . . 70
5.3 Bilanciamento per bot con abilità uguale ma arma diversa . . . . 73
5.3.1 Armi *Rifle-Motosega*, Abilità bot 50-50 . . . . . 73
5.3.2 Armi *Rifle-Lanciagranate*, Abilità bot 50-50 . . . . . 75
5.4 Sommario . . . . . 77

**6 Esperimenti su più parametri** **81**
6.1 Arma *Rifle-Motosega*, abilità differenti . . . . . 81
6.1.1 Armi *Rifle-Motosega*, Abilità bot 80-20 . . . . . 82
6.1.2 Armi *Rifle-Motosega*, Abilità bot 20-80 . . . . . 85
6.2 Arma *Rifle-Lanciagranate*, abilità differenti . . . . . 88
6.2.1 Armi *Rifle-Lanciagranate*, Abilità bot 80-20 . . . . . 88
6.2.2 Armi *Rifle-Lanciagranate*, Abilità bot 20-80 . . . . . 91
6.3 Sommario . . . . . 93

**7 Conclusioni e sviluppi futuri** **95**

**Bibliografia** **99**

# Capitolo 1

# Introduzione

Il bilanciamento delle componenti dei videogiochi è sempre stato uno dei temi principali che ne interessano la fase di sviluppo, soprattutto per le componenti utilizzate nelle modalità a più giocatori. Nei videogiochi multigiocatore sparatutto in prima persona una delle componenti che più influenza il bilanciamento della partita è la mappa nella quale quest'ultima si svolge. Per questo motivo, le grandi aziende videoludiche pianificano costose sessioni di test in modo da verificare l'effettivo bilanciamento della partita sulla mappa realizzata. Rendere automatico questo processo di verifica del bilanciamento equivarrebbe ad un risparmio economico (e temporale) ingente.

In questa tesi descriviamo i risultati degli esperimenti da noi svolti sulla generazione di mappe per il videogioco *Cube 2*, esperimenti finalizzati a massimizzare il bilanciamento delle partite tramite l'uso di algoritmi evolutivi. L'obiettivo di questo lavoro è stato quello di generare mappe per un videogioco sparatutto in prima persona che, data una situazione di svantaggio di un giocatore rispetto ad un altro, rendano bilanciato il combattimento tra i due giocatori. Per imporre lo svantaggio a un giocatore rispetto ad un altro abbiamo agito sull'arma data loro in dotazione e sulla loro abilità in combattimento. Attualmente, le ricerche effettuate in ambito accademico sull'utilizzo di algoritmi evolutivi per generare mappe che bilancino il combattimento tra giocatori di abilità differenti interessano il bilanciamento di sole mappe per videogiochi di strategia in tempo reale, la nostra tesi cerca quindi di dare una risposta concreta alla mancanza di studio relativo a questo argomento nei videogiochi sparatutto in prima persona.

I contributi originali di questa tesi sono il design di algoritmi per l'evoluzione di mappe per un videogioco sparatutto in prima persona che, partendo da una situazione iniziale sbilanciata, risultino bilanciate e dare una risposta alla seguente domanda: è possibile generare mappe per un videogioco sparatutto in prima persona che rendano bilanciato il combattimento tra due giocatori data

una situazione di svantaggio di uno rispetto all'altro?

Questa tesi è strutturata secondo il seguente schema:

Il **Capitolo 2** contiene una descrizione dello stato dell'arte ad oggi, introducendo la generazione procedurale di contenuti, la generazione procedurale di contenuti che utilizza algoritmi di ricerca e, infine, discutendo il concetto di bilanciamento della difficoltà nei videogiochi.

il **Capitolo 3** descrive i nostri obiettivi per questo lavoro, il videogioco utilizzato per i nostri esperimenti (Cube 2) e le 4 rappresentazioni da noi usate per generare le mappe.

il **Capitolo 4** analizza i *bot* di Cube 2, valutando quanto il livello di abilità tra *bot* influenzi il risultato finale della partita, approfondendo successivamente l'analisi su bot dotati di arma *rifle*.

il **Capitolo 5** mostra i risultati dei nostri esperimenti sul bilanciamento di partite in modalità *deathmatch* a due giocatori, analizzando situazioni in cui i *bot* sono dotati di arma diversa e abilità uguale oppure situazioni in cui i *bot* sono dotati di arma uguale e abilità diversa.

il **Capitolo 6** mostra i risultati dei nostri esperimenti avanzati sul bilanciamento di partite in modalità *deathmatch* a due giocatori, analizzando situazioni complesse nelle quali i *bot* sono sia dotati di arma diversa che di abilità differenti.

il **Capitolo 7** conclude questo lavoro, presentando le considerazioni finali e gli sviluppi futuri.

# Capitolo 2

# Stato dell'arte

In questo capitolo introduciamo la generazione procedurale di contenuti, o *Procedual Content Generation* (PCG). Successivamente descriviamo la PCG con algoritmi di ricerca, o *Search Based Procedural Content Generation* (SBPCG), e la sua applicazione per la creazione di mappe nell'ambito dei videogiochi. Infine discutiamo il concetto di bilanciamento della difficoltà nei videogiochi.

## 2.1 Generazione Procedurale di Contenuti

La generazione procedurale di contenuti si occupa di creare contenuti in modo automatico tramite l'uso di algoritmi. Nei videogiochi il termine *contenuto* identifica tutti gli aspetti del gioco che ne influenzano la giocabilità quali, ad esempio, armi, livelli, mappe, dialoghi o missioni. Dalla fine degli anni '80, la generazione procedurale di contenuti è stata utilizzata per generare quantità elevate di contenuti su computer che disponevano di potenza computazione limitata. Uno dei primi esempi di generazione procedurale in un videogioco è stato *Beneath Apple Manor*[1] (1978), gioco nel quale il giocatore doveva raggiungere la mela d'oro percorrendo 10 labirinti senza morire. In ogni labirinto, nemici e tesori venivano generati proceduralmente ad ogni nuova partita. *Rogue*[2] (1980) e *Hack*[3] (1982) condividevano le meccaniche di *Beneath Apple Manor* sia per quanto riguarda le meccaniche di gioco che per quelle di creazione di contenuti. Un altro esempio è stato *Elite*[4] (1984), un videogioco di commercio spaziale ambientato in un universo composto da 8 galassie, ognuna suddivisa in 256 pianeti. La posizione, i prezzi delle materie prime acquistabili, il nome e la descrizione di ogni pianeta erano generati da un algoritmo di generazione

[1] http://en.wikipedia.org/wiki/Beneath_Apple_Manor
[2] http://en.wikipedia.org/wiki/Rogue_(computer_game)
[3] http://en.wikipedia.org/wiki/Hack_(Unix_video_game)
[4] http://en.wikipedia.org/wiki/Elite_(video_game)

procedurale. In *The Sentinel*[5] (1986) il giocatore impersonava un robot libero di esplorare un ambiente tridimensionale. In questo contesto un algoritmo permetteva di creare 10000 livelli differenti racchiusi in un file di pochi *kilobyte*. Ogni livello consisteva in una mappa a scacchiera composta da colline, avvallamenti ed altri oggetti di scenario e non, con la quale il giocatore aveva la possibilità di interagire. In *Murder*[6] (1990), videogioco poliziesco in stile *Cluedo*[7], i parametri di gioco venivano cambiati in base al nome e alla data che venivano forniti ad inizio partita dal giocatore, in modo da aumentare o diminuire la difficoltà del gioco. Sebbene i computer non abbiano più problemi legati alle scarse risorse computazionali, la generazione procedurale di contenuti continua a mantenere grande importanza nel settore dei videogiochi per varie ragioni, prima tra tutte per contenere i costi di progettazione in fase di sviluppo dei videogiochi, sia in quelli a budget ridotto sia negli AAA [1]. In secondo luogo, la PCG viene utilizzata per creare elevati quantitativi di contenuto in modo da aumentare la longevità del videogioco, mantenendo alta la voglia di giocare da parte del giocatore. Infine la PGC viene utilizzata per rendere possibile la creazione di videogiochi che basano le proprie meccaniche su di essa. *Diablo*[8] (1996) e il sequel *Diablo 2*[9] (2000) utilizzano la PGC sia per la prima che per la seconda ragione, generando proceduralmente sotterranei, tesori, la posizione degli oggetti e alcune missioni [26]. In *Civilization IV*[10] (2005) viene generato il terreno di gioco dando la possibilità al giocatore di specificare vari parametri utili alla sua generazione tra cui, ad esempio, il numero di giocatori, il tempo atmosferico ed il tipo di terreno. *Borderlands*[11] (2009), videogioco sparatutto in prima persona, è un esempio di gioco che basa le proprie meccaniche sulla generazione procedurale di contenuti. Ogni arma ritrovata dal giocatore durante la partita viene generata proceduralmente da un algoritmo che ne seleziona i componenti (tra i quali mirino, canna e caricatore) conferendole un aspetto e delle proprietà uniche. *Minecraft*[12] (2011) è un videogioco ambientato in un mondo tridimensionale nel quale un algoritmo procedurale genera un insieme di cubi dalle diverse proprietà che, presi nell'insieme, formano il mondo esplorabile. In *The binding of Isaac*[13] (2011), videogioco di azione/esplorazione di labirinti, vengono utilizzati algoritmi di generazione procedurale per generare

[5] http://en.wikipedia.org/wiki/The_Sentinel_(video_game)
[6] http://pcg.wikidot.com/pcg-games:murder
[7] http://en.wikipedia.org/wiki/Cluedo
[8] http://en.wikipedia.org/wiki/Diablo_(video_game)
[9] http://us.blizzard.com/en-us/games/d2/
[10] http://en.wikipedia.org/wiki/Civilization_IV
[11] http://en.wikipedia.org/wiki/Borderlands_(video_game)
[12] https://minecraft.net
[13] http://en.wikipedia.org/wiki/The_Binding_of_Isaac_(video_game)

mappe, tesori e nemici.

La generazione procedurale di contenuti può essere di tipo *costruttivo* o *generazione-e-test*. In un approccio *costruttivo*, l'algoritmo di generazione procedurale genera il contenuto svolgendo delle operazioni di verifica per controllare che il contenuto generato sia corretto. Una volta generato, una persona valuterà il contenuto verificando che esso risulti idoneo allo scopo per cui è stato generato. In un approccio *generazione-e-test* la generazione procedurale si occupa sia del processo di generazione che di quello di valutazione. Una volta generato (dopo essere stato sottoposto ad operazioni di verifica come avviene per la PGC costruttiva), il contenuto viene valutato tramite una funzione che gli assegna un valore. Utilizzando in retroazione questo valore, l'algoritmo genera un nuovo contenuto basandosi sul punteggio assegnato al contenuto precedentemente generato, ottenendo un contenuto migliore ad ogni iterazione dell'algoritmo rispetto a quelli precedenti. Questo processo termina al raggiungimento di una determinata condizione (quale, ad esempio, il raggiungimento del numero massimo di iterazioni per l'algoritmo).

## 2.2 Generazione procedurale di contenuti con algoritmi di ricerca

Attualmente la generazione procedurale di contenuti si sta sviluppando su due fronti distinti. Mentre nell'ambito industriale quest'ultima viene utilizzata per permettere la creazione di quantità elevate di contenuto risparmiando risorse (in merito a tempo e costi di sviluppo), nell'ambito accademico i ricercatori si sono posti l'obiettivo di automatizzare il processo di valutazione dei contenuti generati. Da quest'ultimo ambito nasce la definizione di *generazione procedurale di contenuti con algoritmi di ricerca*, una generazione procedurale di contenuti di tipo *generazione-e-test* caratterizzata, oltre da quelle già presenti nella generazione procedurale, da due importanti caratteristiche:

- la funzione di valutazione, una volta generato il contenuto, gli assegna un valore reale o un vettore di valori reali. Questa funzione viene definita come *funzione di fitness* e la valutazione assegnata è definita come *valore di fitness*.

- la generazione di nuovi contenuti è condizionata dai valori di fitness assegnati ai contenuti precedentemente valutati. In questo modo è possibile creare nuovi contenuti con valore di fitness più elevato.

Per ottenere dei contenuti migliori sfruttando il valore di fitness assegnato ai contenuti precedentemente generati, è necessario l'uso di un algoritmo di ricerca. Quest'ultimo ci consente di individuare il contenuto migliore all'interno dello *spazio di ricerca*, lo spazio di tutti i possibili contenuti che l'algoritmo di generazione procedurale può generare. Solitamente gli algoritmi di ricerca usati nella generazione procedurale di contenuti con algoritmi di ricerca appartengono all'insieme degli *algoritmi evolutivi*[14] (EA). Questi sono algoritmi che implementano meccaniche basate sull'evoluzione biologica, come la riproduzione, la ricombinazione di cromosomi, la mutazione di geni e la selezione naturale.

Nel caso in cui venga usato un algoritmo evolutivo, un insieme di candidati (la *popolazione*) viene inizialmente creato in modo casuale. Ad ogni generazione, una volta assegnato il valore di fitness a tutti i contenuti candidati, i contenuti passano per una fase di *selezione*. I contenuti migliori vengono copiati nella nuova generazione, i peggiori scartati. I primi, con una certa probabilità, vengono poi ricombinati tra loro e/o mutati in modo da esplorare la spazio di ricerca basandosi sui migliori contenuti disponibili. Per una descrizione dettagliata del funzionamento degli EA è possibile consultare [17]. Negli algoritmi evolutivi e, di conseguenza, nella generazione procedurale di contenuti con algoritmi evolutivi vengono usate due rappresentazioni diverse del contenuto:

- una rappresentazione utilizzata dall'algoritmo di ricerca, definita come *genotipo*. Il genotipo è un vettore di parametri (che possono assumere valori interi, reali, binari, ecc.) che codifica il contenuto e sulla quale vengono eseguite le operazioni di selezione, ricombinazione e mutazione.

- una rappresentazione utilizzata per la valutazione da parte della funzione di fitness, definita come *fenotipo*. Il fenotipo è la rappresentazione finale del contenuto, ovvero, nel nostro caso, come il contenuto appare al giocatore.

Un algoritmo mappa tra loro le due rappresentazioni: dando in input all'algoritmo il genotipo, esso restituisce il fenotipo. Formalmente, definito X l'insieme di tutti i genotipi e Y l'insieme di tutti i fenotipi, abbiamo $y = f(x)$ dove $y \in Y$, $x \in X$ e la funzione $f : X \to Y$ è l'algoritmo che mappa le due rappresentazioni tra loro. Per esempio, supponiamo di cercare una buona mappa per un videogioco utilizzando la generazione procedurale di contenuti con algoritmi evolutivi. Il genotipo potrebbe essere un vettore di interi che identificano la posizione dei muri. Il fenotipo la mappa finale codificata dal genotipo. L'algoritmo che mappa le due rappresentazioni prenderebbe i valori presenti nel genotipo e, per ognuno di essi, inserirebbe un muro in una mappa inizialmente vuota.

[14] http://en.wikipedia.org/wiki/Evolutionary_algorithm

Uno dei principali problemi nell'uso della generazione procedurale di contenuti con algoritmi evolutivi, e in generale negli algoritmi evolutivi, è la scelta della rappresentazione da usare. Infatti, per permettere una ricerca efficiente nello spazio di ricerca, il vettore di parametri usato dall'algoritmo di ricerca deve avere la giusta dimensione [3]. Un vettore troppo piccolo non permette una rappresentazione adeguata del contenuto, un vettore troppo grande causa un uso non efficiente degli algoritmi di ricerca (fenomeno chiamato *curse of dimensionality*[15]). La scelta della rappresentazione è tuttora un tema centrale nella ricerca sugli EA. Per esempio, in [4] Ashlock et al. mettono a confronto quattro tipi di rappresentazioni diverse di mappe, valutandole e discutendone i pro e i contro.

Una delle proprietà base che la rappresentazione dovrebbe avere è la *località*: applicando un piccolo cambiamento nel genotipo si ha (mediamente) un cambiamento minimo nel fenotipo e nel valore di fitness. Un altro problema importante è la scelta della funzione di fitness con la quale valutare i candidati.

### 2.2.1 Funzione di Fitness

Come precedentemente accennato, una volta che il contenuto candidato viene generato è necessario valutarlo con una funzione di fitness. Questa funzione gli assegna un valore reale che fornisce un'indicazione di quanto il contenuto candidato sia idoneo rispetto a ciò che stiamo cercando. È possibile inoltre che la funzione di fitness assegni un vettore di valori reali al posto di un singolo valore, in questo caso parliamo di *funzione di ottimizzazione multi-obiettivo*, o MOEA (Multiobjective Evolutionary Algorithms) [6].

La funzione di fitness è quindi creata *ad-hoc* a seconda del contenuto che si vuole ottenere. Per crearla è necessario decidere cosa deve essere ottimizzato e come formalizzarlo. La formalizzazione (soprattutto nei videogiochi) risulta spesso complessa. Per esempio, è difficile formalizzare in una funzione uno stato emozionale del giocatore ed è altrettanto difficile trovare un buon modo per misurarlo. Una possibile soluzione è usare dati ricavabili direttamente dal contenuto candidato piuttosto che dati astratti (come, per esempio, le emozioni) che lo riguardano. In questo modo è possibile aggirare il problema.

Le funzioni di fitness possono essere classificate in base all'approccio con cui valutano i contenuti candidati [3]:

*Funzione di fitness* ***diretta***: In questo tipo di funzione, alcuni dati sono estratti direttamente dal contenuto generato (fenotipo) e mappati direttamente (o con pochi calcoli) in un valore di fitness. Per esempio, una funzione che calcola il

[15] http://en.wikipedia.org/wiki/Curse_of_dimensionality

numero di muri presenti in un labirinto è una funzione di fitness diretta. A sua volta, una funzione di fitness diretta può essere di due tipi:

- *theory-driven* (basata sulla teoria): in questo caso la funzione di fitness viene creata basandosi su un'intuizione e/o su delle teorie riguardanti cosa vogliono i giocatori in un particolare videogioco. Supponendo per esempio di dover generare le armi migliori per un videogioco, uno studio potrebbe suggerire l'uso di una fitness che valuti bene un'arma con alto ratio di fuoco.
- *data-driven* (bastata sui dati): una funzione di fitness di questo tipo viene creata basandosi sulle informazioni ricavate da un insieme di dati raccolti, ad esempio, tramite questionari o studi psicologici sui giocatori. I dati così ottenuti vengono analizzati ai fini di ottenere un modello che può aiutare a formalizzare anche concetti astratti come le emozioni.

*Funzione di fitness* ***basata sulla simulazione***: Non sempre è possibile creare una funzione di fitness diretta che valuti in modo significativo i contenuti candidati. A volte è necessario interagire con il videogioco direttamente per ottenere le informazioni necessarie ad una valutazione accurata. Le funzioni di fitness basate sulla simulazione si basano su questo: far giocare uno o più agenti artificiali (personaggi mossi dall'intelligenza artificiale) all'interno dell'ambiente di gioco in modo da testare il contenuto generato. Durante questo test vengono estratti i dati d'interesse che sono poi utilizzati dalla funzione di fitness per calcolare il valore di fitness del contenuto. Per esempio, la funzione che calcola la media dei punti ottenuti dall'agente dopo cinque partite è una funzione di fitness basata sulla simulazione.

A sua volta, all'interno di questo approccio possiamo classificare due tipi di funzioni di fitness basate sulla simulazione:

- *statica*: se l'agente che interagisce con l'ambiente di gioco non cambia durante il test del contenuto
- *dinamica*: se l'agente che interagisce con l'ambiente di gioco cambia durante il test del contenuto e la funzione di fitness ne tiene conto.

L'uso di una funzione di fitness basata sulla simulazione è, generalmente, molto più pesante computazionalmente rispetto all'uso di quella diretta, sia in termini di tempo che di spazio utilizzato in memoria. Nella funzione di fitness basata sulla simulazione, infatti, è necessario avviare il videogioco per testare ogni contenuto candidato, utilizzando molte risorse. Diversamente, nella funzione di fitness diretta, vengono svolti pochi calcoli matematici per ogni contenuto candidato.

*Funzione di fitness* ***interattiva***: La funzione di fitness interattiva basa la sua valutazione sull'interazione di giocatori reali nel videogioco. L'interazione dei giocatori con il videogioco e, nello specifico, con il contenuto candidato genera dati utili alla funzione di fitness. Questi dati possono essere raccolti in due modi:

- *esplicitamente*: la raccolta dei dati avviene tramite questionari o tramite segni verbali da parte del giocatore
- *implicitamente*: la raccolta dei dati avviene tenendo traccia di alcuni eventi come, ad esempio, quante volte un giocatore sceglie un particolare contenuto candidato o l'uscita dal gioco da parte del giocatore.

Scegliere il tipo di raccolta dati da utilizzare non è semplice. I dati raccolti in modo esplicito spesso causano l'interruzione della sessione di gioco (a meno che la raccolta dati sia ben integrata nella fase di progettazione del gioco). I dati raccolti in modo implicito spesso portano ad avere dati rumorosi, incompleti ed inaccurati.

### 2.2.2 Generazione procedurale di mappe con algoritmi di ricerca

Nei videogiochi possiamo definire il termine *mappa* come un piano bidimensionale che possiede diverse caratteristiche (muri, fiumi, alberi, ecc.) le quali possono influenzare o meno il gameplay (lo svolgimento del gioco). È inoltre possibile che ad essa sia associata un'altra mappa, chiamata *heightmap*, che ne specifichi l'altezza di ogni punto.

Le mappe sono una componente fondamentale in molti tipi di videogiochi, inclusi gli sparatutto in prima persona (FPS o First Person Shooter) e i giochi di strategia in tempo reale (RTS o Real-Time Strategy). Mentre nel primo caso i giocatori esplorano un mondo ostile da una prospettiva in prima persona, nel secondo coordinano unità (come soldati, carri armati, ecc.) da una prospettiva in terza persona (dall'alto).

Indipendentemente dal tipo di gioco, esistono molte ragioni che motivano la scelta di utilizzare la generazione procedurale di mappe. La ragione più ovvia è avere una mappa nuova ad ogni partita. Questo permette di aumentare la longevità del gioco e di annullare il vantaggio dovuto alla conoscenza della mappa nelle partite multigiocatore (partite con più giocatori). Un'altra ragione è l'utilizzo della generazione procedurale di mappe come strumento di supporto allo sviluppo. Utilizzando algoritmi di generazione procedurale di mappe durante le fasi di sviluppo del videogioco, è possibile ridurre il tempo e il costo necessari al

suo sviluppo. Uriarte e Ontañón hanno recentemente implementato PSMAGE [7], uno strumento di sviluppo utile a generare mappe bilanciate per il gioco di strategia *Starcraft*[16].

Sebbene siano stati compiuti molti studi riguardo la generazione di terreni, pochi di essi hanno affrontato il problema di generare una mappa adatta alle meccaniche di un particolare genere di gioco.

Sorenson e Pasquier [14] hanno utilizzato la generazione procedurale di contenuti con algoritmi evolutivi per evolvere dei labirinti di un videogioco. La rappresentazione della mappa consisteva in una superficie bidimensionale nella quale venivano posizionati corridoi e stanze di dimensioni differenti. Il valore di fitness era ottenuto semplicemente calcolando la distanza dall'ingresso all'uscita del labirinto.

In modo simile, Ashlock et al. [15] hanno utilizzato la generazione procedurale di contenuti con algoritmi di ricerca per risolvere un problema di path-planning (problema che consiste nel trovare la strada migliore per andare da una posizione A ad una posizione B). Un algoritmo generava il labirinto ruotando e posizionando i muri. L'obiettivo (della funzione di fitness) era massimizzare la distanza dall'inizio alla fine del labirinto.

Togelius et al. [2] utilizzano la generazione procedurale di contenuti con algoritmi di ricerca per generare le mappe migliori per un generico videogioco di strategia. L'algoritmo di ricerca usato è un algoritmo evolutivo multiobiettivo (*MOEA*) e, per quanto riguarda la rappresentazione, il genotipo codifica le coordinate delle basi dei giocatori, delle risorse e delle montagne (queste ultime si basano sulla deviazione standard di una distribuzione gaussiana). In [2] vengono definite sia funzioni di fitness dirette che funzioni di fitness basate sulla simulazione, queste ultime calcolano la distanza pesata tra le basi tramite l'algoritmo $A*$[17]. Ogni volta che l'algoritmo viene eseguito, le funzioni vengono ottimizzate a gruppi di due.

In [8], viene approfondito il lavoro svolto precedentemente in [2] applicandolo ad un videogioco commerciale: StarCraft. A questo scopo viene utilizzato un algoritmo MOEA, una rappresentazione basata su quella applicata nel lavoro precedente e funzioni di fitness adatte al videogioco in questione.

Recentemente [9] sono state estese le due precedenti ricerche [2, 8] ottimizzando (tramite l'algoritmo di ricerca MOEA) le funzioni di fitness a gruppi di tre e facendo testare le mappe generate a giocatori reali, raccogliendo i giudizi ottenuti.

[16] http://it.wikipedia.org/wiki/StarCraft
[17] http://en.wikipedia.org/wiki/A*_search_algorithm

Ashlock et al. in [4] confrontano quattro tipi diversi di rappresentazione di labirinti nei videogiochi. Una prima rappresentazione nella quale una griglia $X \times Y$ viene codificata in un vettore di $XY$ bit. Lo stato di ogni bit indica se la posizione corrente della griglia è calpestabile o meno. Una seconda rappresentazione basata sui colori dell'arcobaleno, nella quale ogni cella della griglia è codificata nel genoma da un colore. Due celle adiacenti sono utilizzabili se e solo se i loro colori sono adiacenti nell'arcobaleno o se hanno lo stesso colore. Una terza rappresentazione nella quale il genoma specifica i punti dove aggiungere muri in una mappa vuota. Un'ultima rappresentazione complementare alla precedente nella quale il genoma specifica i punti calpestabili in una mappa inagibile. Queste quattro rappresentazioni vengono analizzate ottimizzando diverse funzioni obiettivo e confrontandone i valori di fitness ottenuti.

In [10], viene utilizzata la generazione procedurale di contenuti basata sulla ricerca per creare mappe per *Dune 2*[18], un videogioco di strategia in tempo reale predecessore di Starcraft. Grazie alle meccaniche di gioco semplicistiche, Mahlmann et al. hanno potuto ottimizzare una funzione di fitness semplice ottenendo comunque buoni risultati. La mappa è rappresentata nel genoma da un vettore di numeri reali, il genoma viene successivamente mappato nel fenotipo tramite un algoritmo di tipo costruttivo. La rappresentazione del fenotipo consiste in una matrice composta da $n \times n$ celle, ognuna di esse dotata di un valore (da 0 a 2) che indica il tipo di terreno (sabbia, roccia o risorsa) presente alla coordinata della mappa identificata dalla posizione della cella.

Lara-Cabrera et al. in [11] utilizzano la generazione procedurale di contenuti con algoritmi evolutivi per creare mappe bilanciate per *Planet Wars*[19], un videogioco di strategia in tempo reale ambientato nello spazio. La mappa è codificata in un genoma che ha questa struttura: i primi 4 parametri sono le coordinate cartesiane del pianeta base di 2 giocatori, seguono 20 gruppi da 4 parametri. Questi 4 parametri rappresentano le coordinate cartesiane (primo e secondo parametro) di un pianeta neutrale, la sua produzione di navi per turno (terzo parametro) e il suo numero di navi iniziali (quarto parametro). La funzione di fitness è basata sulla simulazione e mira a minimizzare il divario di punti tra i due giocatori.

Ad oggi l'unica applicazione di generazione procedurale di contenuti con algoritmi di ricerca per la creazione di mappe in un videogioco FPS è stata fatta in [12] per il videogioco *Cube 2*[20]. In [12], gli autori affrontano il problema di generare mappe per un videogioco FPS multigiocatore, analizzando quattro

[18] http://it.wikipedia.org/wiki/Dune_II
[19] http://planetwars.aichallenge.org
[20] http://sauerbraten.org

tipi diversi di rappresentazioni. La funzione di fitness cerca di massimizzare il *time to fight* (o TTF), definito come la quantità di tempo trascorso da quando il giocatore comincia a combattere a quando viene ucciso.

## 2.3 Bilanciamento della difficoltà nei videogiochi

Bilanciare un videogioco[21] consiste nell'assicurarsi che il gioco fornisca al giocatore un'esperienza coerente con quella pensata dagli sviluppatori in fase di progettazione [19]. Supponiamo che in un videogioco esista una componente che sbilanci le meccaniche di gioco, in tal caso i giocatori sfrutteranno la componente sbilanciata per vincere, a discapito delle altre componenti (alternative alla prima) messe a disposizione nel gioco (che risulteranno più deboli rispetto alla componente che provoca lo sbilanciamento). Questa situazione porta ad uno spreco di tempo e risorse da parte della casa produttrice del videogioco, in quanto essa avrà investito tempo (e risorse) sia in fase di progettazione che in fase di sviluppo per realizzare le componenti che rimangono inutilizzate.

È necessario distinguere tra il bilanciamento in un gioco a giocatore singolo da quello a più giocatori [19]. Il bilanciamento di un videogioco a giocatore singolo consiste nell'assicurarsi che il livello di difficoltà del gioco sia adeguato al pubblico a cui è rivolto. Il bilanciamento di un videogioco a più giocatori consiste invece nel verificare che nessun giocatore parta avvantaggiato rispetto agli altri e che nessuna strategia di gioco domini le altre.

### 2.3.1 Bilanciamento di videogiochi a più giocatori

Al fine di comprendere il contesto entro il quale il lavoro di tesi si inserisce, ci focalizziamo ora sul bilanciamento di videogiochi a più giocatori. Per farlo risulta necessario introdurre due concetti chiave utili all'analisi del bilanciamento: *overpowered* e *underpowered*. Il termine overpowered (o *sovrapotenziato*) indica un elemento di gioco utilizzabile dal giocatore (come, ad esempio, un oggetto, un'abilità o un'arma) che risulta essere la scelta migliore in un numero molto elevato di situazioni o che è eccessivamente difficile da contrastare. Questo porta i giocatori ad utilizzare sempre l'elemento di gioco overpowered a discapito degli altri. Contrariamente, underpowered (o *sottopotenziato*) indica un elemento di gioco che, se utilizzato, pone il giocatore in una posizione di svantaggio rispetto a chi non lo usa. Questo porta i giocatori a non utilizzare mai un elemento di gioco underpowered. La presenza di elementi overpowered o underpowered

[21] http://en.wikipedia.org/wiki/Balance_(game_design)

porta quindi all'inutilizzo di alcuni elementi di gioco, causando uno spreco di risorse utilizzate in fase di sviluppo.

Se in un gioco competitivo multigiocatore (gioco a cui partecipano più giocatori) esistono degli elementi overpowered essi modificheranno le meccaniche del gioco, sbilanciandolo. Infatti, una volta che i giocatori scopriranno che un elemento di gioco prevale sugli altri, lo utilizzeranno sempre. Questo obbligherà gli altri giocatori ad utilizzare l'elemento overpowered in modo da competere alla pari con chi ne fa già uso. Nonostante questo, un videogioco sbilanciato potrebbe comunque essere equo tra tutti i giocatori. Non necessariamente, infatti, un elemento di gioco overpowered (o underpowered) influenza la probabilità di vittoria dei giocatori. Prendiamo come esempio il gioco della *morra cinese*[22] e supponiamo di aggiungere un nuovo segno che vinca contro tutti gli altri tre. In questo modo abbiamo introdotto un elemento overpowered sbilanciando il gioco. Nonostante questo, tutti i giocatori hanno le stesse probabilità di vittoria, in quanto ognuno di loro potrà scegliere di giocare il segno appena introdotto. Lo stesso ragionamento si applica anche, in modo inverso, agli elementi underpowered.

Quando nel gioco è presente un elemento overpowered emerge una *strategia dominante*, ovvero una strategia che domina tutte le altre possibili strategie. In qualsiasi situazione l'approccio di gioco adottato dalla strategia dominante è migliore rispetto a quello adottato da qualsiasi altra strategia che il giocatore può scegliere. Questo provoca una diminuzione dell'insieme delle azioni che un giocatore può svolgere all'interno del gioco, in quanto nessun giocatore sceglierà di utilizzare una strategia debole quando è a conoscenza dell'esistenza di una strategia che la domina. *Tic-tac-toe*[23] è un esempio di gioco da tavola che ha una strategia dominante [19]. Quando solo uno dei due giocatori la conosce, quest'ultimo vince sempre. Nel momento in cui entrambi i giocatori ne vengono a conoscenza (e la usano), la partita termina sempre in parità. Per questo motivo una volta che entrambi i giocatori conoscono la strategia dominante di tic-tac-toe il gioco diventa noioso.

Per facilitare il bilanciamento di un videogioco bisognerebbe evitare a priori (in fase di sviluppo) situazioni che potrebbero portare ad avere una strategia dominante oppure predisporre il gioco per averne più di una. In questo modo il giocatore potrà scegliere la strategia che preferisce senza essere obbligato ad usare quella dominante per vincere.

[22] http://it.wikipedia.org/wiki/Morra_cinese
[23] http://it.wikipedia.org/wiki/Tris_(gioco)

### 2.3.2 Bilanciamento delle mappe nei videogiochi

Per rendere il videogioco bilanciato, ogni elemento o componente del gioco deve essere progettata con cura. Riferendoci ai videogiochi multigiocatore sparatutto in prima persona e a quelli di strategia in tempo reale, uno degli elementi più importanti da bilanciare è la mappa in cui il gioco si svolge.

Per questo motivo vengono investite molte risorse per riuscire a creare delle mappe bilanciate, facendole testare ai giocatori durante le fasi di sviluppo del gioco e studiandone le caratteristiche. Nel primo caso, dai test effettuati sui giocatori è possibile ricavare informazioni interrogando gli stessi o raccogliendo informazioni implicitamente nel gioco (come, ad esempio, i punti della mappa più visitati dai giocatori). Nel secondo caso invece vengono valutate le caratteristiche della mappa (come forma, posizione delle risorse o delle armi e altezza di alcune zone) basandosi su alcuni studi. Gli studi effettuati sulle caratteristiche che rendono buona una mappa multigiocatore sono molti [22, 21, 23, 24], alcuni dei quali sono rivolti al bilanciamento della stessa [20]. Rendere la mappa simmetrica non è sempre una soluzione al problema del bilanciamento. È infatti possibile che, pur essendo simmetrica, la mappa dia luogo ad una strategia dominante [20]. In un videogioco sparatutto in prima persona, per esempio, una mappa simmetrica potrebbe comunque avere una struttura che avvantaggi l'uso di una particolare arma rispetto alle altre.

Nell'ambito accademico alcuni ricercatori hanno affrontato il problema di bilanciare le mappe nei videogiochi, parte di essi applicando la generazione procedurale di contenuti con algoritmi di ricerca. Durante la ricerca sulla generazione procedurale di mappe per Starcraft, Yannakakis et al. [8, 9] utilizzano alcune funzioni di fitness utili a valutare quanto le mappe risultino essere bilanciate. Queste funzioni si basano sulla valutazione delle distanze tra le basi (dei giocatori) e le risorse. In [11], Lara-Cabrera et al. applicano la generazione procedurale di contenuti con algoritmi evolutivi per creare mappe bilanciate in Planet Wars. Due bot (che utilizzano la stessa IA) vengono fatti giocare l'uno contro l'altro nella mappa appena generata estraendo, alla fine della partita, il numero di astronavi rimaste e i punti ottenuti dai due bot durante la partita. Questi quattro valori (due per ogni bot) vengono utilizzati nel calcolo della funzione di fitness, funzione che mira a massimizzare la similarità tra i due valori ricavati dal primo bot e quelli ricavati dal secondo. In [25], Reddad e Verbrugge hanno studiato le strutture delle mappe nei videogiochi di strategia estraendo delle euristiche utili a classificare se una mappa è bilanciata o meno. Queste euristiche sono espresse e formalizzate come un insieme di proprietà geometriche calcolate sulla mappa. A seguito di questa ricerca è stato sviluppato anche

uno strumento (basato su queste euristiche) per la valutazione del bilanciamento delle mappe in *Starcraft 2*[24]. Anche l'algoritmo PSMAGE [7], sviluppato da Uriarte e Ontañón, utilizza la generazione procedurale per generare mappe bilanciate utilizzabili nei tornei. L'algoritmo genera le mappe per il videogioco Starcraft basandosi sul concetto di simmetria. Dopo la generazione, viene valutato il bilanciamento della mappa generata tramite versioni modificate delle funzioni usate in [8].

Tutte le ricerche fino ad ora citate affrontano il problema di bilanciare le mappe in un videogioco di strategia in tempo reale, ma nessuno ha provato ad affrontare il problema all'interno di un videogioco sparatutto in prima persona. Il nostro scopo in questa tesi è colmare questo vuoto, chiedendoci se è possibile bilanciare (in un videogioco sparatutto in prima persona) la situazione di svantaggio di un giocatore rispetto ad un altro agendo sulla mappa di gioco.

## 2.4 Sommario

In questo capitolo abbiamo trattato lo stato dell'arte. Nella Sezione 2.1 abbiamo introdotto la generazione procedurale di contenuti nei videogiochi, fornendone una breve storia. Nella Sezione 2.2 abbiamo descritto la generazione procedurale di contenuti con algoritmi di ricerca. Nella sottosezione 2.2.1 abbiamo preso in analisi i vari tipi di funzioni di fitness utilizzabili. Nella sottosezione 2.2.2 ci siamo focalizzati sulla generazione procedurale di mappe con algoritmi di ricerca e abbiamo presentato alcune ricerche accademiche che la riguardano. Nella Sezione 2.3 abbiamo spostato la nostra attenzione sul concetto di bilanciamento della difficoltà nei videogiochi. Successivamente, nella sottosezione 2.3.1, abbiamo specificato come essa influenzi i videogiochi nelle modalità a più giocatori, introducendo il concetto di strategia dominante. Infine, nella sottosezione 2.3.2, abbiamo trattato il bilanciamento delle mappe nei videogiochi fornendone alcuni esempi di studi accademici.

[24] http://eu.battle.net/sc2/it/

# Capitolo 3

# Evoluzione di mappe bilanciate per FPS

In questo capitolo descriviamo i nostri obiettivi e le componenti sulle quali il nostro lavoro si basa. Nella prima parte illustriamo gli obiettivi di questo lavoro, nella parte successiva descriviamo Cube 2, il videogioco utilizzato per i nostri esperimenti, e nella parte finale descriviamo le quattro diverse rappresentazioni che abbiamo utilizzato.

## 3.1 I nostri obiettivi

In questa tesi ci siamo chiesti se è possibile generare mappe per un videogioco sparatutto in prima persona che, data una situazione di svantaggio di un giocatore rispetto ad un altro, rendano bilanciato il combattimento tra due giocatori. Per rispondere a questa domanda abbiamo esteso il lavoro precedentemente svolto da Cardamone et al. in [12]. In [12], gli autori hanno applicato la generazione procedurale di contenuti con algoritmi genetici per generare mappe per il videogioco sparatutto in prima persona *Cube 2*. La funzione obiettivo dell'algoritmo genetico mirava a massimizzare il *Time To Fight* (TTF), ovvero la quantità di tempo trascorso da quando un giocatore comincia a combattere a quando viene ucciso. L'algoritmo genetico elementare utilizzato in [12] usava il *crossover a un punto*, la *mutazione semplice* e, come tecnica di selezione, il *tournament selection a 2 candidati*. La rappresentazione del genoma variava a seconda della rappresentazione della mappa utilizzata (*All-Black*, *All-White*, *Grind* o *Random-Digger*), mentre il fenotipo consisteva in un file di testo contenente una griglia bidimensionale di caratteri che codificavano le posizioni dei muri e delle risorse nella mappa. Prima di simulare la partita, un algoritmo interno al gioco creava la mappa utilizzabile nel gioco posizionando gli oggetti

e i muri nei punti specificati dal fenotipo. Per ricavare i dati utili al calcolo del TTF, veniva simulata una partita in modalità *deathmatch* tutti contro tutti (modalità nella quale lo scopo dei giocatori è effettuare il maggior numero di uccisioni rispetto agli avversari) all'interno della mappa candidata, facendo combattere quattro giocatori controllati dal calcolatore con abilità (efficienza in combattimento) uguale.

Partendo da questa base, il nostro obiettivo è stato quello di generare mappe che massimizzassero il bilanciamento dei combattimenti tra due giocatori. Per farlo, in questo lavoro simuliamo una partita in modalità *deathmatch* con due giocatori controllati dal calcolatore (al posto dei quattro usati in [12]) per ogni mappa candidata. La nostra funzione obiettivo, diversamente da quanto avveniva in [12], non è più il TTF, ma un indice di quanto le uccisioni tra i due giocatori siano bilanciate, assumendo che il bilanciamento perfetto di una partita sia raggiunto quando il $NumeroUccisioni_{Giocatore1}$ è uguale al $NumeroUccisioni_{Giocatore2}$. Per la realizzazione del nostro obiettivo è stato necessario utilizzare diverse configurazioni di partenza per creare situazioni nelle quali un giocatore risultasse svantaggiato rispetto all'altro. Tali configurazioni consistevano nell'impostare livelli di abilità diversi ai due giocatori, dotarli di arma diversa o imporre loro sia abilità che armi diverse.

## 3.2 Cube 2

*Cube 2: Sauerbraten*[1] è un videogioco open source di tipo sparatutto in prima persona. In questo tipo di videogioco, i giocatori esplorano un mondo ostile da una prospettiva in prima persona e affrontano i nemici che incontrano utilizzando un numeroso arsenale di armi da fuoco e non (giochi quali *Unreal tournament*[2] e *Quake III*[3] appartengono a questa tipologia di gioco) . Cube 2 si basa sul motore di gioco precedente *Cube*[4], un motore veloce e sofisticato che permette un rendering omogeneo di ambienti di gioco con molti poligoni. Il motore di gioco è basato su una struttura ad albero *Octree*[5] per rappresentare l'ambiente e supporta l'uso di illuminazione basta su *lightmap* con gestione delle ombre, schaders dinamici, sistemi particellari ed esplosioni. Come in ogni sparatutto in prima persona, al giocatore viene data la possibilità di utilizzare armi di diverso tipo: una motosega per lo scontro ravvicinato; un fucile (*rifle*) che uccide in un colpo solo con ratio di fuoco ridotto rispetto alle altre armi;

[1] http://sauerbraten.org
[2] http://it.wikipedia.org/wiki/Unreal_Tournament
[3] http://it.wikipedia.org/wiki/Quake_III_Arena
[4] http://cubeengine.com
[5] http://en.wikipedia.org/wiki/Octree

un mitragliatore con ratio di fuoco elevato; un lanciagranate che spara proiettili che rimbalzano sul terreno più volte prima di provocare un'esplosione ad area; un bazooka (lanciarazzi) che spara razzi che provocano un'esplosione ad area al momento del contatto con il terreno ed una pistola che causa danni ridotti rispetto alle altre armi. Cube 2 possiede numerose modalità di gioco che possono essere giocate sia da un singolo giocatore che, tramite LAN o internet, da più giocatori. Nel caso in cui vengano giocate in solitaria, i giocatori mancanti verranno sostituiti da *bot*[6], personaggi controllati dal computer. Questi ultimi sono comunemente usati negli sparatutto in prima persona per permettere al giocatore di fare pratica in assenza di avversari umani. Il numero delle modalità di gioco di Cube 2 è elevato. In *deathmatch* lo scopo dei giocatori è effettuare il maggior numero di uccisioni rispetto agli avversari (*bot* o giocatori). In *capture* l'obiettivo è conquistare alcuni punti della mappa e mantenerne il controllo il più a lungo possibile. In *capture the flag* per vincere bisogna conquistare la bandiera avversaria cercando di non farsi rubare la propria. In *collect* per ogni avversario ucciso viene generato un teschio, lo scopo dei giocatori è raccogliere più teschi possibili. Per ognuna delle modalità appena descritte esiste inoltre la variante *instagib*[7], introdotta per la prima volta in *Quake II*[8] e presente in molti sparatutto in prima persona. In questa variante ad ogni giocatore viene data in dotazione un'unica arma con munizioni infinite che uccide con un solo colpo. Infine Cube 2 dispone di una modalità (giocabile anche in cooperativa) che permette di apportare modifiche alle mappe direttamente nel gioco, funzionalità resa possibile grazie alla flessibilità del motore del gioco. L'editor che Cube 2 mette a disposizione dei giocatori per creare le mappe è molto potente. Permette infatti, per esempio, di modificare la forma del terreno, aggiungere oggetti (anche personalizzati) di vari materiali (come, per esempio, lava o acqua), applicare texture e aggiungere effetti particellari. Inoltre, il gioco mette a disposizione un linguaggio di *scripting*[9] che permette di personalizzare facilmente i menu e le altre componenti di gioco.

Essendo open source, è inoltre possibile modificare il codice sorgente del gioco in modo da aggiungere funzionalità o ricavare dati. Dovendo testare automaticamente le mappe generate in modo da ricavarne dati utili al calcolo del valore di fitness, Cardamone et al. in [12] scelgono di utilizzare Cube 2 per la loro ricerca sulla generazione procedurale di mappe nei videogiochi. Un altro motivo per cui in [12] si sceglie di utilizzare Cube 2 è la possibilità di avviare il videogioco senza visualizzazione grafica. Utilizzando questa funzionalità è pos-

[6] http://it.wikipedia.org/wiki/Bot
[7] http://it.wikipedia.org/wiki/Instagib
[8] http://it.wikipedia.org/wiki/Quake_II
[9] http://it.wikipedia.org/wiki/Linguaggio_di_scripting

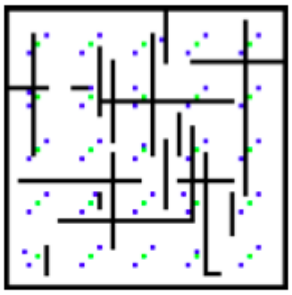


Figura 3.1: Mappa generata con rappresentazione *All-White*

Figura 3.2: Mappa generata con rappresentazione *All-Black*

sibile ridurre notevolmente i tempi necessari alla simulazione del combattimento tra *bot*. Di conseguenza anche il tempo necessario al calcolo della funzione di fitness basata sulla simulazione si riduce. Per quanto ci riguarda, i motivi per cui abbiamo scelto di utilizzare Cube 2 per i nostri scopi sono: estrema flessibilità data da un videogioco open source, riduzione del tempo di calcolo della fitness basata sulla simulazione grazie alla possibilità di escludere la visualizzazione grafica e la disponibilità dei *bot* nel gioco per la simulazione. Inoltre, decidendo di basarci sul lavoro svolto in [12], l'uso di Cube 2 è stata una scelta obbligata.

## 3.3 Le rappresentazioni delle mappe per l'evoluzione

Cardamone et al., in [12], hanno implementato ed utilizzato 4 diverse rappresentazioni di mappe, ovvero la *All-White*, la *All-Black*, la *Grind* e la *Random-Digger*. Nella rappresentazione *All-White*, una mappa è inizialmente vuota (ovvero ha solo i muri esterni che la delimitano) e vengono successivamente inseriti al suo interno i muri. Il genoma, di dimensioni $n_g$, codifica la posizione di $\frac{n_g}{3}$ muri, ognuno dei quali è rappresentato da una tripla $< x, y, l >$. $x$ e $y$ indicano le coordinate della mappa in cui verrà inserito il muro, riferite all'angolo in alto a sinistra di quest'ultimo e $l$ indica la lunghezza che esso avrà. Inoltre, se $l$ assume valore positivo allora il muro sarà posizionato orizzontalmente, in caso contrario sarà posizionato verticalmente. La figura 3.1 mostra un esempio di mappa generata con questo tipo di rappresentazione.

La rappresentazione *All-Black* è una rappresentazione complementare alla *All-White*. La mappa è inizialmente piena di muri e ne viene gradualmente liberato lo spazio aggiungendo aree libere. Il genoma codifica 2 tipi di elementi, le stanze e i corridoi. Le prime sono dei quadrati di spazio libero definite da una tripletta $< x, y, s >$, dove $x$ e $y$ indicano le coordinate della mappa in cui verrà inserita la stanza, riferite al centro di quest'ultima e $s$ indica la dimensione del

lato. I corridoi invece sono spazi liberi rettangolari con un lato di dimensione prefissata e sono anch'essi definiti da una tripletta $< x, y, l >$, valori che indicano le stesse caratteristiche della tripletta nella rappresentazione *All-White*. Con $a$ numero stanze e $b$ numero corridoi, la dimensione del genoma è pari a $n_g = a \times 3 + b \times 3$. La figura 3.2 mostra un esempio di mappa generata con questo tipo di rappresentazione.

Nella rappresentazione *Grind* il genoma codifica gli elementi della mappa direttamente. Questa rappresentazione considera la mappa come una griglia di muri $9 \times 9$, dividendola quindi in 81 quadrati. Ogni lato di questo quadrato può essere rimosso in modo da formare corridoi e stanze. Il genoma codifica quali muri del quadrato in posizione $x = i \ mod \ 9$ e $y = i/9$ (divisione intera) sono attivi e quali no, identificando con $i$ l'$i$-esimo allele del genoma. Ogni allele può assumere 4 valori: 0 se il muro in alto al quadrato è attivo, 1 se quello a sinistra è attivo, 2 se sono entrambi attivi, 3 se nessuno dei due è attivo. La lunghezza del genoma è fissata a $n_g = 81$. La figura 3.3 mostra un esempio di mappa generata con questo tipo di rappresentazione.

Nella rappresentazione *Random-Digger* la mappa è completamente piena di muri, come nel caso della rappresentazione *All-Black*. Nella rappresentazione *Random-Digger*, però, il genoma codifica le probabilità con cui un agente scavatore effettua determinate mosse. Partendo dal centro della mappa, l'agente si muove per la mappa calcolando la direzione da intraprendere ad ogni passo tramite le probabilità definite nel genoma. Ogni spazio della mappa attraversato dall'agente diventa uno spazio libero. Il genoma è composto da solo $n_g = 4$ allele che rappresentano 4 probabilità. Il primo valore è la probabilità con cui l'agente continui a proseguire dritto seguendo la direzione corrente, il secondo la probabilità di girare a destra, il terzo quella di girare a sinistra e l'ultimo è la probabilità di calpestare nuovamente uno spazio già visitato nel caso esso venga selezionato come passo successivo. La figura 3.4 mostra un esempio di mappa generata con questo tipo di rappresentazione.

Un'ultima osservazione va fatta sul posizionamento dei punti di rinascita (*spawn-point*) e su quello delle armi (o dei medipack). Nell'algoritmo utilizzato in [12], sia gli *spawn-point* che le armi vengono posizionati in maniera deterministica e indipendente dal genoma o dalla rappresentazione usata. Questi punti (sia di *spawn-point* che punti di posizionamento oggetti) vengono inseriti solo dopo aver generato la mappa dal genoma, ad una distanza $k$ fissata a priori. Per il nostro lavoro, la posizione in cui vengono collocati gli oggetti non è rilevante. Essi, all'inizio della partita, vengono infatti disattivati e le armi non possono essere raccolte, in modo da focalizzarci sul bilanciare la partita date le sole condizioni iniziali da noi imposte. Per questo motivo, nelle figure che studieremo

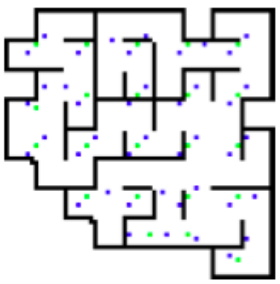


Figura 3.3: Mappa generata con rappresentazione *Grind*

Figura 3.4: Mappa generata con rappresentazione *Random-Digger*

nei capitoli successivi non sono stati rimossi i punti di posizionamento degli oggetti generati dall'algoritmo usato in [12] (i punti verdi). Per quanto riguarda il posizionamento degli *spawn-point* (i punti blu) utilizzeremo la stessa politica adottata in [12].

## 3.4 Sommario

In questo capitolo abbiamo definito l'obiettivo di questa tesi. Dopo aver descritto le basi su cui si fonda il nostro lavoro e le rappresentazioni con cui possiamo generare le mappe, estendiamo ora il lavoro svolto in [12] cercando il bilanciamento di partite a due giocatori tramite la generazione di mappe che lo favoriscano.

# Capitolo 4

# Analisi dei bot in Cube 2

In questo capitolo analizziamo i *bot* di Cube 2 illustrandone le caratteristiche principali. Successivamente analizziamo quanto il divario di livello di abilità tra *bot* influenzi il risultato finale della partita data l'arma usata (uguale per entrambi) da questi ultimi. Infine focalizziamo la nostra attenzione sull'analisi di un'arma in particolare: il *fucile* (o altrimenti detto *rifle*).

## 4.1 I bot di Cube 2

Per creare delle mappe che bilancino lo svantaggio di un giocatore rispetto ad un altro risulta necessario utilizzare una funzione di fitness basata sulla simulazione. Una funzione di fitness diretta non sarebbe stata utile in quanto la sola mappa non è sufficiente a fornirci dati utili al bilanciamento del gioco. Diversamente, una funzione di fitness interattiva sarebbe stata la soluzione migliore ma, non avendo a disposizione giocatori a cui far testare il gioco (ne abbastanza tempo), adottare questo tipo di funzione non è stato possibile. Per questo motivo, dovendo utilizzare una fitness basata sulla simulazione, è stato necessario testare i *bot* di Cube 2 verificando come essi si comportassero in diverse situazioni di gioco. Per la nostra analisi ci siamo focalizzati su partite in modalità *deathmatch* uno contro uno, utilizzando quindi 2 *bot*. Questa configurazione vale per tutti i risultati presentati nei capitoli di questa tesi.

In Cube 2, esiste una sola intelligenza artificiale che viene applicata a più istanze di *bot*. Per questo motivo in una stessa situazione due istanze diverse di *bot* si comportano allo stesso modo. L'intelligenza artificiale sceglie l'azione da compiere a seconda dello stato in cui si trova, alternando quattro stati tra loro a seconda della situazione di gioco. Gli stati sono: *Wait*, sceglie un punto della mappa da raggiungere per tenersi occupato, *Pursue*, una volta visto il nemico lo insegue sparandogli, *Defend*, difende l'obiettivo (presente solo nelle modalità

ad obiettivo, quale *capture the flag*), *Interest*, si muove verso il punto d'interesse (un oggetto di gioco come, ad esempio, un'arma o un punto scelto durante lo stato *Wait*).

Il gioco mette a disposizione la possibilità di personalizzare i *bot* di gioco specificandone l'abilità, parametro che rispecchia quanto il *bot* è preciso nel combattimento: più il valore di abilità impostato è elevato, maggiore è la precisione nella mira del *bot* e la sua l'abilità di individuare i nemici velocemente. Questo valore non influenza l'abilità di schivata del *bot* (cosa che avviene in altri videogiochi come, ad esempio, *Unreal tournament*) o la sua abilità nell'utilizzo di un determinato tipo di arma. È possibile impostare il valore di abilità del *bot* in un intervallo compreso tra 0 e 100, dove 0 è il l'abilità minima e 100 è quella massima. Avendo la stessa IA, questo parametro è l'unico modo in cui si possono diversificare i *bot*.

È necessario osservare come l'applicazione di una sola IA a tutti i *bot* di gioco porti al determinismo delle partite, dove per determinismo si intende che date le stesse condizioni iniziali la partita terminerà allo stesso modo. Il determinismo delle partite in Cube 2 è dovuto anche alla scelta sequenziale degli *spawn point*[1], punti sparsi nella mappa dove i *bot* (e i giocatori) vengono rigenerati dopo la morte. Al posto di essere rigenerati in un *spawn point* casuale, i *bot* vengono rigenerati nel punto successivo di spawn presente in una sequenza. La sequenza viene definita all'inizio della partita basandosi su un *seed* iniziale (ovvero un valore numerico), stesso *seed* implica stessa sequenza di punti in cui i *bot* verranno rigenerati. Per questo motivo, data la stessa mappa e le stesse condizioni iniziali, i *bot* si troveranno sempre nelle stesse situazioni (dovuto alla stessa sequenza di *spawn point*) e si comporteranno sempre allo stesso modo (dovuto alla stessa IA). Queste caratteristiche risultano utili alla nostra analisi, infatti il determinismo rende più stabili i nostri risultati e permette di replicarli date le stesse condizioni iniziali: abilità dei *bot*, mappa e seed degli *spawn point*.

## 4.2 Analisi delle differenze di livello tra bot

Ai fini di utilizzare correttamente i *bot* di gioco per simulare una situazione di svantaggio di uno rispetto all'altro si è reso necessario studiare come il valore di abilità impostata per ogni *bot* influenzi il risultato del gioco. Per farlo abbiamo simulato una partita per ogni combinazione di abilità tra due *bot*, senza considerare le permutazioni (0-0, 0-1, ... , 0-100, 1-1, 1-2, ... , 99-100 , 100-100), e per ogni arma (uguale per entrambi), raccogliendo il numero di uccisioni effettuate e quello delle morti subite. Ognuna di queste partite si svolgeva in una

[1] http://it.wikipedia.org/wiki/Spawn_(videogiochi)

mappa quadrata completamente aperta (senza muri, ad esclusione dei margini della mappa), nella quale non era presente nessun riparo.

Allo scopo di analizzare i dati raccolti risulta necessario introdurre due misure utili alla nostra analisi: il *Kill Ratio* e la differenza tra uccisioni (*Diff*). Il *Kill Ratio* è il rapporto tra le uccisioni effettuate dai due *bot*. In una partita completamente bilanciata il *Kill Ratio* è pari a 1. Formalmente, definendo le uccisioni effettuate con $K$ (Kills), la funzione è così descritta:

$$killRatio = \begin{cases} \frac{K_{Bot1}}{K_{Bot2}} & \text{if } K_{Bot1} > K_{Bot2} \\ -\frac{K_{Bot2}}{K_{Bot1}} & \text{if } K_{Bot1} < K_{Bot2} \\ 0 & \text{if } K_{Bot1} = K_{Bot2} = 0 \end{cases}$$

Questa misura trae spunto dal *Ratio K/D*, una misura molto usata nelle modalità a più giocatori dei videogiochi sparatutto in prima persona come indice di bravura del giocatore. Quest'ultima è data dal rapporto tra le uccisioni effettuate dal giocatore e le morti subite dallo stesso. Simulando partite con solo due *bot*, il *Kill Ratio* coincide con il *Ratio K/D* in quanto: $UccisioniEffettuate_{Bot2} = MortiSubite_{Bot1}$. Sostituendo nell'equazione troviamo che

$$KillRatio = \frac{UccisioniEffettuate_{Bot1}}{MortiSubite_{Bot1}} = RatioK/D_{Bot1}$$

Ciò non vale per armi che possono causare suicidi, quali *lanciagranate* e *bazooka*. La *Diff* è invece la differenza tra le uccisioni effettuate dai due *bot*. Quest'ultima ci è utile a capire l'entità della sproporzione tra i due valori di uccisioni.

$$Diff = |UccisioniEffettuate_{Bot1} - UccisioniEffettuate_{Bot2}|$$

Seguono ora le immagini che mostrano i risultati delle analisi per ogni arma presente nel gioco. Ogni immagine rappresenta una *heatmap*[2], rappresentazione grafica dei dati nella quale ogni punto è rappresentato da un colore che ne identifica il valore. Nelle *heatmap* che seguono, i valori sull'asse delle ascisse corrispondono al livello di abilità del $bot_2$ mentre quelli sull'asse delle ordinate corrispondono al livello di abilità del $bot_1$. Ogni punto identifica il risultato di una partita in termini di *Diff* o di *Kill Ratio*. La partita viene simulata da *bot* con livello di abilità specificato dalle coordinate del punto.

[2] http://en.wikipedia.org/wiki/Heat_map

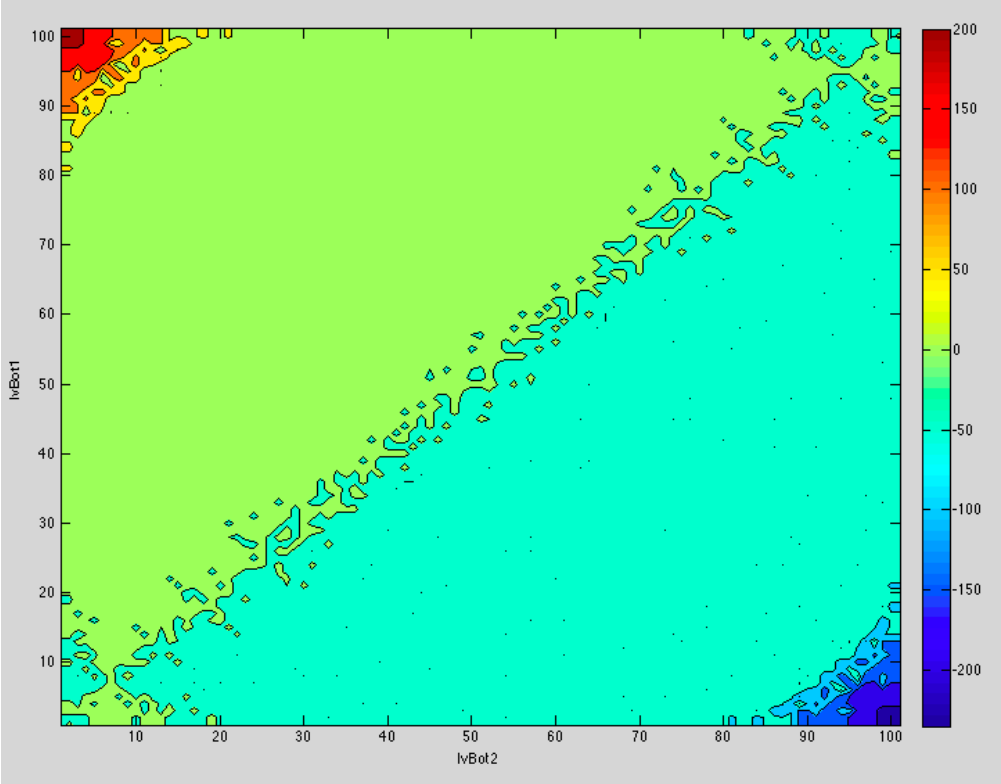



Figura 4.1: Heatmap del *Kill Ratio* per l'arma *Pistola*

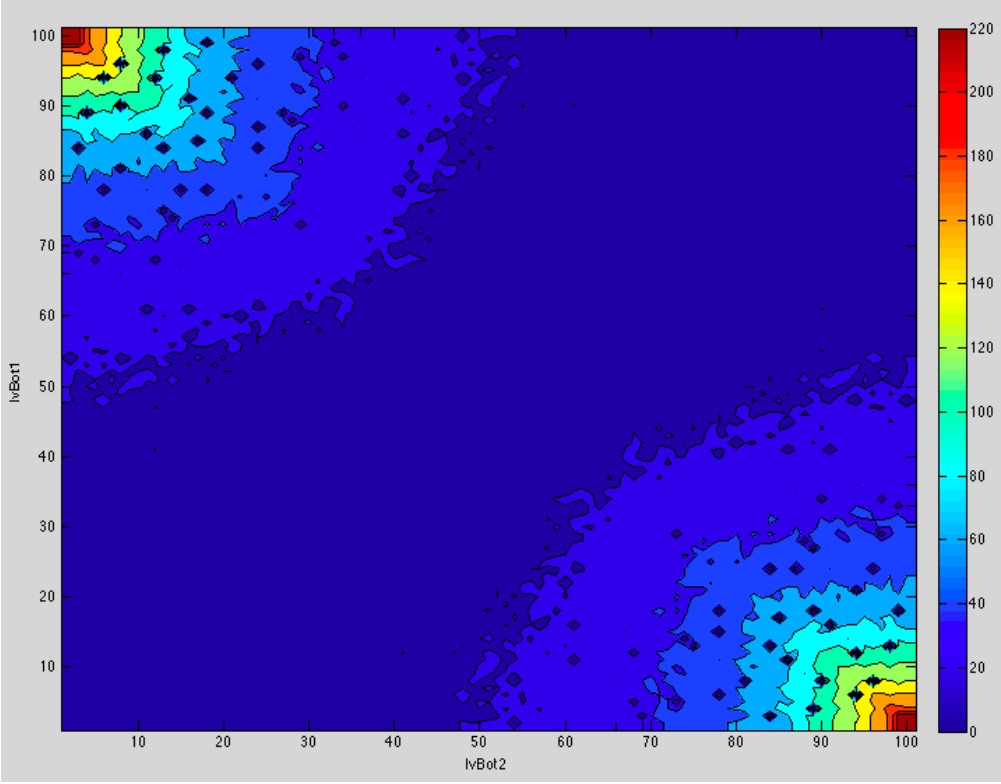



Figura 4.2: Heatmap della *Diff* per l'arma *Pistola*

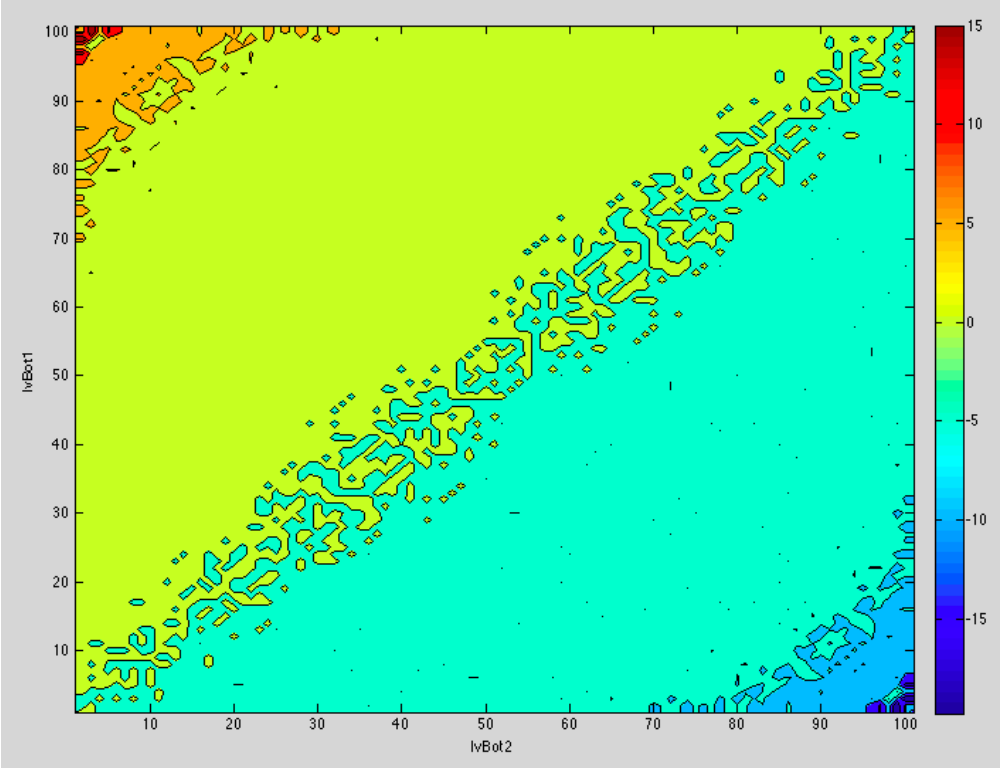



Figura 4.3: Heatmap del *Kill Ratio* per l'arma *Bazooka*

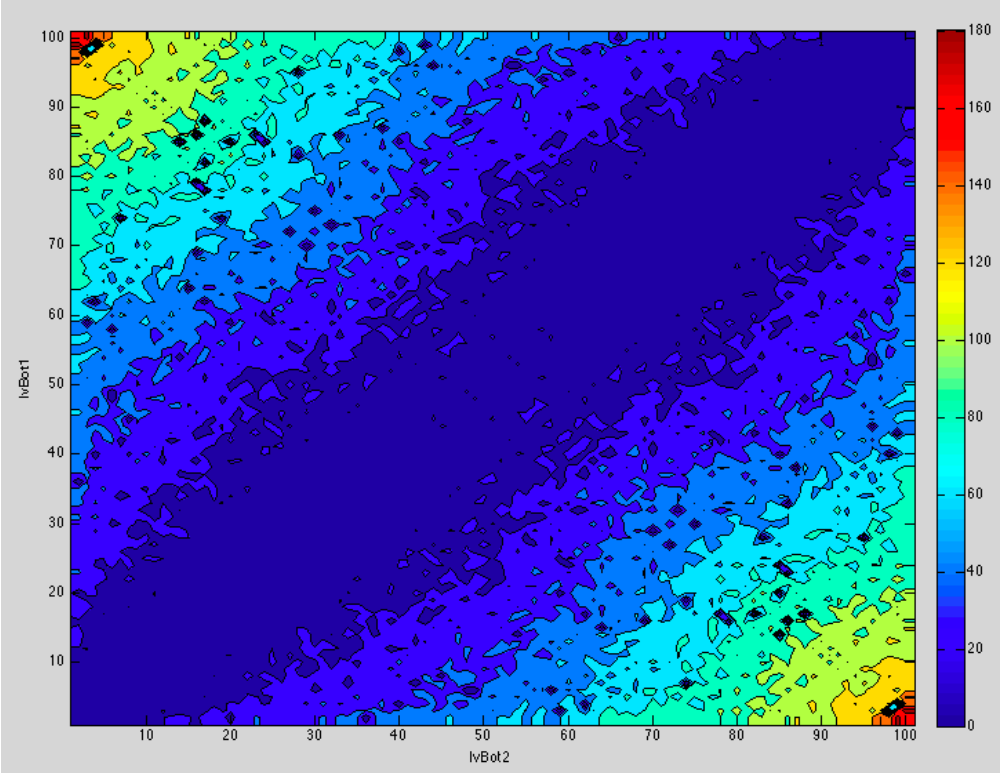



Figura 4.4: Heatmap della *Diff* per l'arma *Bazooka*

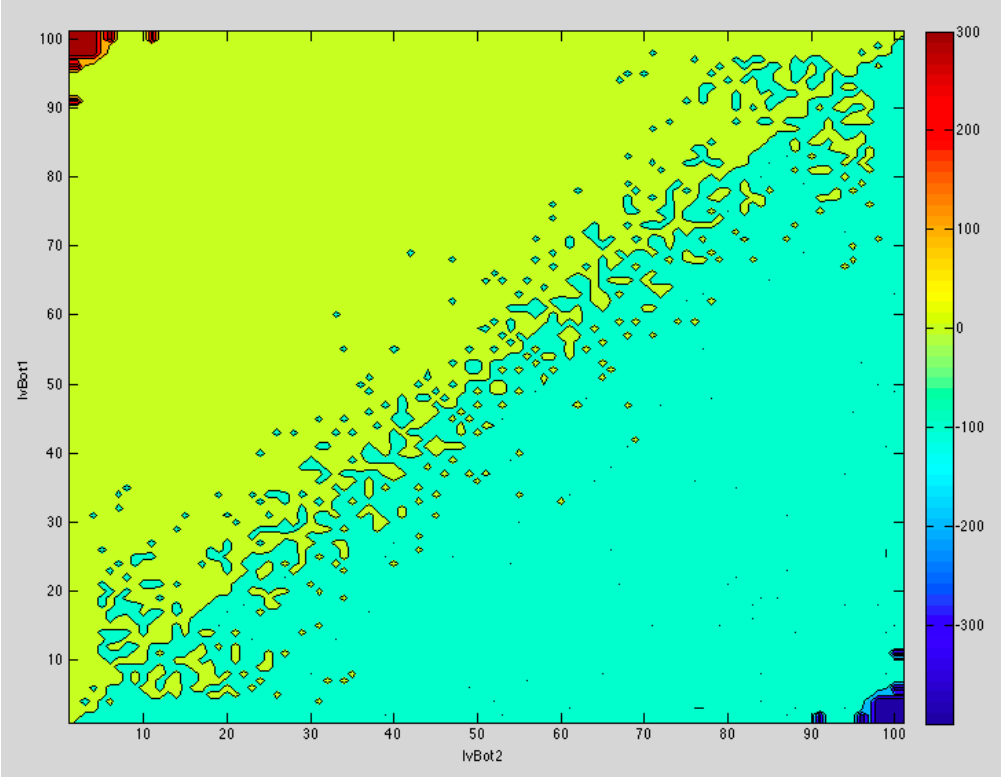



Figura 4.5: Heatmap del *Kill Ratio* per l'arma *Rifle*

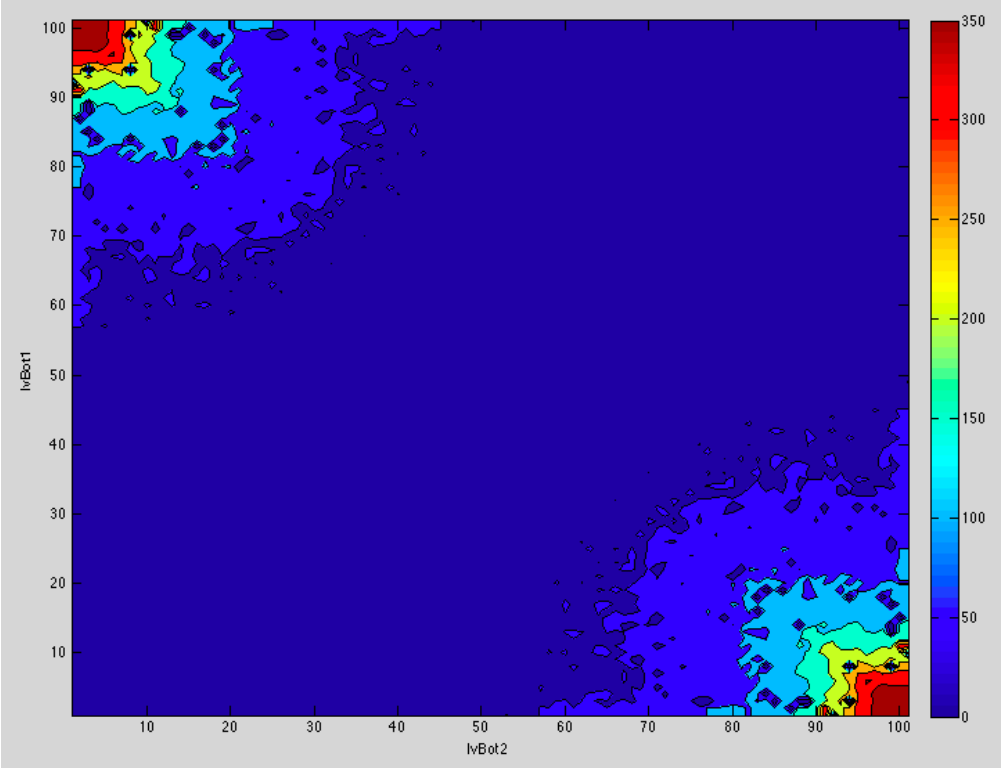



Figura 4.6: Heatmap della *Diff* per l'arma *Rifle*

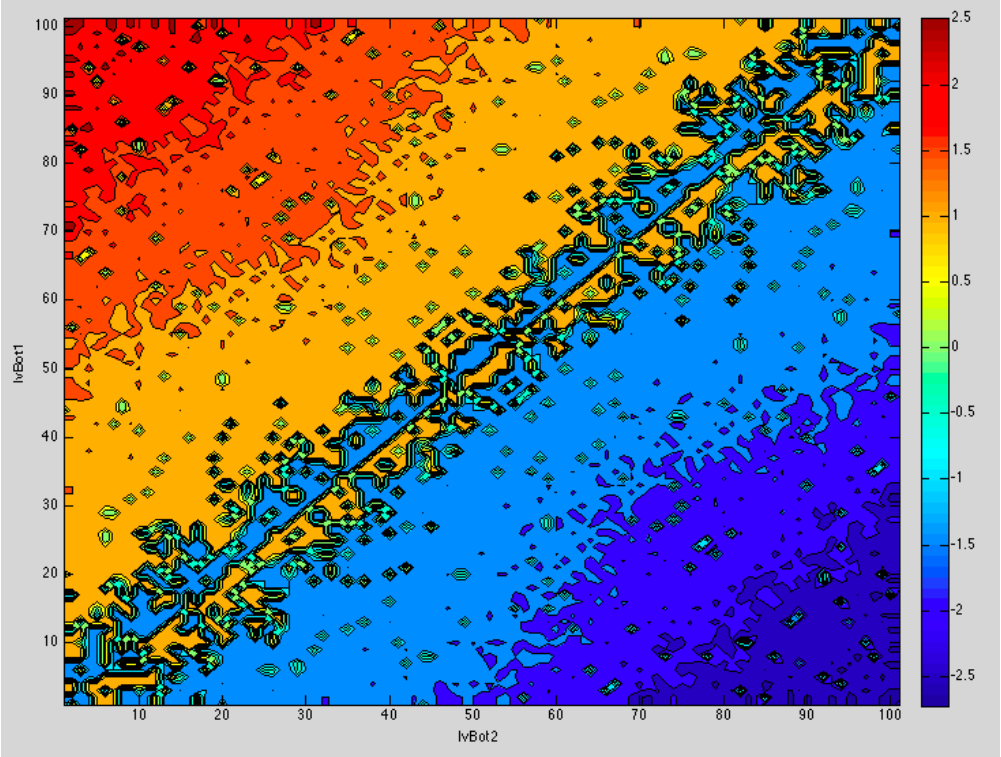



Figura 4.7: Heatmap del *Kill Ratio* per l'arma *Motosega*

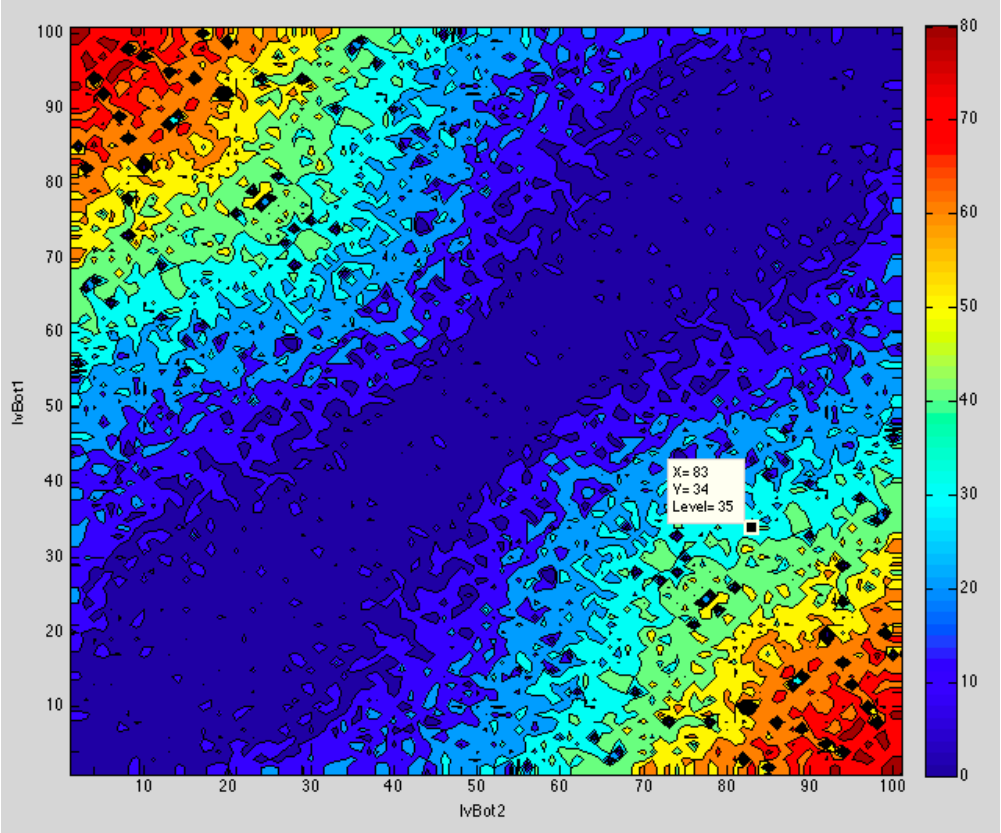



Figura 4.8: Heatmap della *Diff* per l'arma *Motosega*

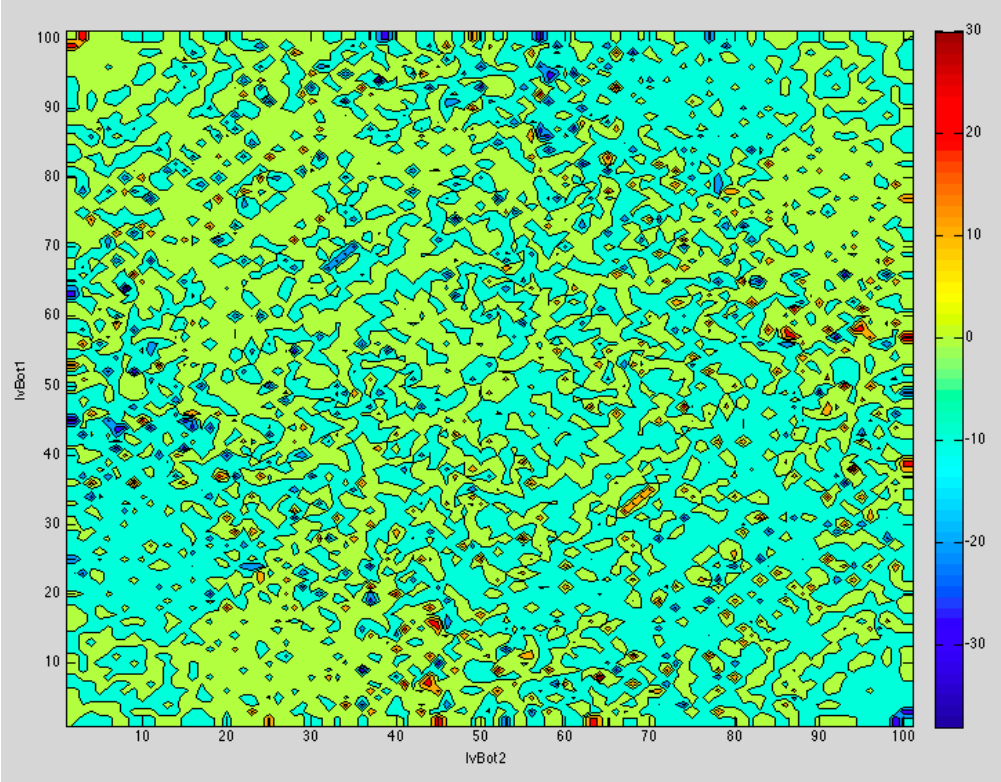



Figura 4.9: Heatmap del *Kill Ratio* per l'arma *Lanciagranate*

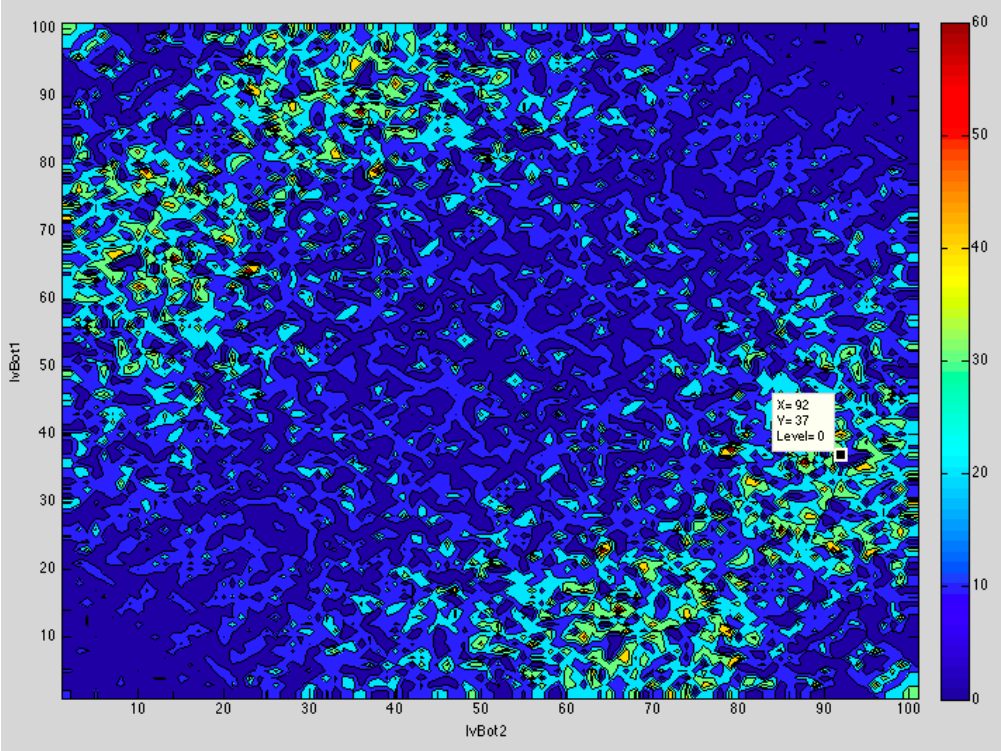



Figura 4.10: Heatmap della *Diff* per l'arma *Lanciagranate*

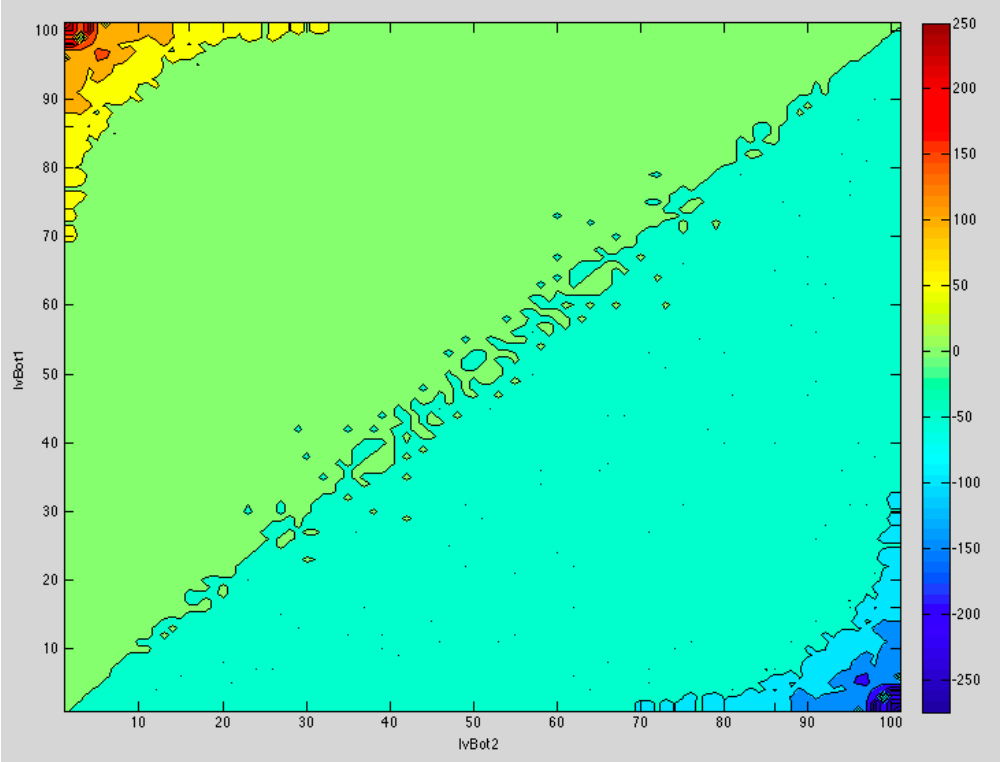



Figura 4.11: Heatmap del *Kill Ratio* per l'arma *Mitragliatore*

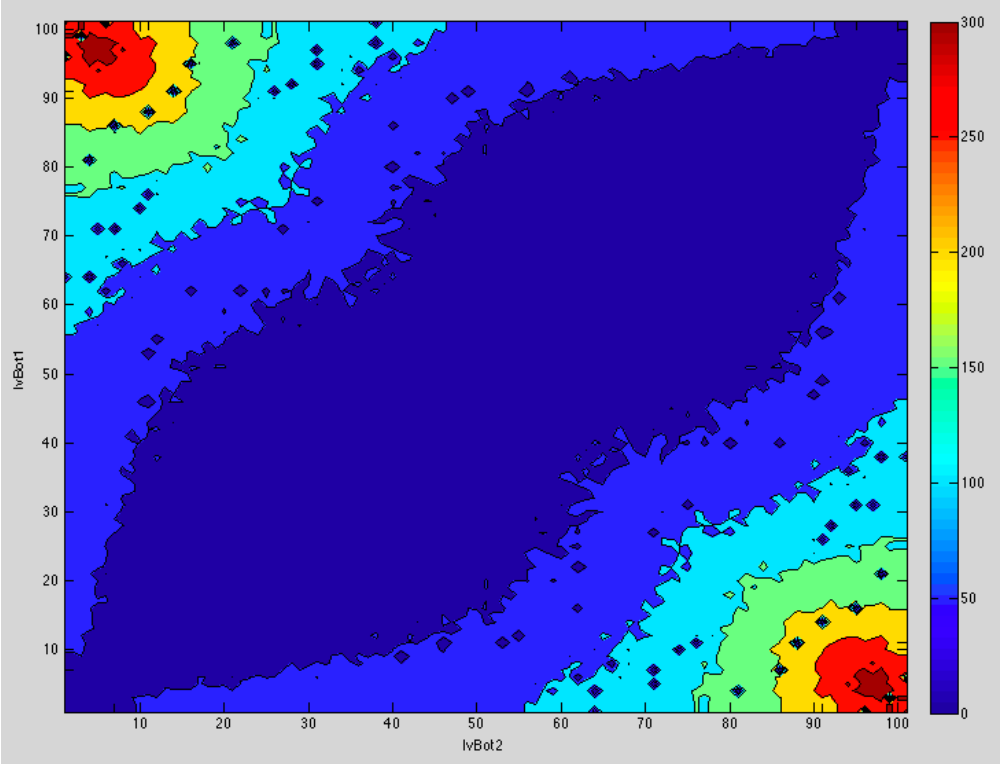



Figura 4.12: Heatmap della *Diff* per l'arma *Mitragliatore*

Possiamo innanzitutto osservare che il livello di abilità influenza molto le armi che necessitano solamente una buona mira per essere usate con efficienza. Infatti il *Kill Ratio* è estremamente elevato per grandi divari di livello di abilità tra i due *bot*: per la *pistola* il ratio massimo è 200 (Figura 4.1), per il *rifle* il ratio massimo è 300 (Figura 4.5) e per il *mitragliatore* il ratio massimo è 250 (Figura 4.11)). Le figure 4.2, 4.6 e 4.12 mostrano invece la *Diff* delle armi appena citate.

Il *bazooka*, pur avendo una *heatmap* del *Kill Ratio* con caratteristiche simili alle precedenti (Figura 4.3), ha il ratio massimo molto minore (pari a 15, meno di 1/10) rispetto alle armi precedentemente citate. È inoltre possibile osservare che la *heatmap* della *Diff* per il *bazooka* (Figura 4.4) mostra una forma diversa rispetto alle altre armi, presentando fasce marcate che identificano risultati simili. Queste due caratteristiche sono dovute al fatto che il *bazooka* (insieme al lanciagranate) è un'arma che necessita abilità per essere usata, altrimenti, sparando un colpo troppo vicino, è possibile incorrere in un suicidio. Nei videogiochi sparatutto in prima persona, i suicidi vengono penalizzati con la sottrazione di un punto dalle uccisioni totali effettuate dal giocatore che si è suicidato. Una differenza così marcata rispetto al ratio delle altre armi rispetto al *bazooka* è dovuto a ciò che abbiamo detto prima: l'abilità del *bot* non influenza la sua capacità nell'uso delle armi, ma solo la precisione della sua mira. Questo porta i *bot* a sparare quando vedono un nemico senza pensare se questo possa causare un suicidio, penalizzandoli nel caso utilizzino un'arma che provoca esplosioni.

La *heatmap* del *Kill Ratio* della *motosega* mostra caratteristiche (Figura 4.7) diverse da quelle delle quattro armi appena viste. Essa infatti presenta limiti molto più marcati e un valore massimo di ratio pari a 2,5 (100 volte meno rispetto a quello del *mitragliatore*). Queste caratteristiche sono dovute al fatto che, per usare un'arma corpo a corpo, i due *bot* si devono avvicinare. Quando il *bot* con abilità maggiore si avvicina all'altro anche quest'ultimo (il *bot* con abilità minore) può attaccare con successo, vanificando il vantaggio riguardante la velocità di mira del *bot* con abilità più elevata. La figura 4.8 mostra la heatmap della *Diff* della *motosega*.

Un discorso a parte è necessario per il *lanciagrante*. Come si può osservare, le figure 4.9 e 4.10 mostrano dei dati molto rumorosi, all'interno dei quali non è possibile identificare alcun pattern. La causa è duplice: l'esplosione dei colpi solo dopo alcuni rimbalzi rende difficile mirare correttamente ad un punto, rendendo ancora più facile provocare accidentalmente un suicidio rispetto a quanto avveniva con l'uso del *bazooka*. In secondo luogo, l'abilità del *bot* non gli permette di calcolare in modo corretto la traiettoria che il proiettile avrà, ma permette solo una mira più accurata. Utilizzando come arma il *lanciagranate*,

| Arma Utilizzata | *Ratio Accettabile* | *Kill Ratio* $< 5$ | *Kill Ratio* $\geq 10$ |
|---|---|---|---|
| Pistola | $DiffLvBot \leq 15$ | $DiffLvBot \leq 40$ | $DiffLvBot \geq 70$ |
| Bazooka | $DiffLvBot \leq 40$ | $DiffLvBot \leq 75$ | $DiffLvBot \geq 95$ |
| Rifle | $DiffLvBot \leq 20$ | $DiffLvBot \leq 50$ | $DiffLvBot \geq 80$ |
| Motosega | $DiffLvBot \leq 65$ | $\forall$ | $\nexists$ |
| Lanciagranate | ?? | ?? | ?? |
| Mitragliatore | $DiffLvBot \leq 30$ | $DiffLvBot \leq 55$ | $DiffLvBot \geq 70$ |

Tabella 4.1: *DiffLvBot* massima (minima) sotto (sopra) la quale il *Kill Ratio* è compreso nella soglia specificata

una mira migliore difficilmente servirà ad assicurarsi un'uccisione se non viene tenuta in considerazione la traiettoria che seguirà il proiettile.

In una partita multigiocatore di qualsiasi sparatutto in prima persona, il *Ratio K/D* medio dei giocatori è compreso tra 1 e 2. Giocando in due giocatori, se un giocatore presenta un ratio pari a 2 vuol dire che l'altro giocatore è morto il doppio delle volte rispetto alle uccisioni che ha fatto. Se il ratio di un giocatore è pari a 3 quel giocatore è molto più bravo rispetto all'altro. È quindi facile intuire come un ratio pari a 300 (il valore ottenuto con l'arma *rifle*) sia dovuto ad uno sbilanciamento enorme della partita.

Per questo motivo soffermandoci ora sul *Kill Ratio*, definendo alcune soglie utili per confrontare l'impatto che ha il cambiamento di abilità dei *bot* sull'esito della partita rispetto all'arma utilizzata. Queste soglie sono tre:

- $1 \leq$ *Kill Ratio* $\leq 2$
- *Kill Ratio* $< 5$
- *Kill Ratio* $\geq 10$

Definiamo la prima soglia, ovvero quella identificata da un *Kill Ratio* compreso tra 1 e 2, con il nome di *Ratio Accettabile*. La tabella 4.1 mostra il valore massimo (o minimo) della differenza tra il livello di abilità dei due *bot* sotto (o sopra) il quale il ratio è compreso nella soglia specificata. La differenza tra il livello di abilità dei due *bot* è definita come

$$DiffLvBot = |LvBot_1 - LvBot_2|$$

Ognuna delle tre soglie è valorizzata per tutte le armi in analisi.

| Arma Utilizzata | *Kill Ratio* massimo | *Diff* massimo |
|---|---|---|
| Pistola | 200 | 220 |
| Bazooka | 15 | 180 |
| Rifle | 300 | 350 |
| Motosega | 2,5 | 80 |
| Lanciagranate | 30 | 60 |
| Mitragliatore | 250 | 300 |

Tabella 4.2: Valori massimi di *Kill Ratio* e *Diff* per ogni arma

## 4.3 Analisi della soglia per il *Rifle*

Dopo aver analizzato come la differenza di abilità tra i due *bot*, per ogni arma, influisca sull'esito della partita, focalizziamoci ora su un'arma in particolare: il *rifle*. Abbiamo scelto quest'ultima perché è quella che provoca il maggior sbilanciamento (sia in termini di *Kill Ratio* che di *Diff*) variando la differenza di abilità dei *bot*, come è possibile vedere dalla Tabella riassuntiva 4.2.

Per generare mappe che bilancino la partita è necessario scegliere una funzione obiettivo adatta che verrà utilizzata dall'algoritmo genetico per valutare i candidati. Solitamente, nei videogiochi sparatutto in prima persona, i giocatori che dominano la partita sono quelli con il *Ratio K/D* più elevato. Prendiamo quindi come funzione obiettivo $f$ l'entropia delle uccisioni effettuate da tutti i giocatori, dove:

$$f = -\sum_{i=1}^{n} \frac{K_i}{K_{tot}} \log_2 \frac{K_i}{K_{tot}}$$

Con $K_i$ uccisioni effettuate dal giocatore $i$, $K_{tot}$ somma delle uccisioni di tutti i giocatori, $n$ numero totale di giocatori. Nella nostra analisi, avendo solo due *bot* in partita, questa formula si riduce a:

$$f = -\frac{K_{bot1}}{K_{bot1} + K_{bot2}} \log_2 \frac{K_{bot1}}{K_{bot1} + K_{bot2}} - \frac{K_{bot2}}{K_{bot1} + K_{bot2}} \log_2 \frac{K_{bot2}}{K_{bot1} + K_{bot2}}$$

L'entropia è la misura del disordine di un sistema e varia in un intervallo tra 0 e 1; 1 rappresenta il massimo disordine, 0 quello minimo. Nel nostro caso l'entropia misura il disordine delle uccisioni totali effettuate durante la partita e assume valore 0 se le uccisioni sono state fatte tutte da un solo *bot*, valore 1 se i due *bot* hanno effettuato lo stesso numero di uccisioni. Il nostro obiettivo è quindi massimizzare il valore della funzione obiettivo, facendo in modo che l'entropia delle uccisioni sia massima. In questo modo otteniamo una mappa che compensi il divario di abilità dei *bot*. Formalmente, la funzione di fitness

risulta quindi:

$$fitness = argmax(f)$$

Per valutare questa fitness per ogni mappa generata dall'algoritmo di generazione procedurale abbiamo la necessità di simulare una partita in modo da poter raccogliere dati sulle uccisioni effettuate da ogni *bot*. Per simulare una partita abbiamo bisogno di specificare l'arma che i *bot* useranno e il livello di abilità assegnato ad ognuno dei due *bot*. Nel nostro caso l'arma sarà il *rifle* per entrambi, ma la scelta dell'abilità dei due *bot* risulta più complicata. Per affrontare quest'ultimo problema sfruttiamo le analisi fatte nella sezione precedente ed in particolare concentriamo la nostra attenzione sulla Tabella 4.1 a Pagina 49. In essa, escludendo il *ratio accettabile* che non necessita di bilanciamento, abbiamo precedentemente identificato due soglie significative di divario di abilità tra *bot* per cui si ha un elevato cambiamento di *Kill Ratio*. Guardando la riga riferita al *rifle* possiamo vedere come i valori significativi sono:

$$DivLvBot \leq 50$$

$$DiffLvBot \geq 80$$

Per selezionare quale abilità dei *bot* dovremo usare per la valutazione delle mappe effettuiamo due volte il processo di generazione procedurale con algoritmi genetici. Una prima volta simulando le partite con *bot* che hanno un differenza di abilità $\leq 50$ e una seconda volta con differenza di abilità $\geq 80$. Successivamente eseguiamo nuovamente l'analisi fatta nella sezione precedente (ma, questa volta, solo per l'arma *rifle*) una volta sulla miglior mappa generata dal primo processo di generazione e una sulla miglior mappa generata dal secondo. A questo punto avremo due *heatmap* dalle quali potremo decretare quale differenza di abilità dei *bot* bilancia meglio tutte le possibili differenze di abilità. I livelli di abilità che utilizziamo per le due esecuzioni del processo di generazione procedurale delle mappe con algoritmi genetici sono:

- per la differenza di abilità $\leq 50$ (**configurazione 35-80**):
  Abilità$_{bot1} = 35$, Abilità$_{bot2} = 80$ (differenza di abilità = 45)

- per la differenza di abilità $\geq 80$ (**configurazione 15-95**):
  Abilità$_{bot1} = 15$, Abilità$_{bot2} = 95$ (differenza di abilità = 80)

L'algoritmo genetico utilizzato per cercare la soluzione migliore nello spazio di ricerca è un algoritmo genetico semplice, con *crossover a un punto*, *mutazione semplice* e *tournament selection a 2 candidati* [17]. L'algoritmo di generazio-

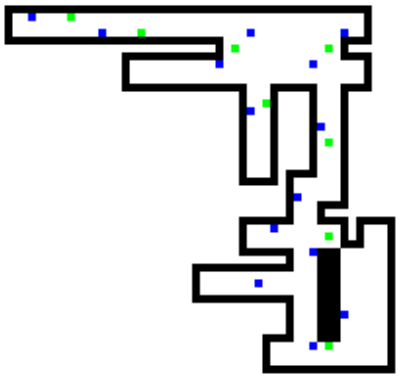


Figura 4.13: Mappa vincitrice dell'evoluzione con divario di abilità bot 45

ne procedurale utilizza questo algoritmo genetico per 30 generazioni su una popolazione di 100 mappe candidate.

### 4.3.1 Evoluzione con *Rifle* e configurazione 35-80

Effettuiamo una prima generazione utilizzando l'algoritmo di generazione procedurale con algoritmi genetici impostando, per la simulazione della partita nelle mappe generate, il livello di abilità del primo *bot* a 35 e il livello di abilità del secondo a 80, dotando entrambi di arma *rifle*. Una volta terminata l'esecuzione, la mappa migliore ricavata è quella mostrata in figura 4.13. Utilizziamo questa mappa per simulare una partita per ogni possibile combinazione di abilità dei due *bot*, con esclusione delle permutazioni, ricavando i dati e creando con essi due *heatmap*. Una che mostra il *Kill Ratio* (Figura 4.14) e una che mostra l'entropia delle uccisioni (Figura 4.15).

### 4.3.2 Evoluzione con *Rifle* e configurazione 15-95

Allo stesso modo, eseguiamo nuovamente il processo di generazione procedurale di contenuti con algoritmi genetici cambiando l'abilità dei *bot* e mantenendo tutto il resto con le stesse impostazioni. L'abilità del $bot_1$ è ora impostata su 15, quella del $bot_2$ invece su 95. La figura 4.16 mostra la mappa migliore ottenuta dall'evoluzione e le figure 4.17 e 4.18 sono *heatmap* che rappresentano i dati ottenuti su questa mappa per ogni livello di abilità dei bot (rispettivamente del *Kill Ratio* e dell'entropia delle uccisioni), allo stesso modo di quanto avvenuto nella sottosezione precedente.

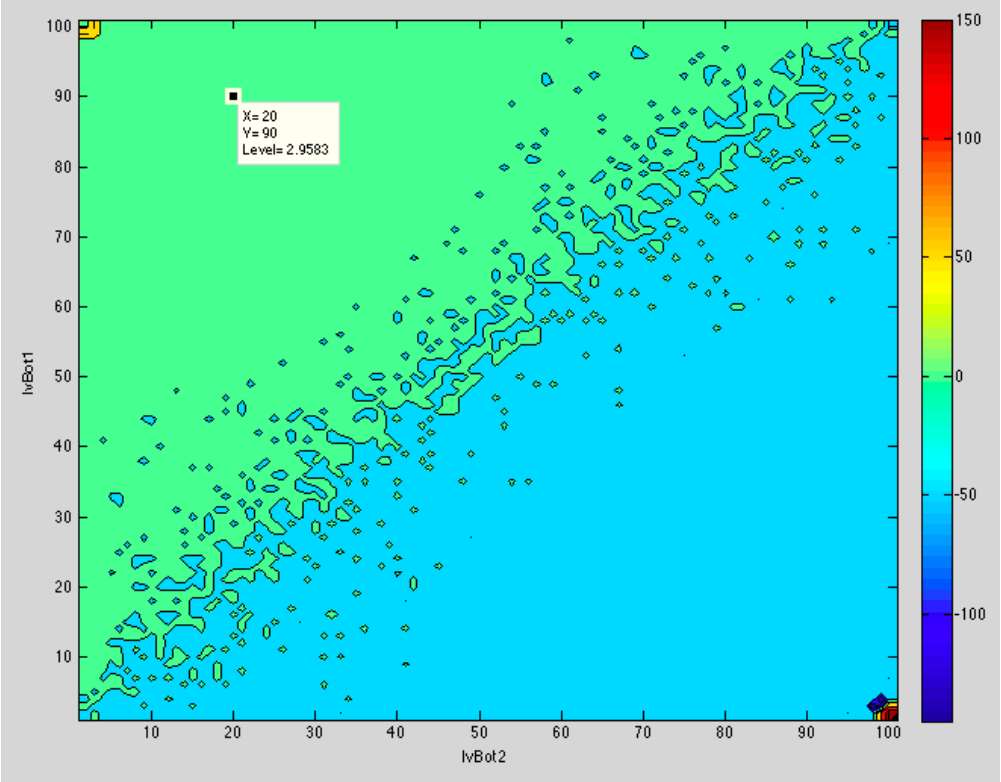



Figura 4.14: *heatmap* del *Kill Ratio* per ogni livello dei due bot. Dati raccolti sulla mappa vincitrice dell'evoluzione con *DiffLvBot*=45

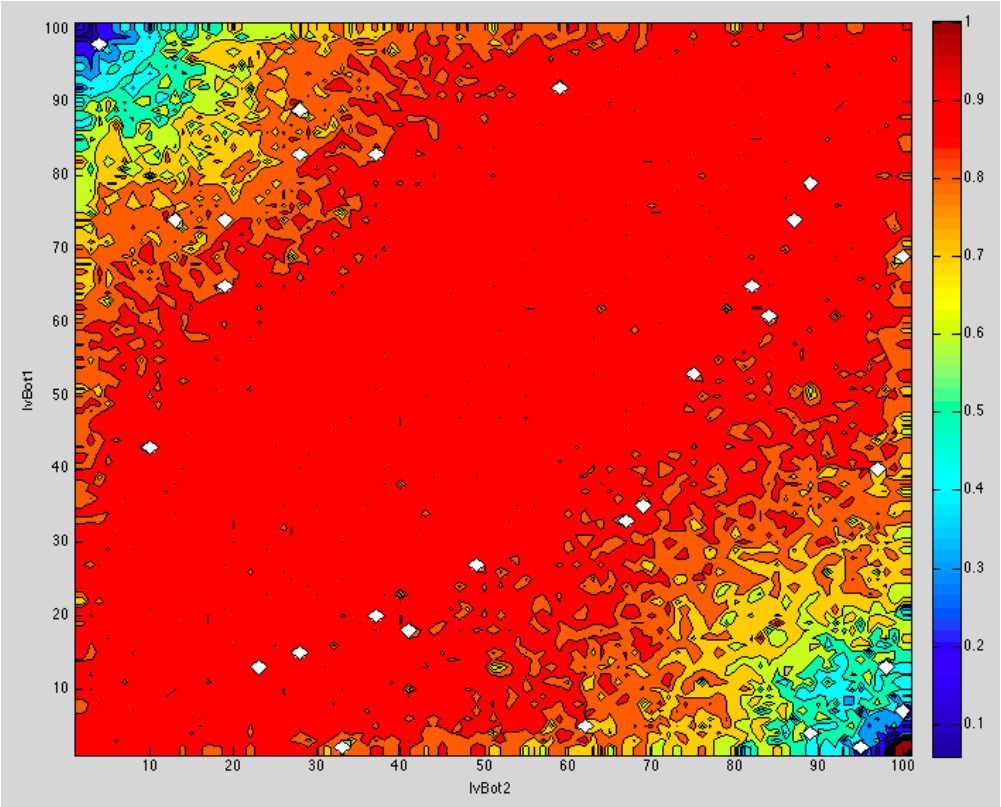



Figura 4.15: *heatmap* dell'entropia delle uccisioni per ogni livello dei due bot. Dati raccolti sulla mappa vincitrice dell'evoluzione con *DiffLvBot*=45

Figura 4.16: Mappa vincitrice dell'evoluzione con divario di abilità bot 80

### 4.3.3 Confronto delle evoluzioni con arma *Rifle*

Possiamo confrontare i grafici mostrati in 4.3.1 e 4.3.2 tra loro per trovare la differenza di abilità che, usata per la simulazione della partita nella fase di valutazione del contenuto, permetta di generare una mappa che massimizzi il bilanciamento di gioco ottenuto. Per farlo necessitiamo però di un altro grafico, quello dell'entropia per tutti i livelli di abilità dei *bot* ottenuti dalla simulazione sulla mappa vuota. Questa *heatmap* è mostrata nella figura 4.19.

Analizziamo ora i risultati ottenuti mettendo a confronto i valori di entropia nei 3 casi in analisi (mappa vuota, mappa con *DivLvBot* di 45 e mappa con *DivLvBot* di 80) per stessi divari di abilità tra *bot*. Tralasciamo divari bassi e concentriamoci su quelli elevati, nello specifico su divari di abilità pari a 70, 80 e 90. Per *DivLvBot* pari a 70 si hanno 30 esempi analizzabili (70-0, 71-1, 72-2, ... , 99-29, 100-30), per *DivLvBot* pari a 80 se ne hanno 20 (80-0, 81-1, 82-2, ... , 99-19, 100-20) ed infine per *DivLvBot* pari a 90 se ne hanno 10 (90-0, 91-1, 92-2, ... , 99-9, 100-10). Le permutazioni non vengono considerate in quanto sarebbero esempi replicati già presi in analisi. Le figure che seguono mettono a confronto i valori di entropia degli esempi sopra citati nei 3 casi in analisi. Rispettivamente, la figura 4.20 mostra i valori di entropia per esempi di *DivLvBot* uguale a 70, la figura 4.21 per esempi di *DivLvBot* uguale a 80 e la figura 4.22 per esempi di *DivLvBot* uguale a 90. In queste figure l'entropia degli esempi ricavati sulla mappa vuota sono rappresentate da barre *blu*, quelli ricavati sulla mappa vincitrice dell'evoluzione con *DivLvBot* pari a 45 sono rappresentati da barre *verdi* e quelli ricavati sulla mappa vincitrice dell'evoluzione con *DivLvBot* pari a 80 sono rappresentati da barre *rosse*.

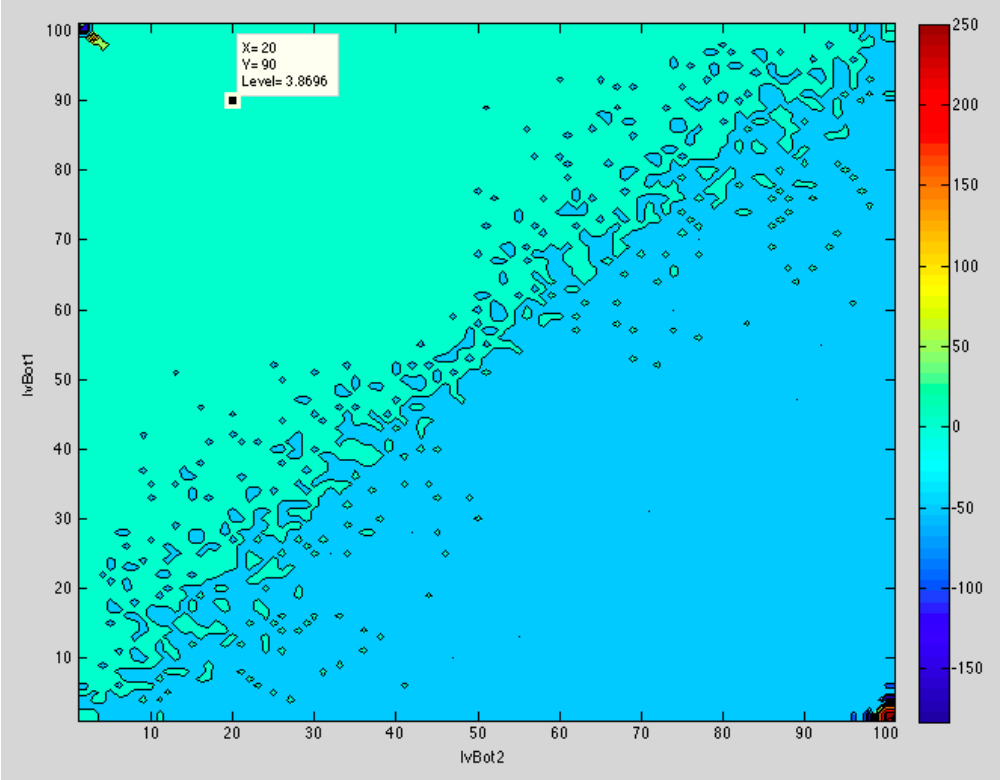



Figura 4.17: *heatmap* del *Kill Ratio* per ogni livello dei due bot. Dati raccolti sulla mappa vincitrice dell'evoluzione con *DiffLvBot*=80

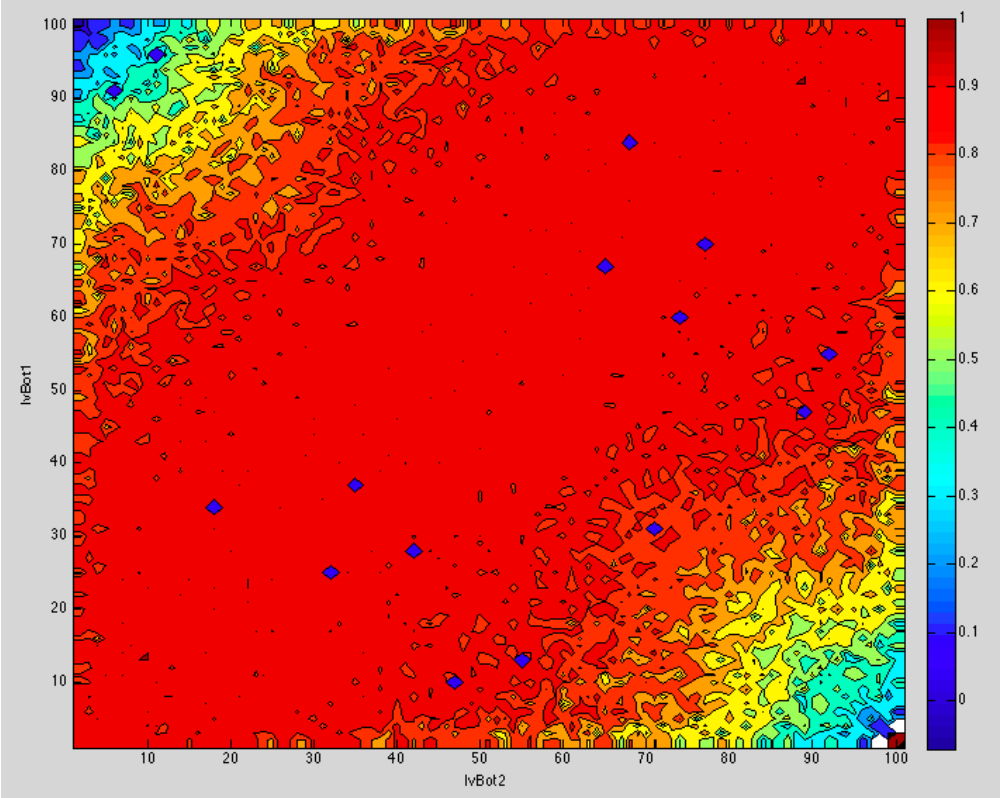



Figura 4.18: *heatmap* dell'entropia delle uccisioni per ogni livello dei due bot. Dati raccolti sulla mappa vincitrice dell'evoluzione con *DiffLvBot*=80

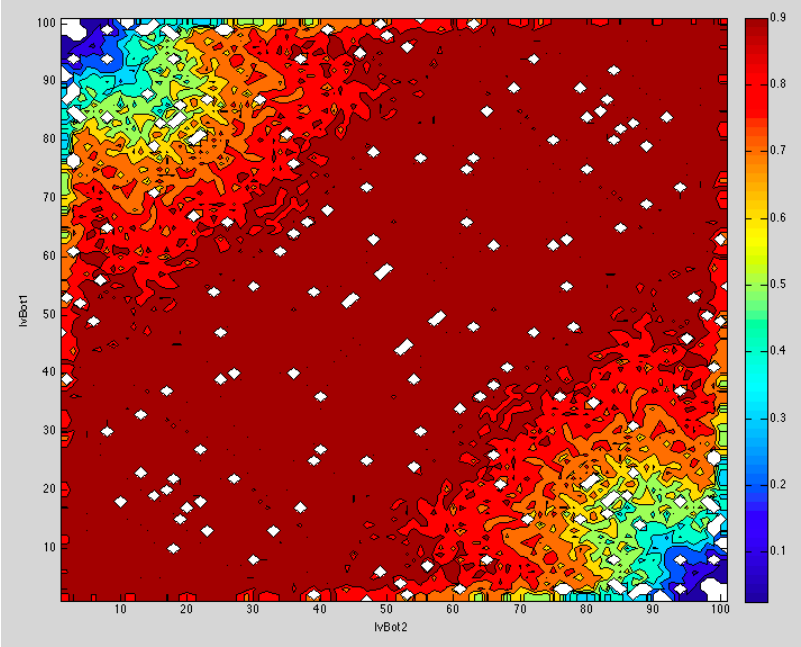



Figura 4.19: *heatmap* dell'entropia delle uccisioni per ogni livello dei due bot. Dati raccolti sulla mappa vuota.

Dai tre istogrammi si nota che il valore più alto di entropia è, per la maggior parte delle volte, dato dagli esempi ottenuti dalla mappa vincitrice dell'evoluzione con *DivLvBot* uguale a 45. Possiamo quindi ipotizzare che l'utilizzo di questo divario di abilità tra i due *bot* per simulare le partite porti ad ottenere mappe maggiormente bilanciate rispetto a quelle che verrebbero generate da una simulazione con *DivLvBot* uguale a 80.

La nostra ipotesi risulta fondata nel momento in cui si analizza il grafico mostrato in figura 4.23. Esso mostra l'andamento dell'entropia media per tutti i valori di differenza di abilità dei *bot* (si noti che, a differenza degli istogrammi precedenti, i risultati per la mappa vincitrice dell'evoluzione con *DivLvBot* uguale a 45 sono rappresentati con colore *rosso*, mentre quelli per la mappa vincitrice dell'evoluzione con *DivLvBot* uguale a 80 sono rappresentati con il *verde*). L'asse delle ordinate rappresenta il valore dell'entropia media, l'asse delle ascisse il *DivLvBot* e il punto identifica il valore medio dell'entropia per un determinato *DivLvBot*. Ogni istogramma precedentemente citato è quindi rappresentato in questo grafico come un punto per ogni caso in analisi, per un totale di 3 punti ad istogramma. Da quest'ultimo grafico è facilmente individuabile che, per quasi ogni differenza di abilità tra i *bot*, la mappa generata con *DivLvBot* uguale a 45 risulta quella con entropia delle kill più elevata. Si noti che, per valori limite di differenza di abilità tra i *bot*, la media è basata su pochi

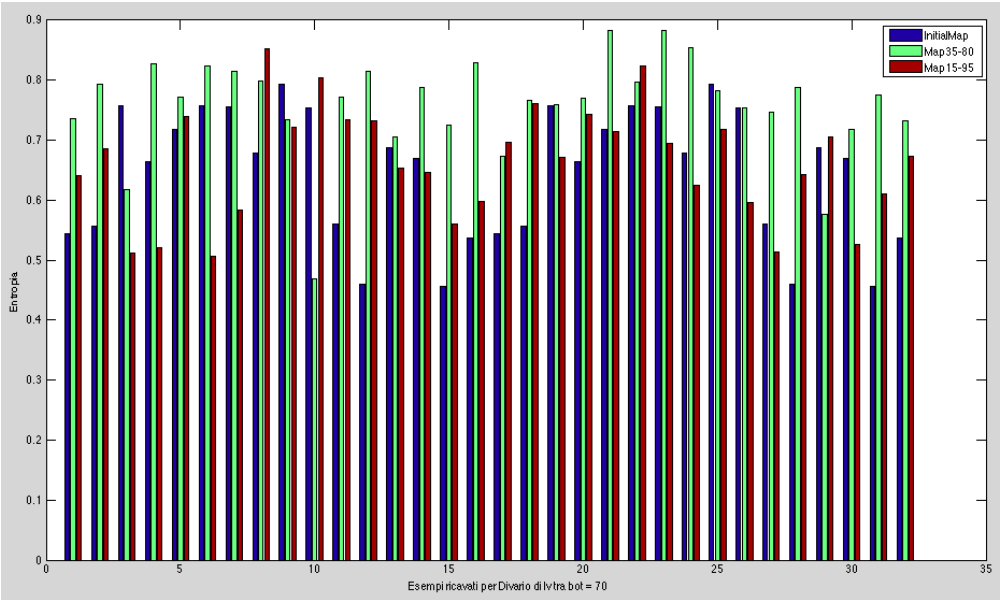



Figura 4.20: Esempi di entropia delle uccisioni per *DivLvBot*=70 nelle 3 analisi

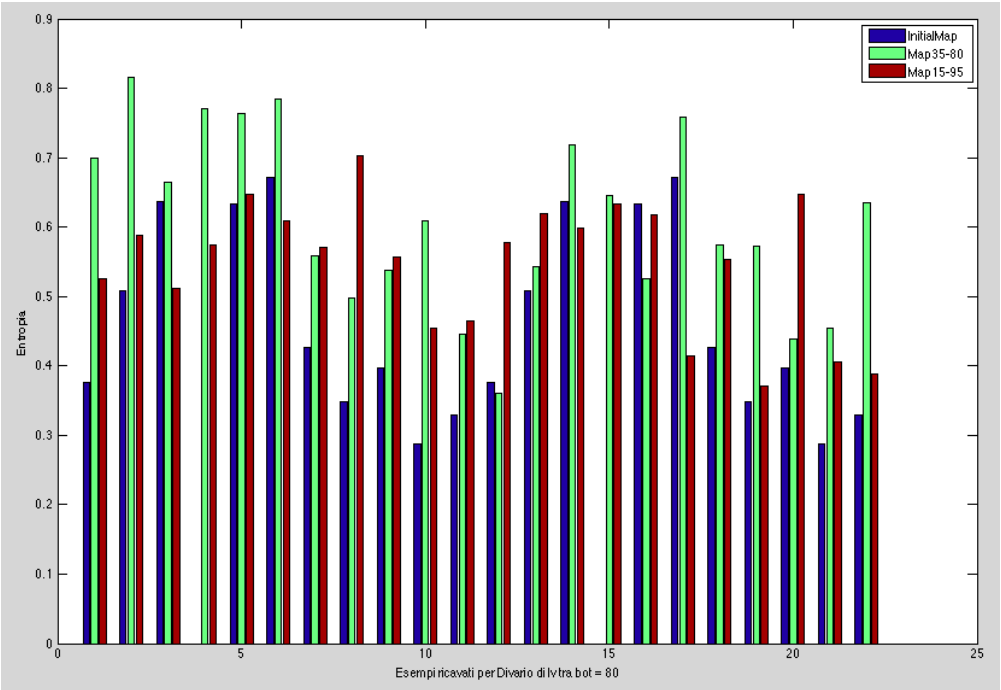



Figura 4.21: Esempi di entropia delle uccisioni per *DivLvBot*=80 nelle 3 analisi

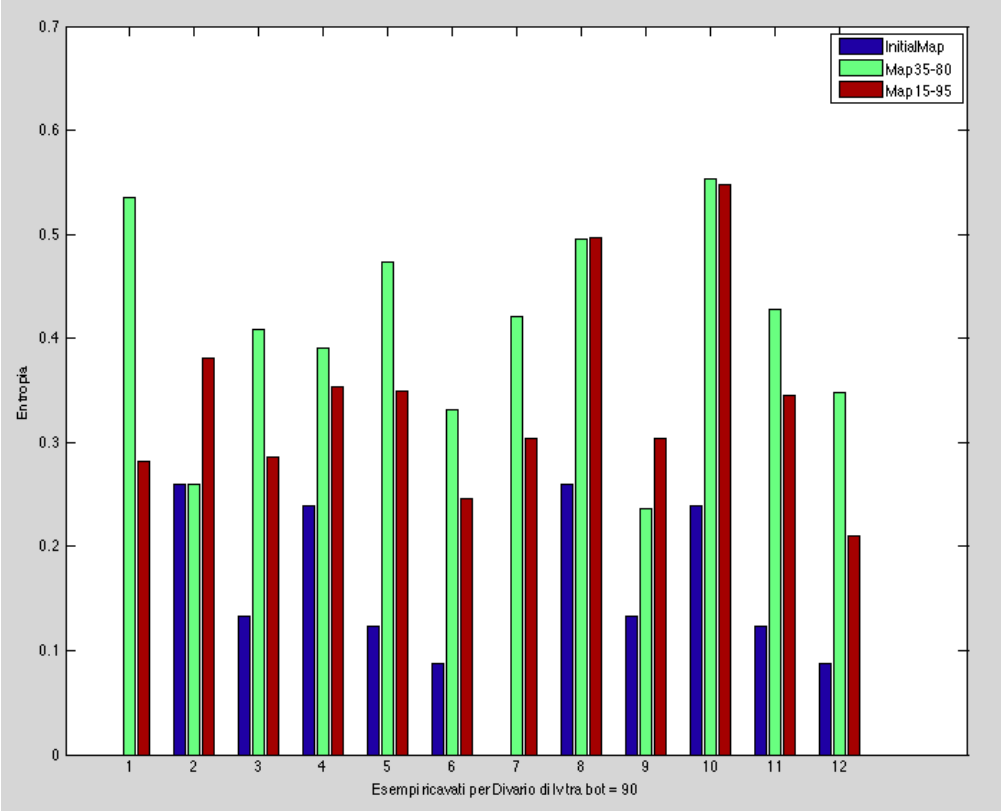



Figura 4.22: Esempi di entropia delle uccisioni per *DivLvBot*=90 nelle 3 analisi

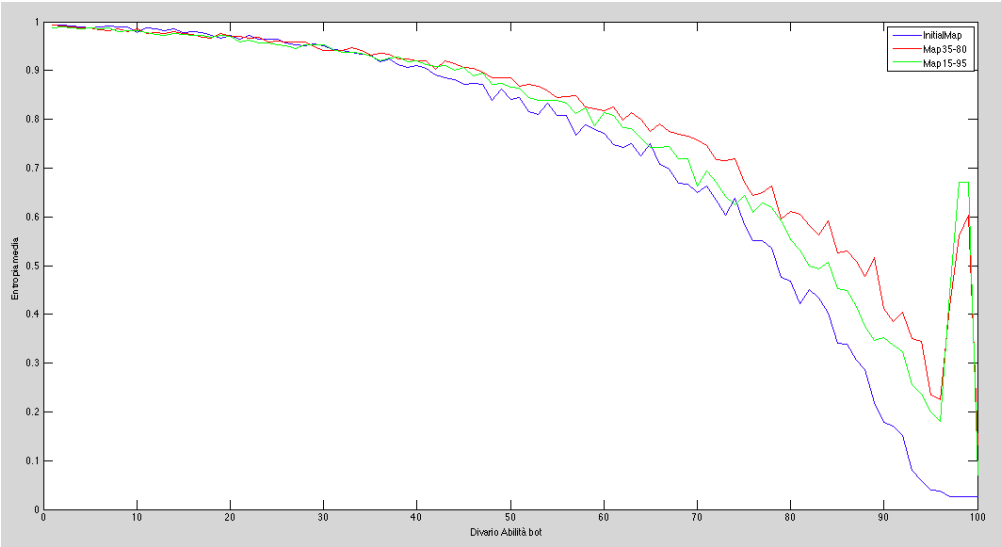



Figura 4.23: Etropia delle uccisioni media per ogni divario di bot nelle 3 analisi

dati (per esempio, per *DivLvBot* uguale a 98 gli esempi che contribuiscono alla media sono 3: 98-0, 99-1, 100-2), ne basta quindi uno errato per comprometterla. Per questo motivo il grafico in figura 4.23 ha valori erronei per alcuni valori limite di *DivLvBot* (98 e 99). Nonostante questo la mappa generata con *DivLvBot* uguale a 45 risulta chiaramente migliore in merito alla nostra analisi. Per questo motivo, quando, nei prossimi capitoli, valuteremo le mappe generate utilizzando *bot* dotati (entrambi) di arma *rifle* per la simulazione delle partite, le abilità impostate ai due *bot* saranno 35 e 80 (con, quindi, *DivLvBot* pari a 45).

## 4.4 Sommario

In questo capitolo abbiamo analizzato i *bot* di Cube 2 (Sezione 4.1) descrivendone l'IA e il livello di abilità. Successivamente abbiamo analizzato, per tutte le armi del gioco, come il livello di abilità dei *bot* influenzi l'esito della partita data la stessa arma (Sezione 4.2). Per farlo abbiamo introdotto il concetto di *Kill Ratio*, di *Diff* e abbiamo mostrato i risultati ottenuti utilizzando delle *heatmap*. Concentrandoci poi sull'analisi dall'arma *rifle* (Sezione 4.3) abbiamo introdotto l'entropia delle kill, una misura utile come funzione obiettivo della funzione di fitness per misurare quanto una mappa risulti bilanciata. Abbiamo poi replicato le analisi sui diversi livelli di abilità dei *bot* (con arma rifle) su altre due mappe, la prima vincitrice dell'evoluzione che utilizza due *bot* con differenza di abilità pari a 45 per simulare la partita (SottoSezione 4.3.1), la seconda vincitrice dell'evoluzione che utilizza due *bot* con differenza di abilità pari a 80 per simulare la partita (SottoSezione 4.3.2). Abbiamo poi confrontato i risultati di queste due analisi (Sottosezione 4.3.3) concludendo che, utilizzando una simulazione con *bot* che hanno differenza di abilità pari a 45, si ottengono mappe che portano ad avere partite maggiormente bilanciate.

# Capitolo 5

# Esperimenti su singolo parametro

In questo capitolo mostriamo i risultati dei nostri esperimenti sul bilanciamento del gioco, ottenuti utilizzando la generazione procedurale di mappe con algoritmi genetici. La funzione obiettivo utilizzata valuta la mappa generata simulando una partita tra due *bot* in modalità *deathmatch*. Inizialmente, utilizzando *bot* dotati di una stessa arma, mostriamo i risultati degli esperimenti al variare della differenza di abilità tra i due *bot*. Successivamente mostriamo i risultati ottenuti utilizzando due *bot* con abilità uguali ma armi diverse.

## 5.1 Introduzione

Come precedentemente discusso, i *bot* di *Cube 2* utilizzano un unico algoritmo nel quale è possibile specificare solo il livello di abilità (un valore compreso tra 0 a 100) che ne definisce l'efficacia nella mira e la velocità con cui individua i bersagli. L'utilizzo di una sola IA uguale per tutti i *bot* non ci consente di effettuare analisi sul bilanciamento di partite alle quali partecipino giocatori con strategie diverse [27]. Per simulare questa situazione, infatti, avremmo bisogno che ogni *bot* avesse una diversa IA che implementi comportamenti diversi. Questa limitazione ci porta a focalizzare la nostra attenzione su due soli possibili tipi di sbilanciamento valutabili in questa situazione: (i) il divario del livello di abilità tra i due *bot* e (ii) l'assegnazione di armi differenti ad essi.

Prima di effettuare le nostre analisi, abbiamo studiato l'algoritmo di generazione procedurale di mappe implementato da Cardamone et al. in [12] e abbiamo considerato le quattro rappresentazioni utilizzate dall'algoritmo, scegliendone un sottoinsieme utile al nostro scopo. Tra le quattro rappresentazioni presenti, *All-White* (mappa completamente vuota, il genoma codifica la posizione nel-

la quale vengono aggiunti i muri), *All-Black* (l'opposto della *All-White*, mappa completamente piena di muri i quali vengono rimossi gradualmente creando aree libere), *Grind* (mappa codificata da una griglia di muri $9 \times 9$, il genoma codifica quali muri della griglia sono attivi e quali no) e *Random-Digger* (mappa completamente piena di muri, il genoma codifica i passi di un agente che, partendo da un punto, scava i punti della mappa creando un percorso calpestabile), abbiamo scelto di utilizzane due, la *All-Black* e la *Grind*. Il motivo di questa scelta è da ricercarsi nel fatto che la rappresentazione *Random-Digger* possiede una rappresentazione del genoma indiretta, mentre la rappresentazione *All-White* genera mappe molto simili alla *Grind* utilizzando però una codifica che permette una maggiore libertà nella creazione della mappa rispetto a quest'ultima. Nel *Random-Digger*, il genoma codifica solo 4 valori che rappresentano le probabilità con cui l'agente, che libera i blocchi della mappa, effettui determinate azioni (come, per esempio, girare a destra). L'utilizzo di questa codifica porta a due conseguenze, il principio di località (piccolo cambiamento nel genoma implica piccolo cambiamento nel fenotipo) non è rispettato e, per questo motivo, l'algoritmo finisce con l'effettuare una ricerca casuale nello spazio, non risultando utile al nostro scopo. Nella *All-White* invece, le mappe generate sono molto simili a quelle ottenute da una rappresentazione *Grind* ma, mentre la *All-White* utilizza una codifica della posizione dei muri e della loro lunghezza, la seconda codifica solo se il muro è presente oppure no. Seppur con la prima rappresentazione sia possibile generare mappe con maggiore liberà, al nostro fine è maggiormente utile utilizzare una rappresentazione stabile che ricerchi in uno spazio di ricerca ridotto, la rappresentazione *Grind*.

Dopo aver selezionato un sottoinsieme delle rappresentazioni di Cardamone et al. [12] da utilizzare, abbiamo modificato l'operatore di crossover utilizzato dall'algoritmo genetico. Al *single point crossover*, utilizzato nell'implementazione iniziale, abbiamo sostituito il *matrix crossover*, un crossover *ad-hoc* che considera la natura bidimensionale del genoma (il genoma codifica una matrice). Viene scelto inizialmente un punto casuale del genoma (come avviene nel *single point crossover*), tracciando poi l'asse orizzontale e verticale al punto la matrice viene spezzata in quattro sottomatrici. Una di esse (superiore sinistra, superiore destra, inferiore sinistra, inferiore destra) viene scelta casualmente e i due genomi che subiscono l'operazione di crossover si scambiano la sottomatrice scelta. La figura 5.1 (tratta da [18]) mostra come agisce il *matrix crossover*. L'utilizzo di questo operatore di crossover ci permette di rispettare maggiormente il principio di località, ricercando efficientemente nello spazio di ricerca la mappa migliore. Infine, per gli esperimenti mostrati in questo capitolo, utilizzeremo una funzione di fitness che massimizza l'entropia delle uccisioni effettuate dai

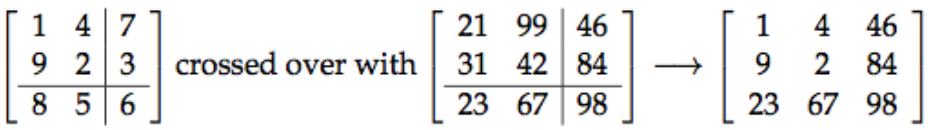



Figura 5.1: *Matrix crossover* tra due genomi

due *bot*, in modo da trovare una mappa che bilanci la partita a seconda della situazione di gioco. La funzione di fitness, utilizzata già nel capitolo precedente, è quindi:

$$fitness = argmax(-\frac{K_{bot1}}{K_{bot1}+K_{bot2}} \log_2 \frac{K_{bot1}}{K_{bot1}+K_{bot2}} - \frac{K_{bot2}}{K_{bot1}+K_{bot2}} \log_2 \frac{K_{bot2}}{K_{bot1}+K_{bot2}})$$

L'algoritmo, in tutti gli esperimenti effettuati in questo capitolo, ha utilizzato una popolazione di 100 candidati fatta evolvere per 30 generazioni, utilizzando un algoritmo genetico semplice con *matrix crossover*, *mutazione semplice* e *tournament selection a 2 candidati*.

## 5.2 Bilanciamento per bot con arma uguale

Analizziamo ora i risultati degli esperimenti condotti sulla generazione di mappe sia con rappresentazione *All-Black* che con *Grind*. La valutazione delle mappe candidate è ottenuta tramite la simulazione di una partita nella mappa utilizzando due *bot* che hanno in dotazione una stessa arma, l'unica utilizzabile durante la partita. È necessario inoltre osservare che sono state rimosse tutte le risorse presenti sulla mappa, come medipack o altre armi, in modo da ottenere dati riconducibili al solo svantaggio, imposto da noi inizialmente, di un *bot* rispetto all'altro.

### 5.2.1 Arma *Rifle*, Abilità bot 50-50

Impostando come arma il *rifle* e come abilità 50 ad entrambi i *bot*, verifichiamo che le partite siano effettivamente bilanciate data una situazione iniziale bilanciata (stessa arma e stessa abilità). La figura 5.2 mostra il grafico dell'andamento dell'entropia media e massima dei candidati per ogni generazione utilizzando la rappresentazione *All-Black*. Come è possibile vedere, l'entropia massima (pari a 1, ovvero il bilanciamento totale) è raggiunta subito alla prima generazione, indice del fatto che la soluzione ottima al problema è stata trovata generando mappe in maniera casuale. Infatti, la valutazione della prima generazione avviene su mappe codificate da genomi generati in maniera casuale all'inizio dell'algoritmo. Da queste analisi possiamo concludere che il problema

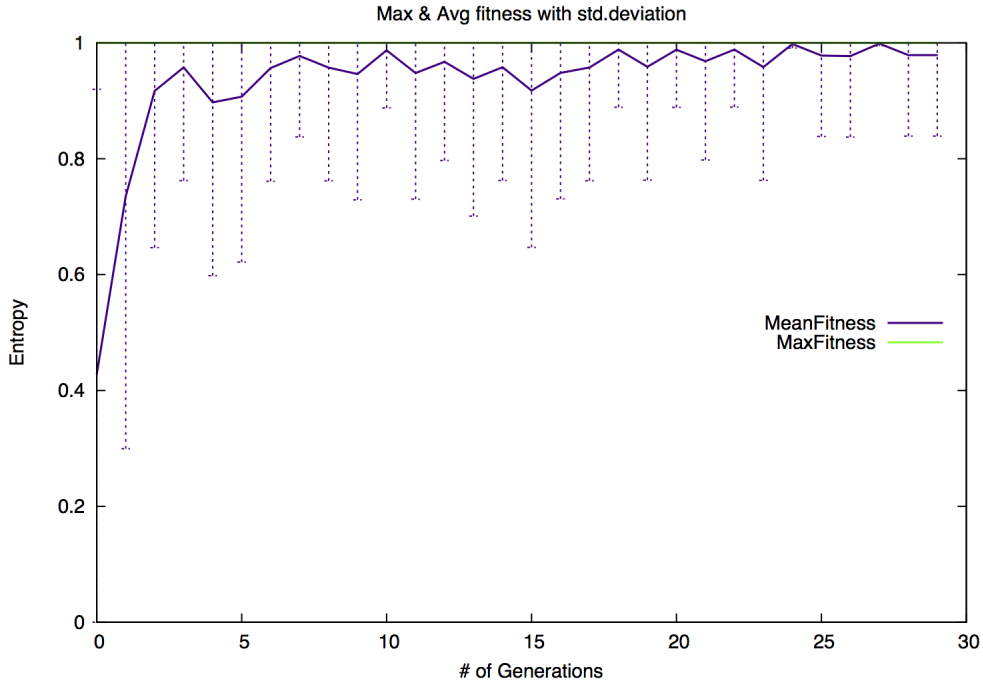



Figura 5.2: Fitness massima e media con rappresentazione *All-Black* nelle generazioni, Arma *Rifle*, Abilità 50-50

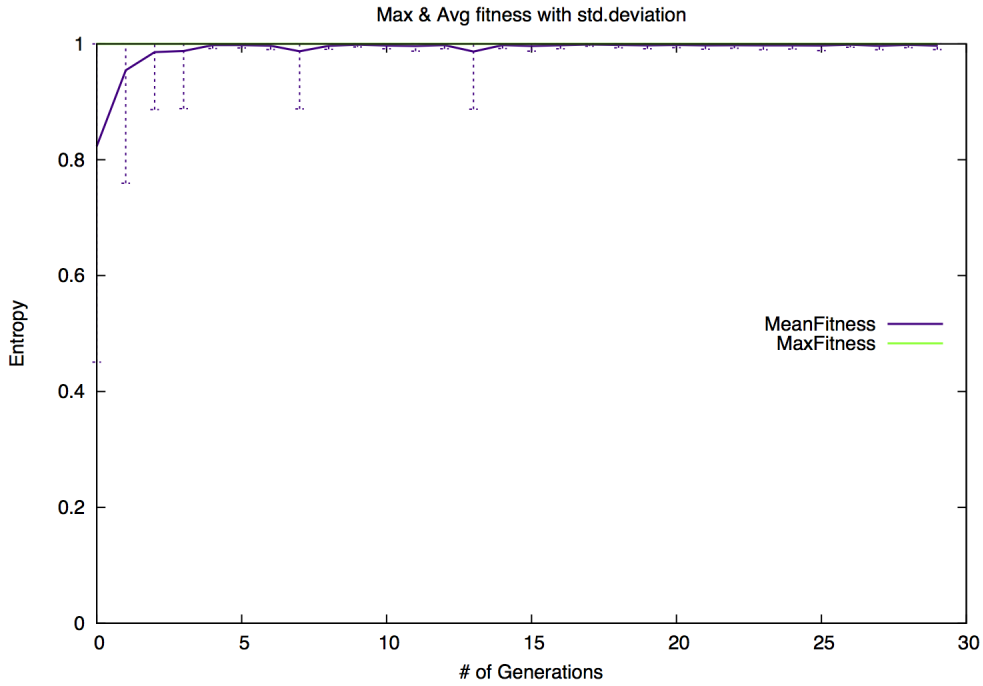
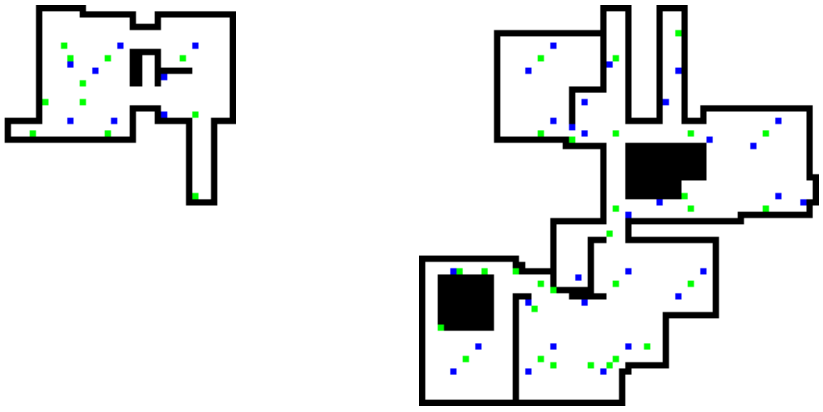



Figura 5.3: Fitness massima e media con rappresentazione *Grind* nelle generazioni, Arma *Rifle*, Abilità 50-50



(a) Mappa $1^a$ evoluzione (b) Mappa $2^a$ evoluzione

Figura 5.4: Mappe vincitrici dell'evoluzione con rappresentazione *All-Black*, arma *Rifle*, abilità 50-50

del bilanciamento è troppo semplice adottando questa configurazione, in quanto una qualsiasi mappa risulta bilanciata. Osservando la figura 5.3 è possibile constatare che, utilizzando la rappresentazione *Grind*, la situazione non cambia, il problema del bilanciamento, anche in questo caso, risulta troppo semplice. Le figure 5.4a e 5.4b mostrano 2 delle mappe migliori alla fine dell'evoluzione con rappresentazione *All-Black*. Come è possibile osservare, le due mappe sono molto diverse tra loro ma, nonostante questo, entrambe hanno entropia pari a 1. Da ciò si può dedurre che, data la configurazione usata in questa analisi, la partita risulta bilanciata indipendentemente dalla mappa in cui si svolge la partita.

### 5.2.2 Arma *Motosega*, Abilità bot 50-50

Dopo aver analizzato un'arma da distanza, verifichiamo ora se si ottengono gli stessi risultati dotando i *bot* di un'arma da ravvicinato. Mantenendo quindi il livello di abilità di entrambi i *bot* a 50, imponiamo loro di utilizzare solo la *motosega*. Le figure 5.5 e 5.6 mostrano i risultati dell'evoluzione ottenuti rispettivamente con rappresentazione *All-Black* e *Grind*. Mentre i risultati ottenuti con la rappresentazione *All-Black* sono simili a quelli dell'analisi effettuata nel sottocapitolo precedente (ovvero il problema è troppo semplice), quelli ottenuti con l'uso della rappresentazione *Grind* risultano diversi. Come possiamo vedere in figura 5.6, il valore di fitness massimo ottenuto da un candidato non riaggiunge mai il valore 1. La motivazione è da ricercarsi nel fatto che la rappresentazione *All-Black* permette di creare facilmente mappe più piccole rispetto alla *Grind*. Utilizzando un'arma da ravvicinato in una mappa troppo grande, le uccisioni totali accumulate nella partita dai due *bot* saranno poche (in quanto i

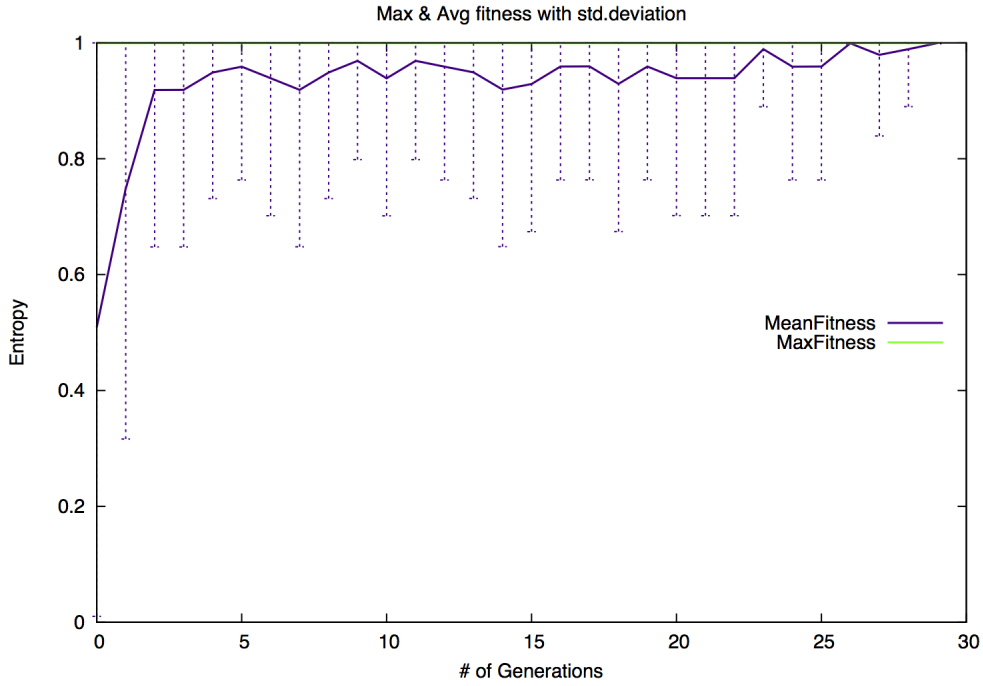



Figura 5.5: Fitness massima e media con rappresentazione *All-Black* nelle generazioni, Arma *Motosega*, Abilità 50-50

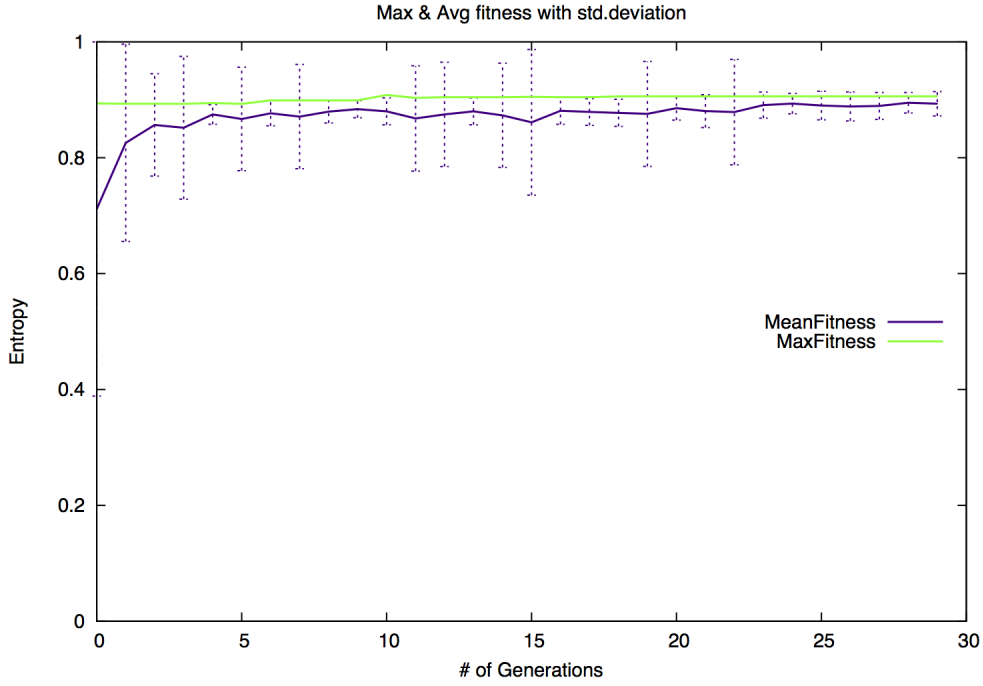
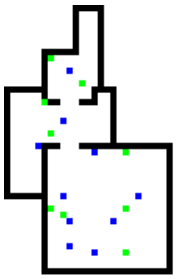
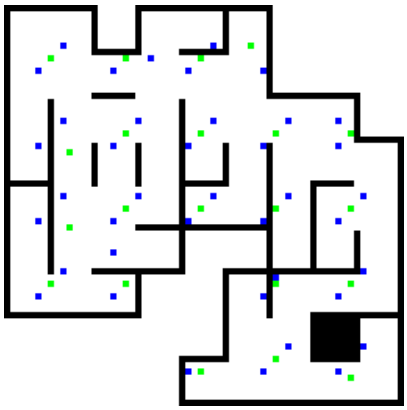



Figura 5.6: Fitness massima e media con rappresentazione *Grind* nelle generazioni, Arma *Motosega*, Abilità 50-50



(a) Mappa migliore con rappresentazione *All-Black*



(b) Mappa migliore con rappresentazione *Grind*

Figura 5.7: Mappe vincitrici dell'evoluzione con arma *Motosega*, abilità 50-50

*bot* si incontreranno raramente). Questo vorrà dire che una piccola differenza di uccisioni tra un *bot* e l'altro alla fine della partita impatterà notevolmente sul valore dell'entropia finale, riducendolo. Infatti, la mappa migliore ricavata dell'evoluzione con rappresentazione *All-Black* (finess uguale a 1) è molto piccola (Figura 5.7a) mentre quella ricavata con rappresentazione *Grind* (fitness uguale a 0.92) è più grande (Figura 5.7b). Nonostante questa differenza, in entrambi i casi il problema risulta troppo semplice, anche se con la rappresentazione *Grind* abbiamo dei piccolissimi miglioramenti durante le generazioni.

### 5.2.3 Arma *Rifle*, Abilità bot 35-80

Dopo aver analizzato i risultati ottenuti dal bilanciare partite con *bot* aventi stessa abilità e stessa arma, creiamo ora una situazione di svantaggio di un *bot* rispetto all'altro in modo che la partita risulti sbilanciata. Per farlo, cambiamo l'abilità dei *bot* in modo che uno risulti più forte dell'altro mantenendo però la stessa arma per entrambi. Partendo con l'arma *rifle*, dobbiamo decidere che divario di abilità dare ai *bot*. Grazie alle analisi da noi effettuate nel capitolo precedente e, nello specifico, nella sezione 4.3, abbiamo trovato che il divario migliore da utilizzare per ottenere la mappa che porta il massimo miglioramento è *DivLvBot* pari a 45. Per questo motivo, per valutare le mappe generate (con entrambe le rappresentazioni), dotiamo entrambi i *bot* dell'arma *rifle* e impostiamo il livello di abilità del primo a 35 e quello del secondo a 80. La figura 5.8 mostra l'andamento della fitness nelle generazioni per la rappresentazione *All-Black*. Come si può vedere, si ha un miglioramento costante con l'avanzare delle generazioni, segno che, agendo sulla mappa, si sta riducendo il vantaggio tra i due *bot*, bilanciando la partita. Alla fine dell'evoluzione, la mappa con va-

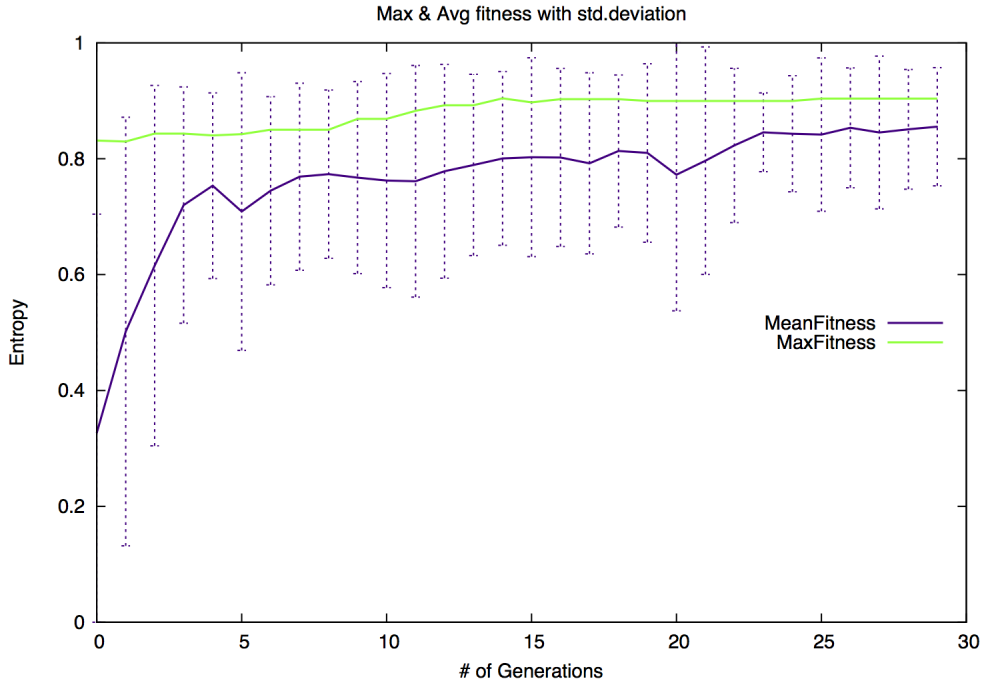



Figura 5.8: Fitness massima e media con rappresentazione *All-Black* nelle generazioni, Arma *Rifle*, Abilità 35-80

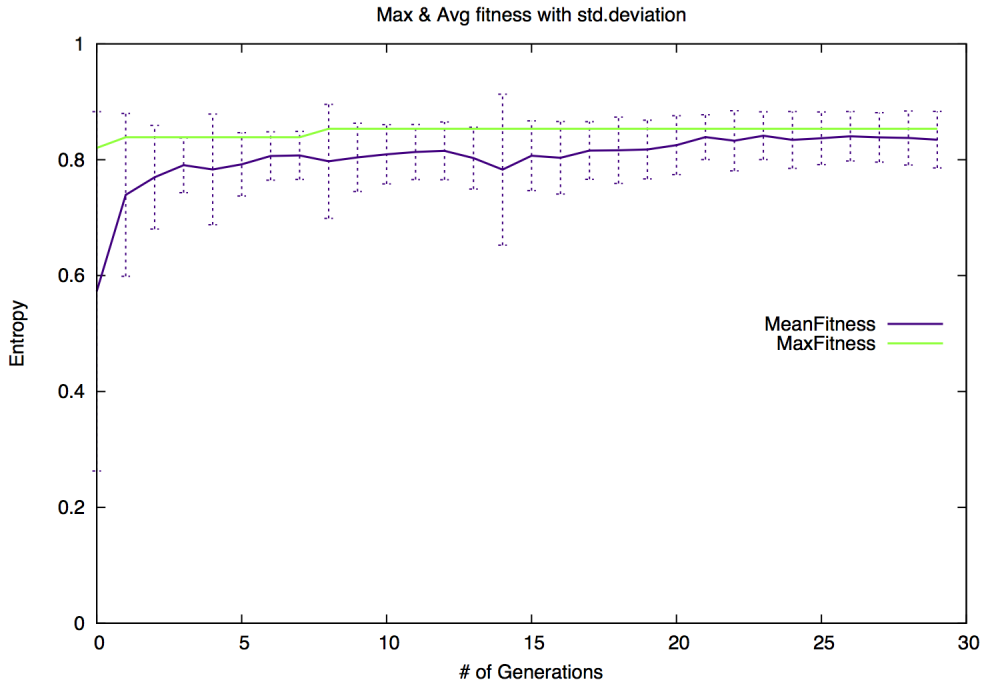
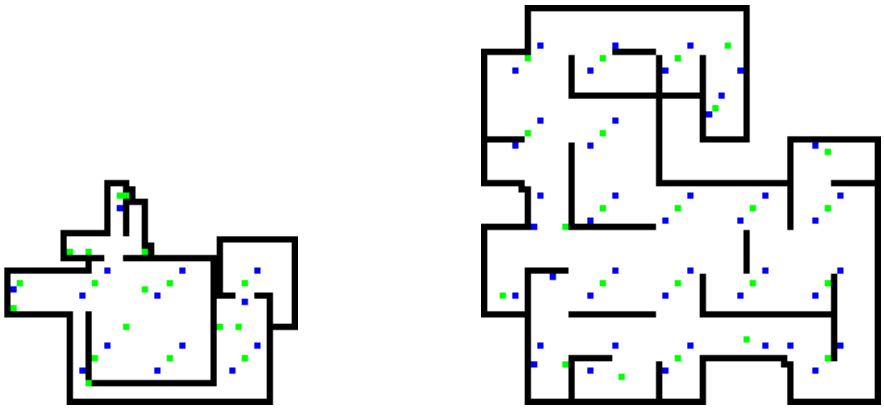


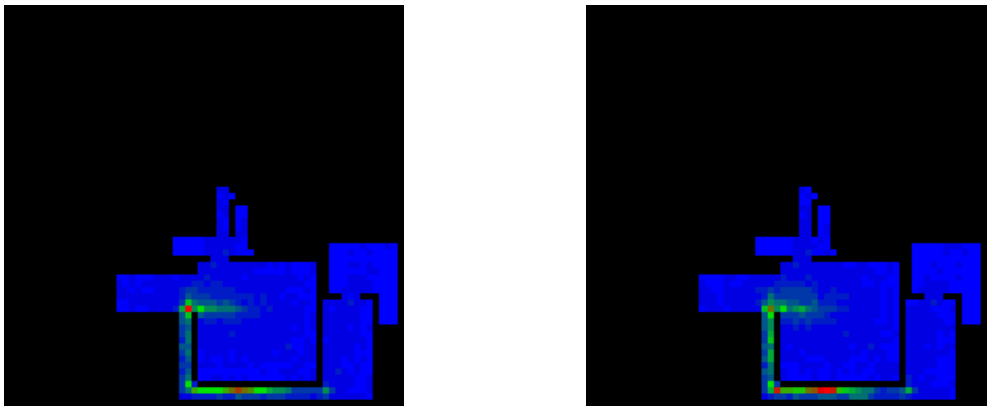

Figura 5.9: Fitness massima e media con rappresentazione *Grind* nelle generazioni, Arma *Rifle*, Abilità 35-80



(a) Mappa migliore con rappresentazione *All-Black*

(b) Mappa migliore con rappresentazione *Grind*

Figura 5.10: Mappe vincitrici dell'evoluzione con arma *Rifle*, abilità 35-80



(a) Heatmap uccisioni

(b) Heatmap morti

Figura 5.11: Heatmap delle uccisioni e delle morti della mappa migliore con rappresentazione *All-Black*, arma *Rifle*, abilità 35-80

lore di fitness più elevato ha fitness uguale a 0.90 ed è mostrata in figura 5.10a. La forma della mappa ottenuta è strana ma, considerando che l'abilità dei *bot* influenza l'efficacia della mira con l'arma (come affermato nella sezione 4.1), è possibile darle un'interpretazione. In una zona della mappa aperta, i *bot* sono liberi di muoversi su varie traiettorie, rendendoli difficili da mirare. Per un *bot* con abilità elevata questo è un vantaggio, in quanto prima che il *bot* con abilità minore (ovvero mira peggiore) riesca a colpirlo, il primo *bot* l'ha già colpito più volte. Considerando che, in questa sede, stiamo valutando l'uso dell'arma *rifle* che uccide con un colpo solo, le aree aperte della mappa portano ad avere una partita molto sbilanciata.

La mappa 5.10a presenta una sola area aperta e un lungo corridoio che la circonda. Essendo abbastanza stretto, il corridoio permette al *bot* con abilità minore di prendere la mira più velocemente (essendoci meno traiettorie che l'altro *bot* può intraprendere per attraversarlo), dandogli più possibilità di uccidere il giocatore con abilità maggiore. Questa ipotesi è confermata dalle figure 5.11a e 5.11b che mostrano le heatmap (rappresentazioni grafiche dei dati nella quale ogni punto è rappresentato da un colore che ne identifica il valore) rispettivamente delle uccisioni e delle morti ottenute facendo giocare i *bot* per 100 partite. Come è possibile vedere, i punti dove sono avvenute più uccisioni/morti (indicati in rosso) si trovano in prossimità del corridoio o al suo interno. Il colore rosso indica i punti dove sono avvenute più uccisioni, sfumando per il verde e terminando con il colore blu che indica i punti dove sono avvenute poche uccisioni (blu chiaro se non sono avvenute). Diversamente, utilizzando la rappresentazione *Grind*, si ha un lieve miglioramento con il passare delle generazioni (Figura 5.9). Il motivo è dovuto al fatto che questo tipo di rappresentazione non è idoneo alla generazione di mappe per questo tipo di arma. Le mappe create in questo modo, infatti, presentano molte aree in cui un *bot* può prendere alle spalle l'altro. Se il *bot* con abilità elevata prende alle spalle quello con abilità bassa, quest'ultimo non ha possibilità di sopravvivere. Nel caso in cui, invece, sia il *bot* con abilità inferiore a prendere alle spalle l'altro, la velocità di mira del secondo *bot* può comunque portare alla morte del primo. Le aree appena descritte sono infatti presenti nella mappa migliore (fitness uguale a 0.85) ottenuta dalla generazione con rappresentazione *Grind*, essa è mostrata in figura 5.10b.

### 5.2.4 Arma *Motosega*, Abilità bot 15-95

Dopo aver analizzato il bilanciamento del gioco nel quale i due *bot* con abilità differenti utilizzavano la stessa arma da distanza (il *rifle*), analizziamo ora

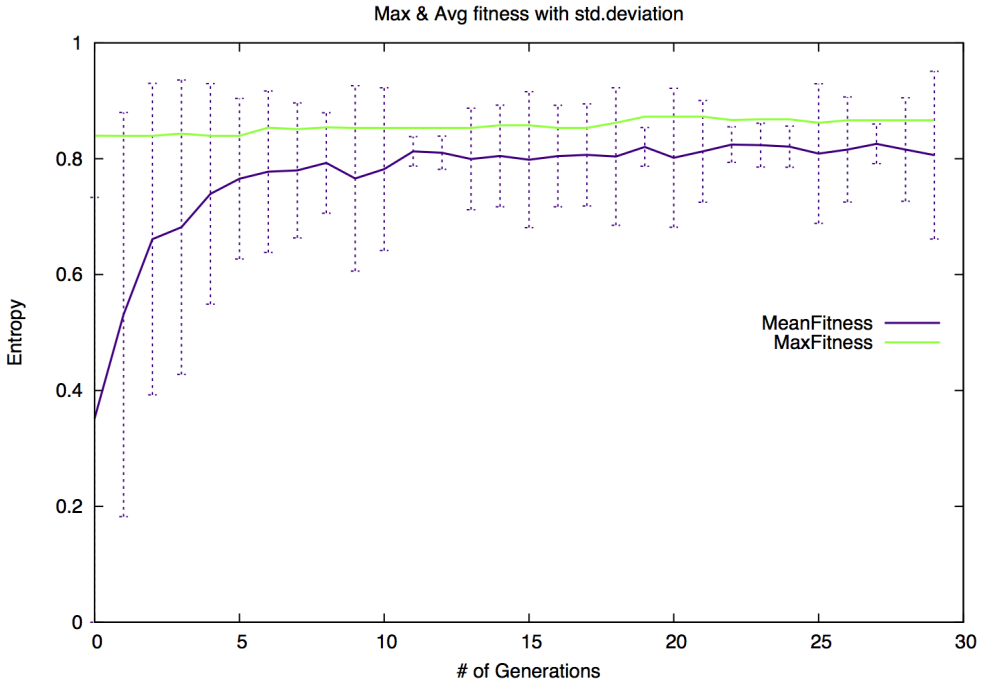



Figura 5.12: Fitness massima e media con rappresentazione *All-Black* nelle generazioni, Arma *Motosega*, Abilità 15-95

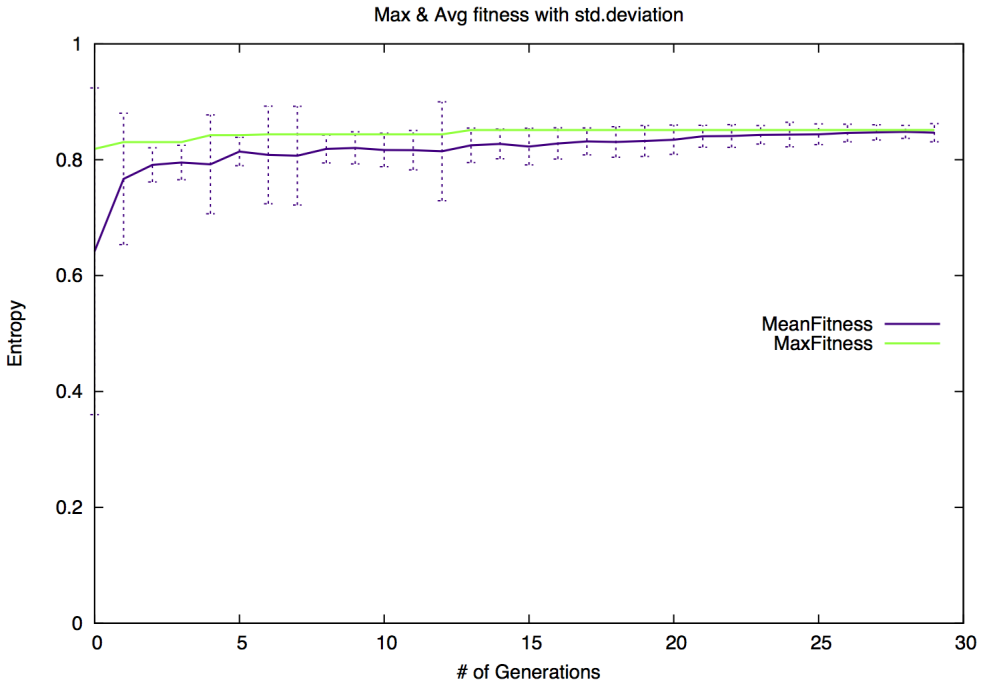
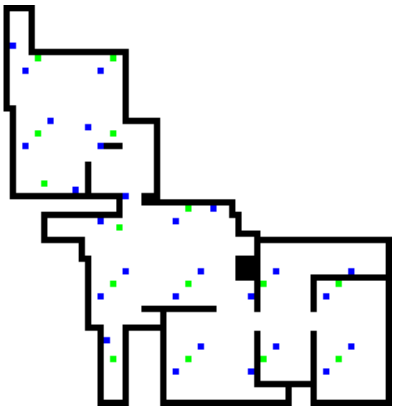
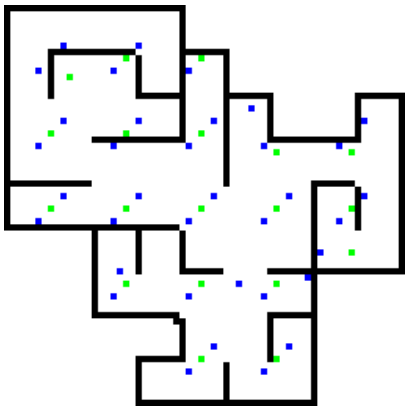



Figura 5.13: Fitness massima e media con rappresentazione *Grind* nelle generazioni, Arma *Motosega*, Abilità 15-95



(a) Mappa migliore con rappresentazione *All-Black*



(b) Mappa migliore con rappresentazione *Grind*

Figura 5.14: Mappe vincitrici dell'evoluzione con arma *Motosega*, abilità 15-95

cosa otteniamo se utilizziamo un'arma da ravvicinato. Come nel sottocapitolo precedente, dopo aver scelto di impostare l'arma uguale per entrambi i *bot* (in questo caso la *motosega*), dobbiamo scegliere che divario di abilità utilizzare per le nostre simulazioni all'interno mappe generate. Guardando la tabella 4.1, descritta nel capitolo precedente, il *Kill Ratio* (il rapporto tra le uccisioni del *bot* vincitore rispetto a quelle del *bot* perdente) non è mai superiore a 5. Le soglie di differenza di abilità tra i *bot* (vedi sezione 4.2) per quest'arma sono quindi solo due, quella in cui il ratio è accettabile e quella in cui è minore di 5. Escludendo la prima fascia, essendo il bilanciamento non necessario se il *Kill Ratio* è *accettabile* ($1 < \textit{Kill Ratio} < 2$), scegliamo un divario da utilizzare per le nostre simulazioni che sia maggiore di 65. Impostiamo quindi, per le nostre simulazioni, come arma la *motosega* e come abilità 15 per il primo *bot* e 95 per il secondo (*DivLvBot* pari a 80). Le figure 5.12 e 5.13 mostrano l'andamento del valore di fitness (medio e massimo) nelle generazioni. In entrambe le rappresentazioni si ha un lieve miglioramento dopo alcune generazioni, ma l'incremento rimane comunque basso e poco rilevante. Questi risultati sono da ricercarsi nella scarsa utilità con cui l'abilità del *bot* si ripercuote sull'uso della *motosega*. Influenzando solo la mira, l'abilità del *bot* non è realmente rilevante ai fini dell'uso di un'arma che non ha bisogno di mirare con precisione. Qualsiasi mappa creabile per bilanciare una situazione in cui due *bot* utilizzano entrambi la stessa arma da ravvicinato, finirà comunque nel far scontrare i *bot* a pochissima distanza l'uno dall'altro, rendendo realmente rilevante solo la velocità con cui si individuano i nemici. Considerando che, dati due *bot* che utilizzano entrambi un'arma da ravvicinato, il primo *bot* che attacca è il probabile vincitore dello scontro, esistono due modi per aumentare la probabilità di sopravvivenza

del *bot* con abilità minore. Il primo è creare una mappa piena di angoli in cui è possibile prendere alle spalle il nemico, questo è ciò che avviene utilizzando la rappresentazione *Grind* (figura 5.14b). Il secondo è creare una mappa ampia e senza muri che permetta al *bot* con abilità minore di individuare il nemico prima che quest'ultimo lo raggiunga, in modo che i due comincino ad attaccare insieme. La mappa migliore ottenuta utilizzando la rappresentazione *All-Black* (figura 5.14a) segue queste caratteristiche.

## 5.3 Bilanciamento per bot con abilità uguale ma arma diversa

Dopo aver bilanciato le partite al variare dell'abilità dei due *bot* mantenendo la stessa arma per entrambi, analizziamo ora cosa succede sbilanciando il gioco imponendo ai giocatori di combattere con armi diverse. Impostando quindi abilità 50 ad entrambi i *bot*, scegliamo di analizzare due possibili situazioni, *Rifle-Motosega* e *Rifle-Lanciagranate*. La prima situazione, in cui un *bot* è dotato di arma *rifle* e l'altro di *motosega*, ci consente di studiare cosa accade quando i due *bot* sono costretti ad adottare una strategia opposta durante la partita, imposta dall'arma usata. Infatti, il *bot* dotato di *rifle* si terrà a distanza mentre il *bot* con arma da ravvicinato si dovrà obbligatoriamente avvicinare per uccidere l'avversario prima che quest'ultimo lo colpisca (come precedente affermanto, il *rifle* uccide con un solo colpo). La seconda situazione, in cui un *bot* è dotato di arma *rifle* e l'altro di *lanciagranate*, ci consente di studiare cosa accade se un giocatore utilizza un'arma difficilmente controllabile. Il *lanciagranate*, infatti, spara proiettili esplosivi che rimbalzano più volte sul terreno prima di esplodere, rendendola un'arma difficile da controllare (non basta mirare il nemico e sparare) e a doppio taglio (è possibile che il *bot* che la usa si suicidi se spara il colpo troppo vicino).

### 5.3.1 Armi *Rifle-Motosega*, Abilità bot 50-50

Le figure 5.15 e 5.16 mostrano i risultati delle analisi in termini di fitness media e massima per le due rappresentazioni. Le simulazioni per la valutazione delle mappe candidate sono state effettuate con un *bot* dotato di arma *rifle* e l'altro di arma *motosega*, entrambi con livello di abilità pari a 50. Come è possibile vedere da queste due figure, il problema risulta troppo semplice in entrambi i casi. In entrambi, infatti, si ha un incremento impercettibile dell'entropia (nella scala dei millesimi, da 0.99 a 1) che risulta molto elevata già alla prima generazione, indice del fatto che qualsiasi mappa risulterebbe bilanciata per

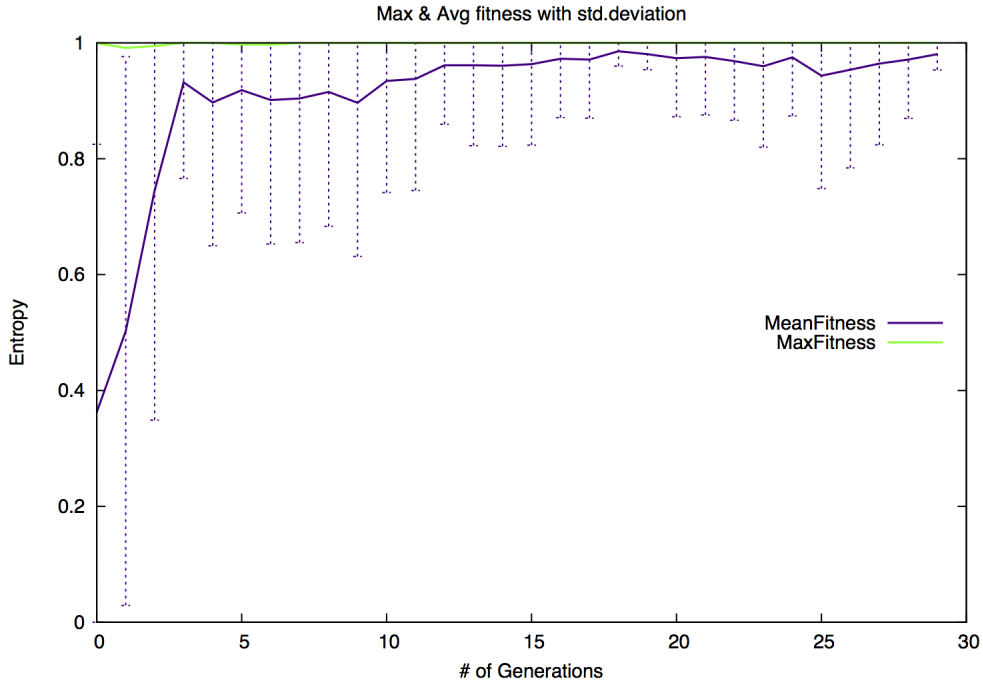



Figura 5.15: Fitness massima e media con rappresentazione *All-Black* nelle generazioni, Armi *Rifle-Motosega*, Abilità 50-50

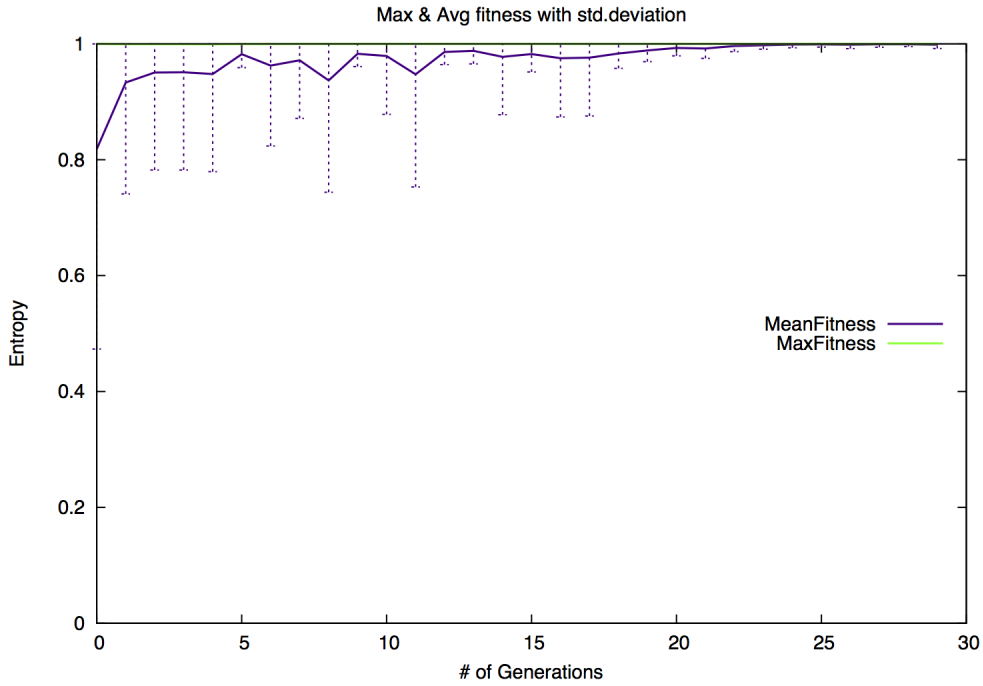
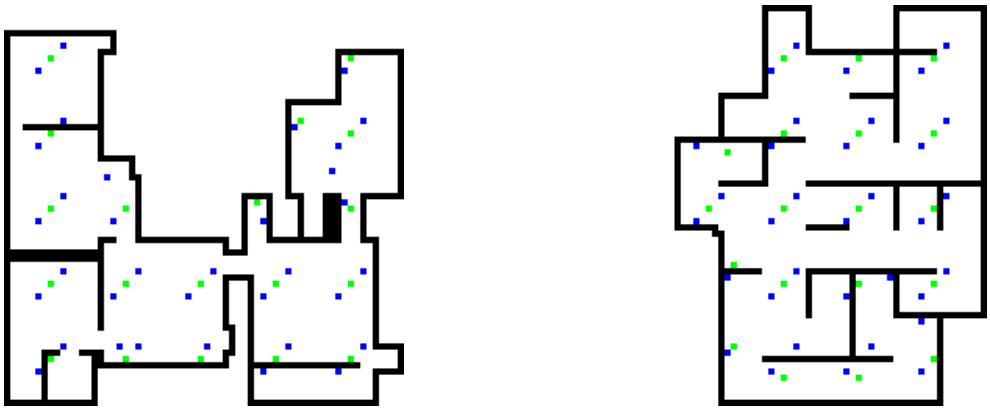



Figura 5.16: Fitness massima e media con rappresentazione *Grind* nelle generazioni, Armi *Rifle-Motosega*, Abilità 50-50



(a) Mappa migliore con rappresentazione *All-Black*

(b) Mappa migliore con rappresentazione *Grind*

Figura 5.17: Mappe vincitrici dell'evoluzione con armi *Rifle-Motosega*, abilità 50-50

questa configurazione. Da questi risultati potremmo dedurre che le due armi sono effettivamente bilanciate tra loro, ma basti pensare al fatto che l'arma *rifle* uccide con un colpo solo per rendersi conto che questa deduzione risulterebbe in realtà errata. Analizzando i *bot* di *Cube 2* abbiamo scoperto che il problema risiede nella strategia che la loro IA implementa. La strategia utilizzata, infatti, è quella di andare incontro all'avversario quando lo si vede, indipendentemente dall'arma che si sta utilizzando. Questa strategia impatta negativamente su armi efficaci a distanza (come il *rifle*) rendendo più facile, per un *bot* con arma da ravvicinato, avvicinarsi al bersaglio e colpirlo. Le figure 5.17a e 5.17b mostrano le mappe con entropia massima rispettivamente delle rappresentazioni *All-Black* e *Grind*.

### 5.3.2 Armi *Rifle-Lanciagranate*, Abilità bot 50-50

Dotiamo ora un *bot* con arma *rifle* e all'altro con *lanciagranate* e manteniamo il livello di abilità pari a 50 per entrambi. Prima di effettuare le nostre analisi abbiamo però bisogno di cambiare la funzione obiettivo in quanto, per un'arma che può provocare suicidi, una funzione che massimizza l'entropia delle uccisioni potrebbe non essere la scelta ottimale. Massimizzare le uccisioni con un'arma che provoca suicidi potrebbe voler dire morire molte più volte di quante si uccide l'avversario, non esattamente ciò che ci si aspetterebbe in una partita reale. Per ovviare a questo problema utilizziamo come funzione obiettivo l'entropia delle morti, ovvero la formula dell'entropia applicata al numero di morti dei *bot* subite

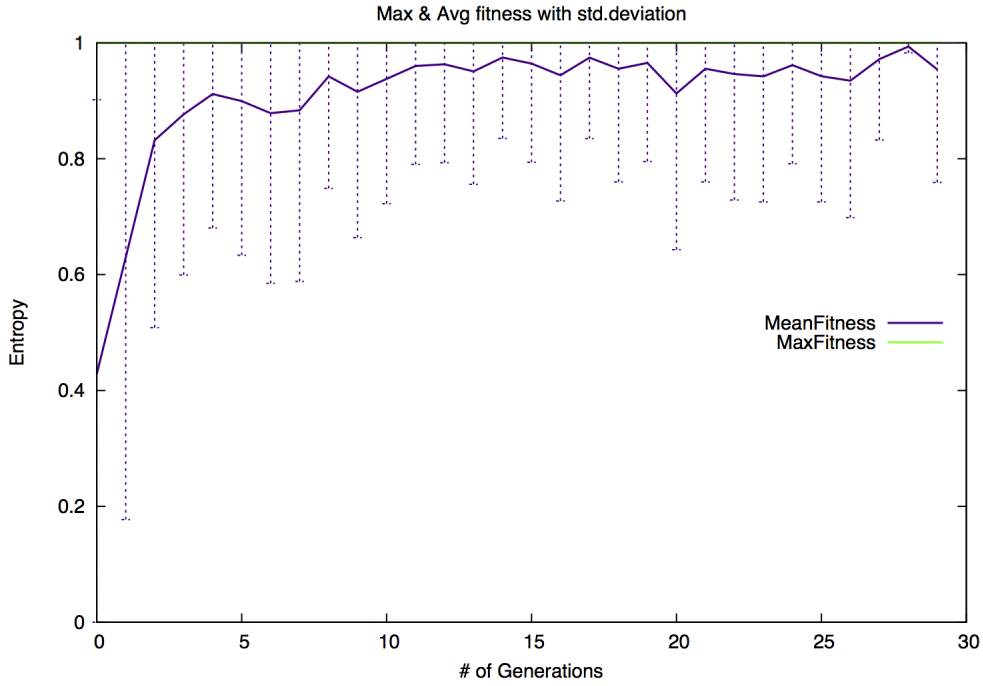



Figura 5.18: Fitness massima e media con rappresentazione *All-Black* nelle generazioni, Armi *Rifle-Lanciagranate*, Abilità 50-50

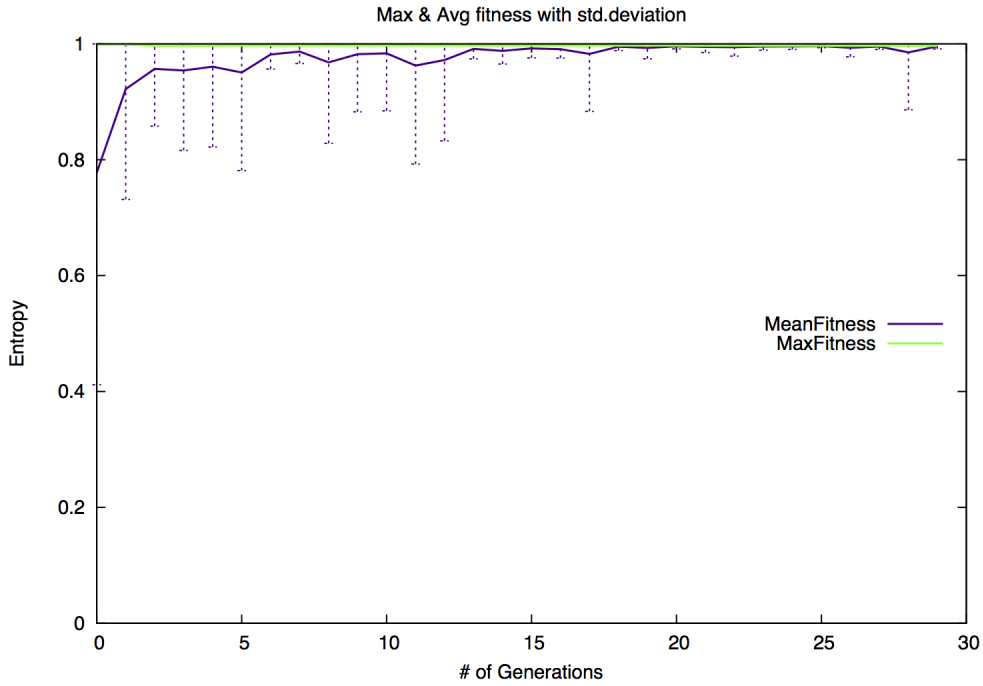
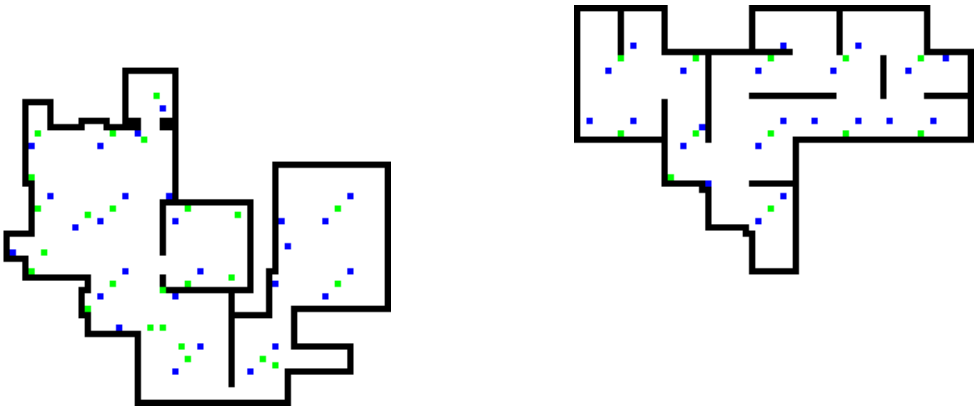



Figura 5.19: Fitness massima e media con rappresentazione *Grind* nelle generazioni, Armi *Rifle-Lanciagranate*, Abilità 50-50



(a) Mappa migliore con rappresentazione *All-Black*

(b) Mappa migliore con rappresentazione *Grind*

Figura 5.20: Mappe vincitrici dell'evoluzione con armi *Rifle-Lanciagranate*, Abilità 50-50

durante la partita:

$$f = -\frac{D_{bot1}}{D_{bot1} + D_{bot2}} \log_2 \frac{D_{bot1}}{D_{bot1} + D_{bot2}} - \frac{D_{bot2}}{D_{bot1} + D_{bot2}} \log_2 \frac{D_{bot2}}{D_{bot1} + D_{bot2}}$$

in questo modo si cercherà implicitamente di minimizzare i suicidi del *bot* che utilizza l'arma *lanciagranate* massimizzando le sue uccisioni. Nonostante l'utilizzo di questa nuova funzione obiettivo, i risultati non cambiano rispetto a quelli ottenuti predecedentemente. Le figure 5.18 e 5.19 mostrano infatti come il problema sia troppo semplice, con un incremento impercettibile del valore di fitness nelle generazioni se usata la rappresentazione *All-Black* (anche in questo caso nell'ordine dei millesimi). La causa è la medesima di quella riscontrata nel capitolo precedente, la strategia adottata dai *bot*. Avvicinandosi sempre, il *bot* che utilizza l'arma *rifle* avvantaggia quello che utilizza l'arma *lanciagranate*, rendendogli più facile colpirlo nonostante i rimbalzi dei proiettili rendano difficile colpire con quest'ultima arma. Inoltre, il solo mirare il bersaglio da parte del *bot* che utilizza l'arma *lanciagranate* non è sufficiente ad utilizzare quest'ultima efficientemente, portando il *bot* a non sfruttare pienamente le potenzialità di questa arma. Le figure 5.20a e 5.20b mostrano le mappe con entropia delle uccisioni massima rispettivamente delle rappresentazioni *All-Black* e *Grind*.

## 5.4 Sommario

In questo capitolo abbiamo analizzato i risultati ottenuti dal bilanciamento di partite tramite mappe generate con generazione procedurale di contenuti e algoritmi genetici. Inizialmente (Sezione 5.1) abbiamo motivato la scelta delle

rappresentazioni usate, introdotto l'operatore di crossover utilizzato dal nostro algoritmo genetico e riportato la funzione obiettivo che volevamo far massimizzare all'algoritmo. Successivamente, nella Sezione 5.2, abbiamo cominciato l'analisi dei nostri esperimenti impostando ai *bot* la stessa abilità e la stessa arma, prima il *rifle*(sottosezione 5.2.1) e poi la *motosega* (sottosezione 5.2.2). Successivamente abbiamo analizzato se era possibile bilanciare lo scompenso dovuto al divario di abilità tra i *bot* data la stessa arma, prima per il *rifle*(sottosezione 5.2.3) e poi per la *motosega* (sottosezione 5.2.4). Infine (Sezione 5.3) abbiamo dato in dotazione ai *bot* diverse armi, mantenendo lo stesso livello di abilità e analizzandone gli effetti. Prima dotando un *bot* con arma *rifle* e l'altro con arma *motosega* (sottosezione 5.3.1), successivamente dotando il primo di arma *rifle* e l'altro di *lanciagranate* (sottosezione 5.3.2). La tabella riassuntiva 5.1 mostra sinteticamente i risultati ottenuti, valorizzandoli in base alla rappresentazione usata (*All-Black* o *Grind*) e alle varie condizioni iniziali da noi impostate (abilità *bot* e arma usata).

| Impostazioni iniziali | | | Rappresentazione | |
|---|---|---|---|---|
| | | | *All-Black* | *Grind* |
| Abilità uguale | Arma uguale | Rifle | troppo semplice | troppo semplice |
| | | Motosega | troppo semplice | troppo semplice |
| | Arma diversa | Rifle-Motosega | miglioramento impercettibile | miglioramento impercettibile |
| | | Rifle-Lanciagranate | miglioramento impercettibile | troppo semplice |
| Divario di Abilità | Arma uguale | Rifle | miglioramento costante | miglioramento lieve |
| | | Motosega | miglioramento lieve | miglioramento lieve |

Tabella 5.1: Risultati ottenuti sul problema del bilanciamento negli esperimenti effettuati nel capitolo 5

# Capitolo 6

# Esperimenti su più parametri

Nel capitolo precedente abbiamo analizzato i risultati dei nostri esperimenti sul bilanciamento del gioco al variare di singoli parametri. Le partite erano quindi simulate da *bot* dotati di arma diversa oppure da *bot* ai quali abbiamo impostato livelli di abilità differenti. In questo capitolo analizziamo invece i risultati dei nostri esperimenti sul bilanciamento del gioco ottenuti variando entrambi i parametri, utilizzando quindi *bot* con livelli di abilità differenti tra loro e, inoltre, dotati di arma diversa. Inizialmente mostriamo i risultati ottenuti utilizzando un *bot* dotato di arma *rifle* e l'altro dotato di *motosega*, mettendo in evidenza sia il caso in cui il primo *bot* abbia abilità più alta rispetto al secondo che il caso inverso. Successivamente mostriamo come cambiano i risultati ottenuti se si utilizza un *bot* dotato di arma *rifle* e l'altro dotato di *lanciagranate* (analizzando ancora una volta i due casi di abilità differente).

## 6.1 Arma *Rifle-Motosega*, abilità differenti

Il primo insieme di esperimenti è stato condotto sulla generazione di mappe sia con rappresentazione *All-Black* che con *Grind*, come avvenuto nel capitolo precedente. La valutazione delle mappe candidate è ottenuta tramite la simulazione di una partita tra due *bot*, uno di essi dotato di arma *rifle* e l'altro di *motosega*. Ai fini di imporre una situazione di svantaggio tra i due *bot* (in aggiunta a quella data dalla differente arma in dotazione), impostiamo il livello di abilità di un *bot* a 80 e il livello di abilità dell'altro a 20. In questa sede, quindi, effettuiamo due volte gli esperimenti, uno nel quale il *bot* con arma *rifle* è avvantaggiato (abilità uguale a 80) rispetto a quello con la *motosega* (abilità uguale a 20) e l'altro nel quale si ha la situazione inversa (*bot* con arma *motosega* abilità uguale a 80, *bot* con arma *rifle* abilità uguale a 20). L'algoritmo che utilizziamo per la ricerca della soluzione ottima nello spazio di ricerca è lo stesso utilizzato nel capito-

lo precedente, ovvero un algoritmo genetico con *matrix crossover*, *mutazione semplice*, *tournament selection a 2 candidati* e funzione di fitness che mira a massimizzare l'entropia delle uccisioni tra i due *bot*. Allo stesso modo di quanto avveniva per gli esperimenti descritti nel capitolo precedente, tutte le risorse (oggetti ed armi) presenti sulla mappa sono disattivati in modo da ottenere dati riconducibili al solo svantaggio, imposto da noi inizialmente, di un *bot* rispetto all'altro.

### 6.1.1 Armi *Rifle-Motosega*, Abilità bot 80-20

Utilizzando questa configurazione, il valore di fitness è calcolato utilizzando i dati ottenuti dalla simulazione di una partita nella quale il *bot* con arma *rifle* è avvantaggiato rispetto a quello con arma *motosega*. Considerando che un giocatore che utilizza un'arma a distanza che uccide in un solo colpo (*rifle*) è avvantaggiato rispetto ad uno che ne utilizza una da ravvicinato (*motosega*), la configurazione che abbiamo impostato avvantaggia il giocatore che utilizza l'arma più potente, sbilanciando di molto la partita. La limitazione dovuta all'implementazione dell'IA in Cube 2, ovvero la strategia di combattimento che porta il *bot* ad andare incontro al nemico non appena viene visto indipendentemente dall'arma in uso (situazione descritta nella sottosezione 5.3.1), annulla il vantaggio dato dall'uso di un'arma a distanza rispetto ad una da ravvicinato. Il fattore che realmente sbilancia la partita in questa configurazione risulta quindi essere solamente la differenza di abilità tra i due *bot*. La figura 6.1 mostra l'andamento della fitness media e massima per l'evoluzione con l'utilizzo della rappresentazione *All-Black*. Come è possibile osservare, si ha un lieve miglioramento (alle generazioni 1, 5 e 9) del valore massimo di fitness che risulta comunque abbastanza elevato (circa 0.94) fin dalla prima generazione. In questa configurazione, il *bot* con arma *rifle* (quello avvantaggiato) mira in modo veloce e preciso dando poche possibilità all'altro *bot* di effettuare uccisioni. L'unico modo che ha il *bot* svantaggiato di uccidere l'avversario è incontrarlo dietro un angolo, in modo da avere l'opportunita di ucciderlo prima che l'avversario riesca a sparargli. La figura 6.2a mostra come la mappa con fitness più elevata alla fine dell'evoluzione con rappresentazione *All-Black* sia una mappa composta da un numero elevato di ampie stanze e di corridoi stretti. Per individuare le zone della mappa dove effettivamente avvengono le uccisioni (avendo molte zone eterogenee nella mappa ottenuta) simuliamo 100 partite, mantenendo la configurazione dei *bot* come sopra, raccogliendo dati sulla posizione in cui avvengono le uccisioni effettuate dai *bot* e generando con essi una heatmap (rappresentazione grafica dei dati nella quale ogni punto è rappresentato da un colore che

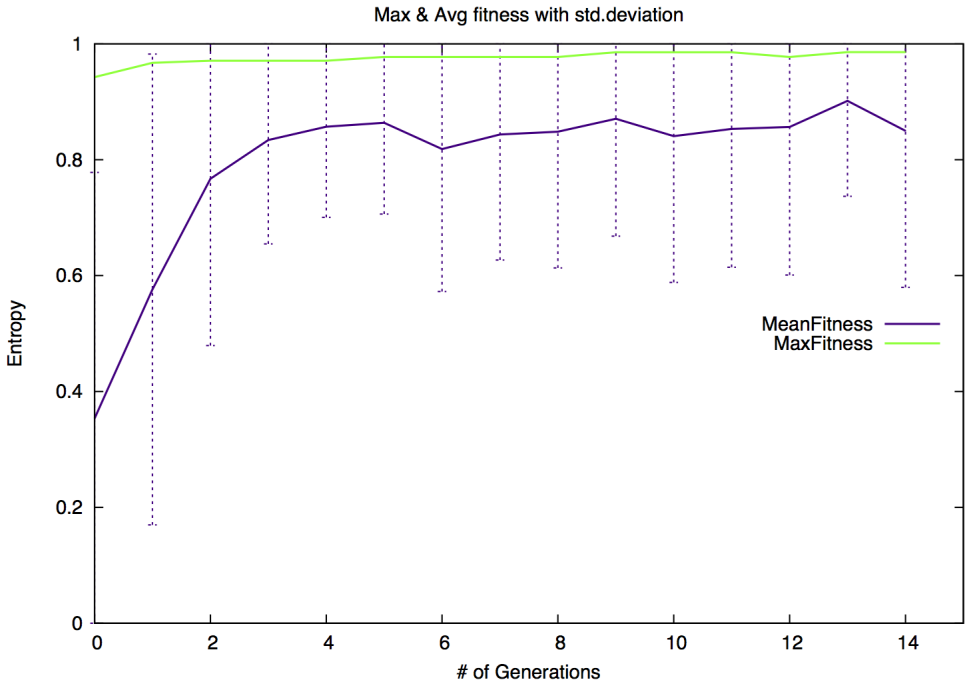
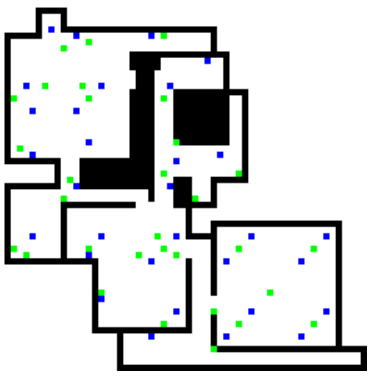
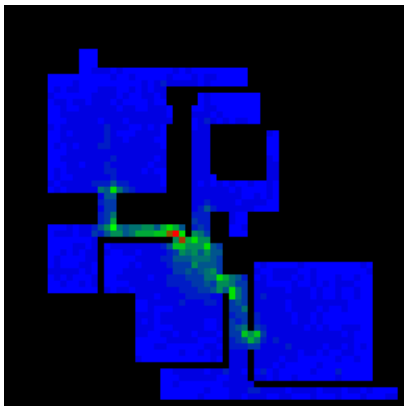



Figura 6.1: Fitness massima e media con rappresentazione *All-Black* nelle generazioni, armi *Rifle-Motosega*, Abilità 80-20



(a) Mappa migliore con rappresentazione *All-Black*



(b) Heatmap rappresentazione *All-Black*

Figura 6.2: Mappa vincitrice dell'evoluzione con rappresentazione *All-Black*, Armi *Rifle-Motosega*, Abilità 80-20

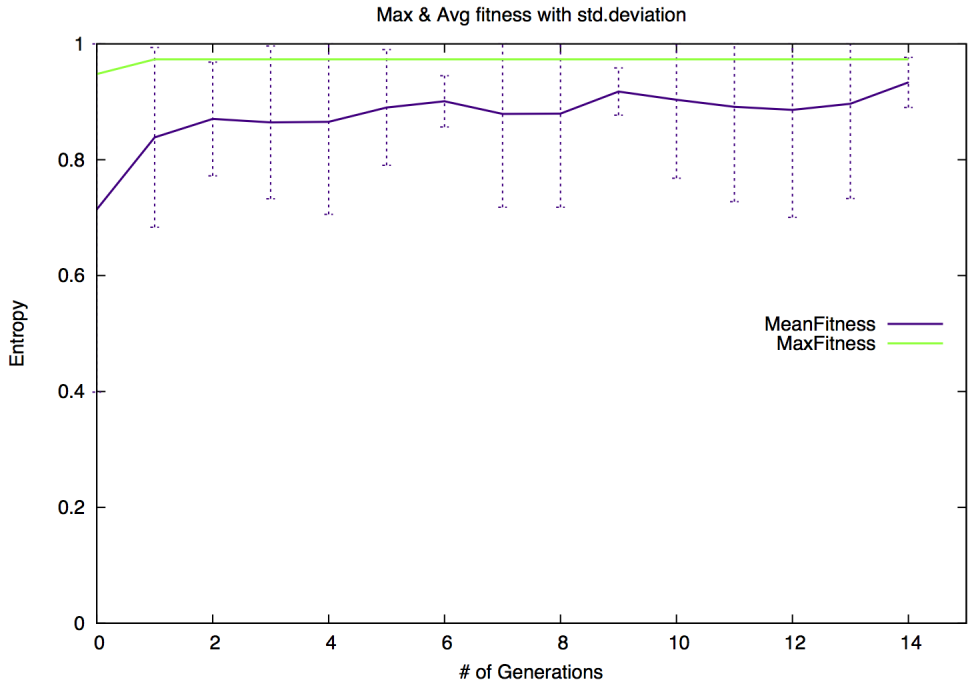
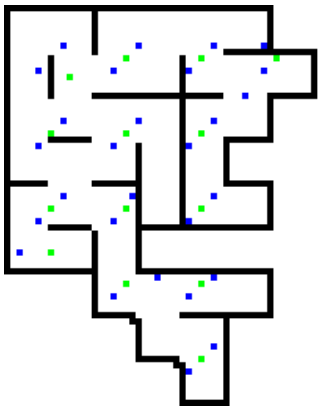
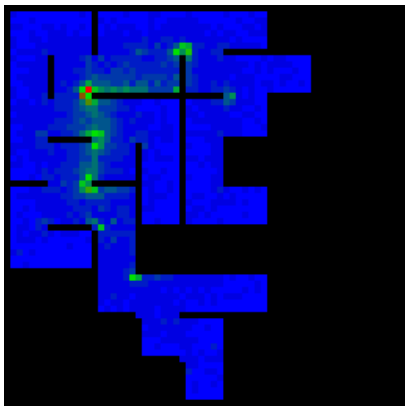



Figura 6.3: Fitness massima e media con rappresentazione *Grind* nelle generazioni, Armi *Rifle-Motosega*, Abilità 80-20



(a) Mappa migliore con rappresentazione *Grind*



(b) Heatmap rappresentazione *Grind*

Figura 6.4: Mappa vincitrice dell'evoluzione con rappresentazione *Grind*, Armi *Rifle-Motosega*, abilità 80-20

ne identifica il valore). Quest'ultima è mostrata in figura 6.2b e mette in evidenza come la maggior parte delle uccisioni avviene nei corridoi stretti e, nello specifico, negli angoli di uscita di alcuni corridoi (si ricorda che, nella heatmap, partendo dal colore blu chiaro, che indica un punto nel quale non sono avvenute uccisioni, si sfuma verso il verde per poi raggiungere il rosso chiaro, che indica un punto nel quale le uccisioni sono massime). La figura 6.2b conferma quindi la nostra ipotesi: in questa configurazione il *bot* con la *motosega* riesce ad effettuare il maggior numero di uccisioni quando incontra il *bot* avversario dietro un angolo, uccidendolo prima che quest'ultimo abbia il tempo di sparare. La figura 6.3 mostra invece l'andamento della fitness per l'evoluzione con l'utilizzo della rappresentazione *Grind*. Diversamente dal caso precedente la fitness massima si stabilizza subito dopo un breve incremento iniziale, facendoci dedurre che il problema risulta troppo semplice. Come detto in precedenza, il *bot* con arma *motosega* ha possibilità di uccidere il *bot* avversario se la mappa presenta molti angoli dove quest'ultimo può essere preso alla sprovvista o alle spalle, dando modo al *bot* con la *motosega* di uccidere. Per il modo in cui la rappresentazione *Grind* genera le mappe, queste ultime sono propense ad essere piene di angoli e posti nei quali i *bot* si incontrano a distanza ravvicinata. Questo è il motivo per cui il problema risulta troppo semplice con questa rappresentazione, un numero elevato di mappe presenti nello spazio di ricerca sono ottime per il nostro scopo. Ciò non avviene invece con la rappresentazione *All-Black*, che non è pensata per generare mappe piene di angoli ma mappe con grosse stanze e corridoi lunghi, rendendo la ricerca di una mappa che avvantaggi il giocatore con arma *motosega* più difficile. La mappa con fitness più elevata alla fine dell'evoluzione con rappresentazione *Grind* (figura 6.4a) è infatti piena di angoli e zone in cui un *bot* può essere preso alle spalle dall'avversario. Inoltre, a conferma di ciò che abbiamo affermato fino ad ora, la figura 6.4b (la heatmap delle uccisioni di quest'ultima mappa) mostra che il maggior numero di uccisioni avviene in prossimità degli angoli.

### 6.1.2 Armi *Rifle-Motosega*, Abilità bot 20-80

Invertendo i valori delle abilità, il *bot* con arma *motosega* risulta ora avvantaggiato (abilità 80) rispetto a quello con arma *rifle* (abilità 20). Considerando che, con questa configurazione, il *bot* con arma *rifle* è poco preciso (e lento nella mira) e che, come precedentemente affermato (sottosezione 5.3.1), i *bot* vanno incontro al nemico appena lo vedono indipendentemente dall'arma equipaggiata, le mappe che dovremo ottenere per bilanciare il divario di abilità dovranno essere formate da ampie stanze o lunghi corridoi. Una mappa con queste carat-

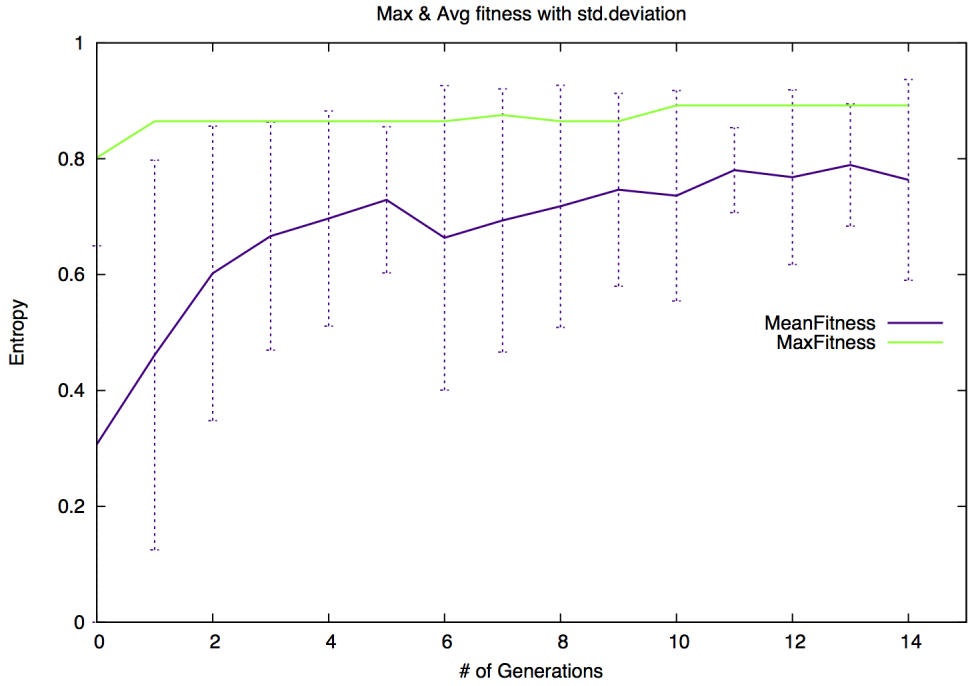



Figura 6.5: Fitness massima e media con rappresentazione *All-Black* nelle generazioni, Armi *Rifle-Motosega*, Abilità 20-80

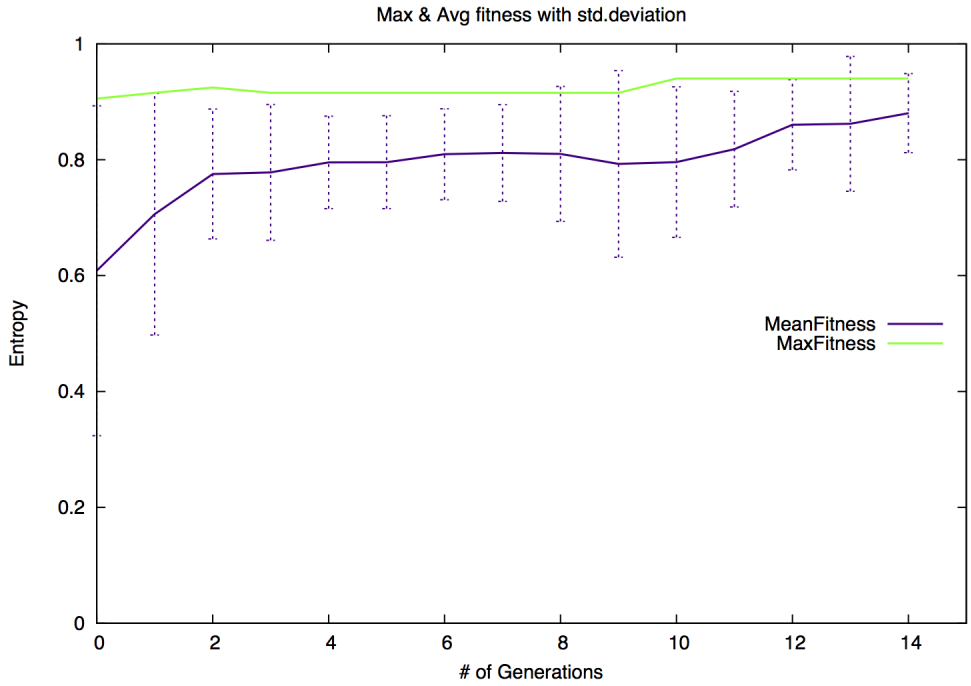
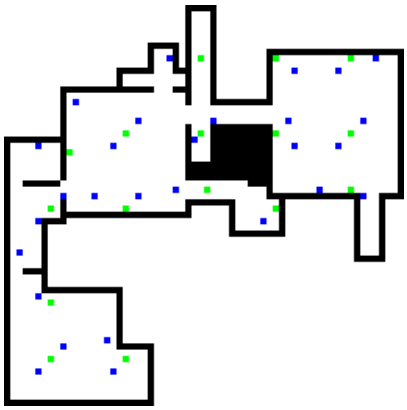
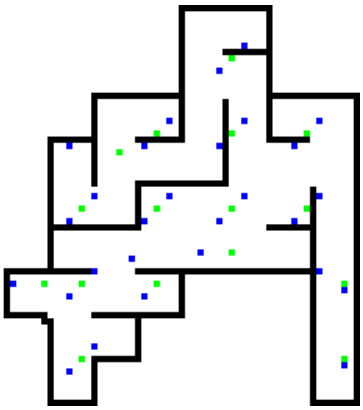



Figura 6.6: Fitness massima e media con rappresentazione *Grind* nelle generazioni, Armi *Rifle-Motosega*, Abilità 20-80



(a) Mappa migliore con rappresentazione *All-Black*



(b) Mappa migliore con rappresentazione *Grind*

Figura 6.7: Mappe vincitrici dell'evoluzione con armi *Rifle-Motosega*, Abilità 20-80

teristiche permetterebbe al *bot* con arma *rifle* di avere il tempo di prendere la mira correttamente e sparare un colpo che vada a segno, uccidendo l'avversario prima che quest'ultimo si avvicini troppo e lo uccida. Le figure 6.5 e 6.6 mostrano i dati riguardanti le evoluzioni rispettivamente con rappresentazione *All-Black* e *Grind*, in entrambi i casi possiamo vedere un miglioramento lieve del valore di fitness massimo. Mettendo a confronto i grafici 6.1, 6.5 e 6.3, 6.6 ci accorgiamo che, per entrambe le rappresentazioni, il valore di fitness massimo (ovvero il maggior bilanciamento) è raggiunto quando il livello di abilità più elevato è impostato al *bot* che utilizza l'arma *rifle* (figure 6.1 6.3). Questo risultato è paradossale, in quanto il bilanciamento di una situazione in cui il vantaggio, in termini di abilità, è dato al giocatore con l'arma più potente dovrebbe essere più difficile rispetto al bilanciamento di una situazione in cui l'abilità impostata favorisce il giocatore con l'arma meno potente. Questa situazione è dovuta al fatto che, per come l'IA dei *bot* in Cube 2 è stata realizzata, i *bot* attaccano l'avversario andandogli incontro appena lo vedono, indipendentemente dall'arma da loro utilizzata. Questa limitazione è stata precedentemente discussa in 5.3.1. Le figure 6.7a e 6.7b mostrano le mappe con fitness più elevata alla fine dell'evoluzione, la prima con rappresentazione *All-Black* mentre la seconda con rappresentazione *Grind*. La prima mappa presenta zone molto ampie e corridoi molto lunghi mentre la seconda, rispetto alle mappe che la rappresentazione *Grind* può generare, ha pochi angoli ed è composta da un corridoio diagonale che taglia tutta la mappa, dall'angolo in basso a sinistra a quello in alto a destra. Questi elementi permettono, come descritto in precedenza, di avvantaggiare il *bot* che utilizza l'arma *rifle* dandogli il tempo di prendere

la mira prima che il *bot* con la *motosega* lo raggiunga.

## 6.2 Arma *Rifle-Lanciagranate*, abilità differenti

Mantenendo il *rifle* come arma in dotazione al primo *bot*, dotiamo ora il secondo *bot* di *lanciagranate* ed effettuiamo gli stessi esperimenti svolti nella sezione precedente, utilizzando questa configurazione per la simulazione delle partite. Per la generazione di mappe utilizziamo nuovamente le rappresentazioni *All-Black* e *Grind* e analizziamo, per ognuna di esse, sia il caso in cui il *bot* con arma *rifle* sia avvantaggiato (*bot* con arma *rifle* abilità uguale a 80, *bot* con arma *lanciagranate* abilità uguale a 20) che quello in cui il *bot* avvantaggiato sia quello con arma *lanciagranate* (*bot* con arma *rifle* abilità uguale a 20, *bot* con arma *lanciagranate* abilità uguale a 80). Mantenendo lo stesso algoritmo di ricerca (algoritmo genetico con *matrix crossover*, *mutazione semplice*, *tournament selection a 2 candidati*) per la ricerca del candidato migliore, utilizziamo però l'entropia delle morti dei *bot* come funzione obiettivo (funzione introdotta nella sottosezione 5.3.2) in modo da evitare di massimizzare i suicidi del *bot* dotato di arma *lanciagranate* piuttosto che le uccisioni da lui effettuate. Come in tutti gli esperimenti svolti in precedenza, tutte le risorse (oggetti ed armi) presenti sulla mappa sono disattivate ai fini di focalizzare la ricerca su una mappa che bilanci la sola situazione iniziale da noi imposta.

### 6.2.1 Armi *Rifle-Lanciagranate*, Abilità bot 80-20

Impostando questi valori di abilità ai *bot* otteniamo un duplice effetto, rendiamo reattivo e preciso il *bot* con arma *rifle* (abilità uguale a 80) e abbassiamo drasticamente le possibilità che ha il *bot* con il *lanciagranate* di uccidere l'avversario. Quest'ultimo effetto è dovuto al fatto che il *lanciagranate* è un'arma molto difficile da utilizzare in modo efficiente, infatti non basta mirare il bersaglio e sparare per riuscire ad ucciderlo, ma è necessario calcolare correttamente la traiettoria (dovuta ai rimbalzi) che il proiettile seguirà prima di esplodere. Le figure 6.8 e 6.9 mostrano l'andamento della funzione di fitness media e massima durante le generazioni, rispettivamente ottenute con rappresentazione *All-Black* per la prima e *Grind* per la seconda. Come è possibile osservare, si ha un lieve incremento del valore di fitness massimo in entrambe le rappresentazioni raggiungendo quasi il valore massimo (entropia uguale a 0.99) se si utilizza la rappresentazione *Grind*. Analizzando le figure 6.10a e 6.10b, che rappresentano le mappe con il valore di fitness maggiore risultanti dalle due generazioni (la prima per rappresentazione *All-Black* e la seconda per *Grind*), possiamo notare come in entrambi i casi (soprattutto nella mappa generata con rappresentazione

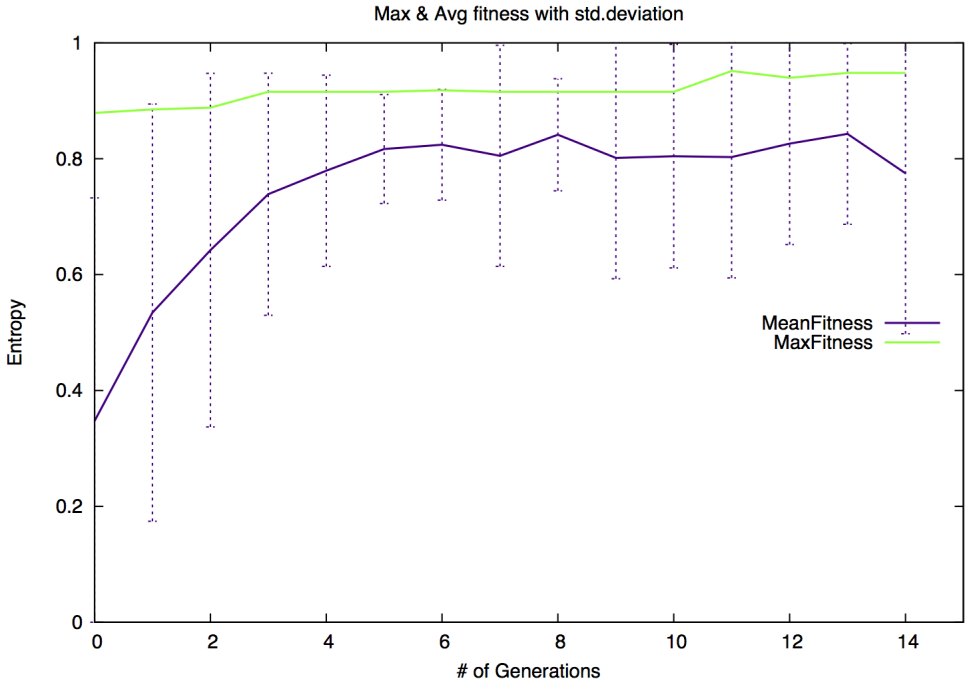



Figura 6.8: Fitness massima e media con rappresentazione *All-Black* nelle generazioni, Armi *Rifle-Lanciagranate*, Abilità 80-20

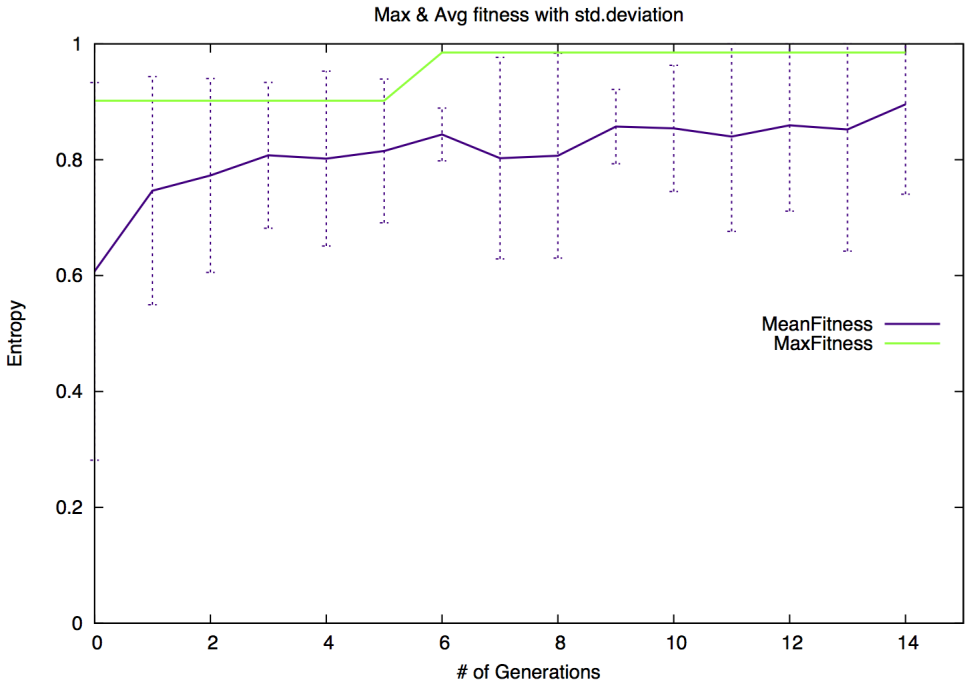
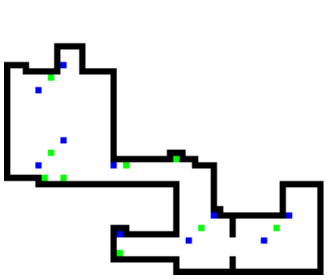
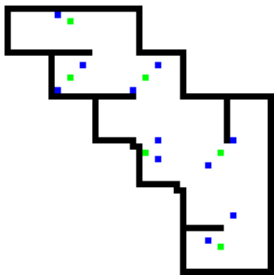



Figura 6.9: Fitness massima e media con rappresentazione *Grind* nelle generazioni, Armi *Rifle-Lanciagranate*, Abilità 80-20

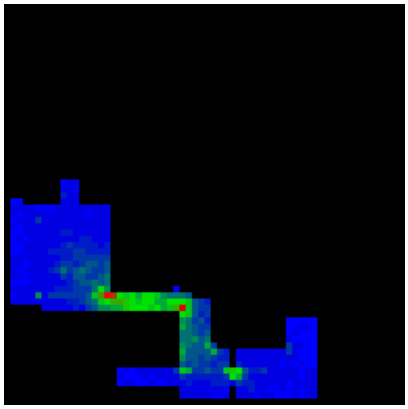


(a) Mappa migliore con rappresentazione *All-Black*

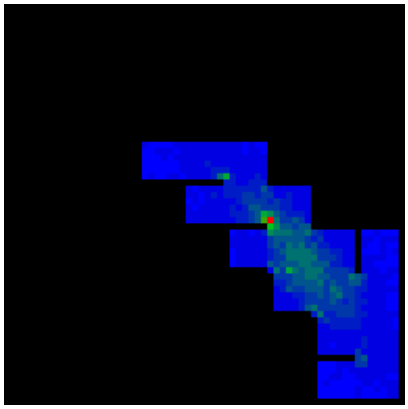


(b) Mappa migliore con rappresentazione *Grind*

Figura 6.10: Mappe vincitrici dell'evoluzione con armi *Rifle-Lanciagranate*, Abilità 80-20



(a) Heatmap rappresentazione *All-Black*



(b) Heatmap rappresentazione *Grind*

Figura 6.11: Heatmap delle uccisioni per le mappe vincitrici dell'evoluzione con armi *Rifle-Lanciagranate*, Abilità 80-20

*Grind*) la mappa generata è di dimensioni contenute, formata da corridoi stretti e stanze piccole (queste ultime nel caso della rappresentazione *All-Black*). Il motivo è da ricercarsi nel fatto che, per come l'IA di Cube 2 è implementata, i *bot* sparano ogni volta che vedono un avversario, cercando di mirarlo nel miglior modo possibile e continuando a sparare alla massima velocità permessa dall'arma. Utilizzando come arma il *lanciagranate* e avendo abilità bassa, il *bot* spara in modo poco preciso non appena vede l'avversario, affidandosi al rimbalzo fortuito del proiettile per uccidere l'avversario. E' facilmente intuibile come, in una mappa con corridoi stretti e stanze piccole, la probabilità che un proiettile sparato mirando erroneamente rimbalzi colpendo l'avversario è elevata, consentendo al *bot* con abilità bassa di accumulare qualche uccisione in più. Questo fenomeno è facilmente osservabile, soprattutto per la rappresentazione *All-Black*, dall'analisi delle heatmap per le due mappe vincitrici dell'evoluzione, mostrate in figura 6.11a (*All-Black*) e 6.11b (*Grind*).

### 6.2.2 Armi *Rifle-Lanciagranate*, Abilità bot 20-80

Invertendo i livelli di abilità dei due *bot* otteniamo una situazione nella quale il *bot* con arma *rifle* è poco preciso (abilità uguale a 20) mentre il *bot* con arma *lanciagranate* è preciso (abilità uguale a 80) ma non efficiente nell'uso dell'arma in quanto, per un uso efficiente del *lanciagranate*, è necessario tenere in considerazione la traiettoria che il proiettile seguirà prima che esploda, operazione non effettuata dall'IA dei *bot* di Cube 2. Le figure 6.12 e 6.13 mostrano i risultati in termini di entropia media e massima delle due evoluzioni, rispettivamente con rappresentazione *All-Black* e *Grind*. Possiamo notare come, fin da subito, viene raggiunto il valore massimo di entropia (pari a 1) in entrambi i casi, indice del fatto che questi due problemi risultano entrambi troppo semplici. Questo vuol dire che qualsiasi mappa andrebbe bene per il bilanciamento del gioco con la configurazione da noi adottata in questa sottosezione, situazione che risulta molto strana. Analizzando le morti e le uccisioni effettuate in partita con questa configurazione abbiamo notato un comportamento che spiega il risultato ottenuto, il numero di uccisioni effettuate dal *bot* con arma *rifle* sono irrisorie rispetto a quelle del *bot* avversario. Volendo bilanciare le morti dei *bot* e considerando che il *bot* con arma *rifle* effettua poche uccisioni, a parità di morti dei due *bot*, quello con arma *lanciagranate* commette molti suicidi. Questo vuol dire che in realtà, massimizzando l'entropia delle morti in una situazione di vantaggio in termini di abilità da parte del *bot* dotato di *lanciagranate*, si bilanciano le morti di quest'ultimo *bot* rispetto alle sue uccisioni. Fornendo un esempio, in una mappa con entropia delle morti pari a 1 (quindi massima) il numero totale delle

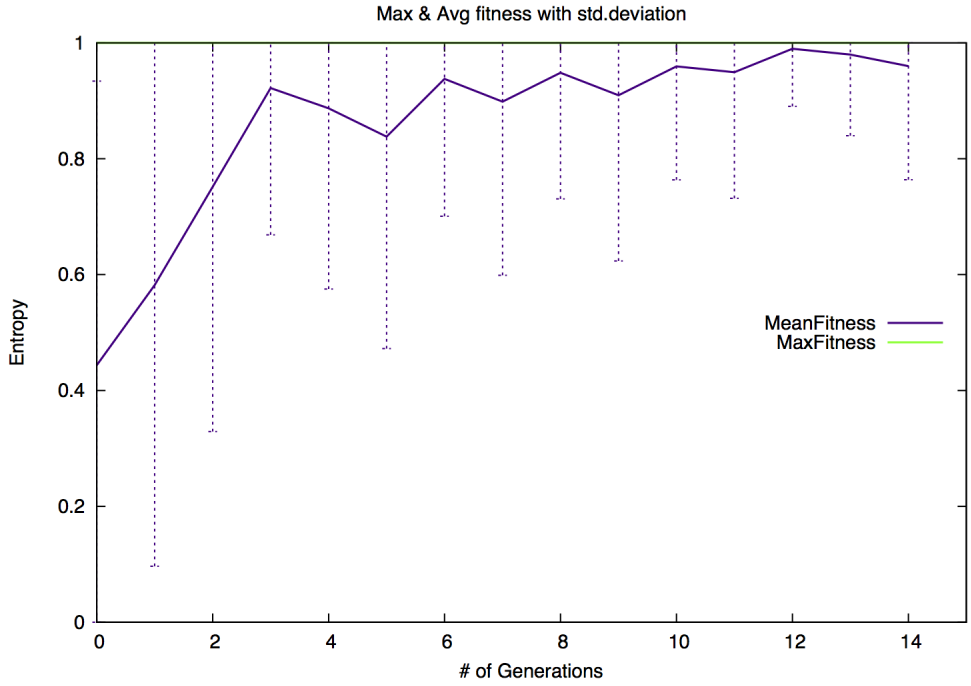



Figura 6.12: Fitness massima e media con rappresentazione *All-Black* nelle generazioni, Armi *Rifle-Lanciagranate*, Abilità 20-80

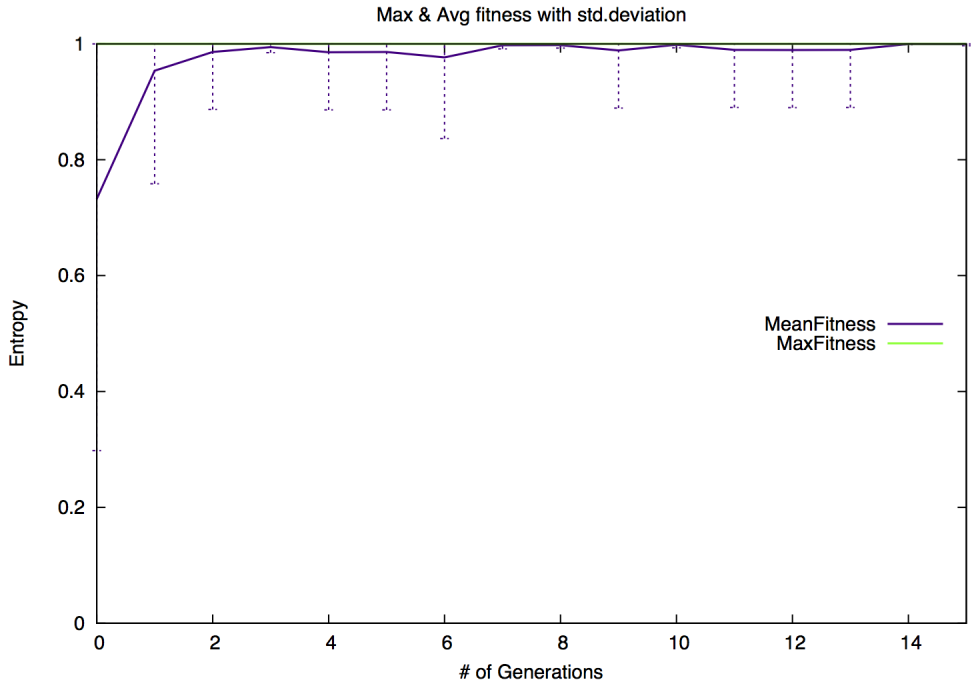
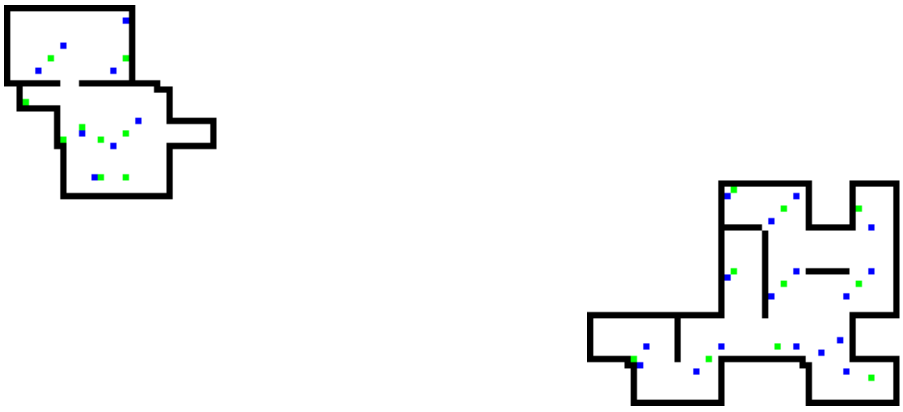



Figura 6.13: Fitness massima e media con rappresentazione *Grind* nelle generazioni, Armi *Rifle-Lanciagranate*, Abilità 20-80



(a) Mappa migliore con rappresentazione *All-Black*

(b) Mappa migliore con rappresentazione *Grind*

Figura 6.14: Mappe vincitrici dell'evoluzione con armi *Rifle-Lanciagranate*, Abilità 20-80

morti del $bot_1$ (quello con arma *lanciagranate*) risultano uguali a quelle del $bot_2$ (quello con arma *rifle*), pari a 71. Analizzando le uccisioni effettuate dai *bot* nel corso della partita, però, il numero di uccisioni effettuate dal $bot_1$ risultano pari a 71 mentre quelle effettuate dal $bot_2$ risultano pari a 13. Questo vuol dire che, escludendo dalle morti del $bot_1$ le uccisioni del $bot_2$, i $Suicidi_{bot1} = 71 - 13 = 58$, in realtà stiamo quindi bilanciando i suicidi del $bot_1$ rispetto alle sue uccisioni (58 rispetto a 71). Infatti, in questo esempio, il contributo dato dal $bot_2$ (quello con arma *rifle* e abilità bassa) alla soluzione è il 13% del totale, tutto il resto dipende solo dal $bot_1$. È possibile quindi concludere che, se a un *bot* dotato di *lanciagranate* viene impostato un livello di abilità maggiore rispetto ad un altro *bot* che non usa la stessa arma, non è utile utilizzare come funzione obiettivo l'entropia delle morti. Le figure 6.14a e 6.14b mostrano due delle possibili mappe con entropia massima ottenibili, la prima con rappresentazione *All-Black* e la seconda con rappresentazione *Grind*.

## 6.3 Sommario

In questo capitolo abbiamo mostrato i risultati dei nostri esperimenti riguardanti il bilanciamento del gioco in situazioni più complesse, nella quale lo sbilanciamento da noi introdotto consisteva sia nel dotare i due *bot* di armi differenti che di impostare un divario tra le loro abilità. Impostando prima il livello di abilità più elevato ad un *bot* e poi all'altro, abbiamo inizialmente analizzato i risultati degli esperimenti svolti su simulazioni nelle quali i *bot* erano dotati uno di arma *rifle* e l'altro di *motosega* (Sezione 6.1) e successivamente i risultati degli espe-

| Impostazioni iniziali | | Rappresentazione | |
|---|---|---|---|
| | | *All-Black* | *Grind* |
| Arma *Rifle-Motosega* | Abilità 80-20 | miglioramento lieve | troppo semplice |
| | Abilità 20-80 | miglioramento lieve | miglioramento lieve |
| Arma *Rifle-Lanciagranate* | Abilità 80-20 | miglioramento lieve | miglioramento lieve |
| | Abilità 20-80 | troppo semplice | troppo semplice |

Tabella 6.1: Risultati ottenuti sul problema del bilanciamento negli esperimenti effettuati nel capitolo 6

rimenti con un *bot* dotato di arma *rifle* e l'altro di *lanciagranate* (Sezione 6.2). La tabella riassuntiva 6.1 mostra sinteticamente i risultati ottenuti in questo capitolo, valorizzandoli in base alla rappresentazione usata (All-Black o Grind) e alle varie condizioni iniziali da noi impostate (abilità *bot* e arma usata).

# Capitolo 7

# Conclusioni e sviluppi futuri

In questo lavoro ci siamo chiesti se è possibile generare mappe per un videogioco sparatutto in prima persona che, data una situazione di svantaggio di un giocatore rispetto ad un altro, rendano bilanciato il combattimento tra i due giocatori. Per rispondere a questa domanda, basandoci sul lavoro precedentemente svolto in [12], abbiamo generato mappe, tramite algoritmi evolutivi, per il videogioco Cube 2, valutandone il bilanciamento. Utilizzando le rappresentazioni *All-Black* e *Grind* per generare le mappe, abbiamo esteso gli operatori e la funzione obiettivo dell'algoritmo genetico utilizzato in [12] focalizzandoci sull'obiettivo di massimizzare l'entropia delle uccisioni (o, in alcuni casi, delle morti), misura che ci ha permesso di valutare il bilanciamento di una partita alla quale partecipavano due giocatori. Per estrarre i valori utili al calcolo di questa funzione obiettivo (l'entropia delle uccisioni) è risultato necessario raccogliere alcuni dati (tra i quali il numero di morti e il numero di uccisioni di ogni giocatore) giocando una partita in ogni mappa generata. Considerando che giocare una partita in media dura 15 minuti, valutare ogni mappa generata avrebbe implicato l'utilizzo di una quantità molto elevata di tempo di cui noi non disponevamo. Inoltre, far testare manualmente ogni mappa generata ai giocatori non sarebbe stato fattibile, in quanto l'elevata quantità di partite da giocare, una per ogni mappe generata (con una popolazione di 100 mappe fatta evolvere per 30 generazioni le mappe da valutare sono 3000), avrebbe portato loro noia e frustrazione. Per risolvere queste limitazioni abbiamo utilizzato i *bot* forniti da Cube 2 per simulare la partite, così da raccogliere i dati utili al calcolo della funzione obiettivo per ogni mappa. Questi *bot* utilizzavano un'unica IA ed era possibile configurarli agendo su un unico parametro, il livello di abilità. Per impostare correttamente il grado desiderato di sbilanciamento all'inizio della partita, abbiamo analizzato come il livello di abilità del *bot* influisca sulla sua efficienza nell'utilizzo di una determinata arma, per ogni arma disponibile

in Cube 2. Successivamente abbiamo effettuato diversi esperimenti partendo da configurazioni differenti (combinazioni del livello di abilità dei *bot* ed arma fornita ad essi) ed applicando un algoritmo evolutivo al fine di generare mappe che massimizzassero il bilanciamento della partita. In alcuni casi abbiamo inoltre generato le heatmap delle uccisioni per la mappa con valore di fitness più elevato alla fine di un'evoluzione, in modo da approfondire l'analisi per una mappa che presentava caratteristiche particolari di difficile interpretazione.

I risultati ottenuti impostando abilità diverse ai *bot* si sono rivelati interessanti, mostrando un lieve aumento, nelle generazioni, del bilanciamento della partita. Nello specifico, abbiamo ottenuto il risultato più interessante (ovvero un incremento costante, durante le generazioni, del bilanciamento della partita) utilizzando la rappresentazione *All-Black* per generare le mappe, dotando entrambi i *bot* di arma *rifle* e impostando loro abilità diverse ($bot_1$ abilità uguale a 80 e $bot_2$ abilità uguale a 35). Dai risultati ottenuti in questo lavoro possiamo quindi dire che in Cube 2, data una situazione iniziale di svantaggio di un giocatore rispetto all'altro, è effettivamente possibile bilanciare i combattimenti tra due giocatori agendo sulla mappa. L'utilizzo di *bot* con una stessa IA parametrizzabile tramite un solo valore ha però limitato i risultati delle nostre analisi per alcune configurazioni iniziali. Prime le configurazioni nelle quali i *bot* erano dotati di arma diversa ma di abilità uguale, il questi casi, infatti, il problema di bilanciare un combattimento al quale partecipavano *bot* dotati di arma diversa risultava troppo semplice a causa della stessa strategia adottata dai due *bot*, invariante rispetto all'arma utilizzata. Le altre configurazioni limitate dall'IA dei *bot* sono state quelle nelle quali un *bot* veniva dotato di arma *lanciagranate*. Quest'ultima fa eccezione rispetto alle altre armi perché i suoi colpi rimbalzano più volte sul terreno prima di esplodere, rendendo necessario calcolare la traiettoria che avrà il proiettile per utilizzare l'arma in modo efficiente. I *bot* di Cube 2 si limitano a mirare correttamente l'avversario senza curarsi della traiettoria del proiettile, necessiterebbero quindi l'utilizzo di una strategia molto più complicata per utilizzare efficientemente il *lanciagranate*.

Uno degli sviluppi futuri di questo lavoro sarà la ricerca di un videogioco che utilizzi rappresentazioni di mappe compatibili con quelle da noi utilizzate e che implementi *bot* maggiormente sofisticati rispetto a quelli di Cube 2 ai fini di replicare (ed estendere) gli esperimenti svolti in questa sede, in modo da ottenere risultati migliori e maggiormente accurati. Un possibile videogioco candidato è *Shootmania*[1], il quale, però, ad oggi implementa un'IA basilare per i *bot* che non possiede complessità sufficiente per lo svolgimento dei nostri esperimenti. In futuro vorremmo inoltre ripetere i nostri esperimenti utilizzando

[1] http://maniaplanet.com/shootmania/

come funzione obiettivo dell'algoritmo genetico la *dispersione delle uccisioni*, grado che indica quanto le uccisioni effettuate dai giocatori siano sparse per la mappa (dispersione minima se le uccisioni sono tutte localizzate in un punto della mappa, dispersione massima se il numero di uccisioni effettuate è uguale in ogni punto della mappa). Osservando le heatmap infatti, ci siamo accorti che, a causa della strategia utilizzata dall'IA dei *bot*, le uccisioni tendono ad essere concentrate in un punto della mappa (solitamente il punto centrale). Sarebbe interessante analizzare se sia possibile generare mappe nelle quali ogni zona sia sfruttata per il combattimento, minimizzando le zone inutilizzate della mappa. Sapendo che l'IA implementata in Cube 2 non è particolarmente sofisticata, non è detto però che questa nuova funzione obiettivo porti a risultati rilevanti.

# Bibliografia


[1] Kyle Orland, Scott Steinberg, Dave Thomas,
*The Videogame Style Guide and Reference Manual*

[2] Julian Togelius, Mike Preuss,Georgios N. Yannakakis,
*Towards multi-objective procedural map generation*

[3] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, Cameron Browne,
*Search-Based Procedural Content Generation: A Taxonomy and Survey.*

[4] Daniel Ashlock, Colin Lee, Cameron McGuinness,
*Search-Based Procedural Generation of Maze-Like Levels*, 2011

[5] Franz Rothlauf,
*Representations for Genetic and Evolutionary Algorithms*, 2006

[6] Kalyanmoy Deb,
*Multi-Objective Optimization Using Evolutionary Algorithms: An Introduction*, 2011

[7] Alberto Uriarte, Santiago Ontañón
*PSMAGE: Balanced Map Generator for Starcraft*, 2013

[8] Julian Togelius, Mike Preuss, Nicola Beume, Simon Wessing, Johan Hagelbäck, Georgios N. Yannakakis,
*Multiobjective Exploration of the StarCraft Map Space*, 2010

[9] Julian Togelius, Mike Preuss, Nicola Beume, Simon Wessing, Johan Hagelbäck, Georgios N. Yannakakis, Corrado Grappiolo,
*Controllable Procedural Map Generation via Multiobjective Evolution*, 2013

[10] Tobias Mahlmann, Julian Togelius and Georgios N. Yannakakis,
*Spicing up map generation*, 2012

[11] Raúl Lara-Cabrera, Carlos Cotta, Antonio J. Fernández-Leiva,
*Procedural Map Generation For a RTS Game*, 2012

[12] Luigi Cardamone, Pier Luca Lanzi, Georgios Yannakakis and Julian Togelius,
*Evolving interesting maps for a first person shooter*
Proceedings of the 2011 international conference on Applications of evolutionary computation, Volume Part I, 63 -72

[13] *List of Games Featuring Procedural Generation*
http://pcg.wikidot.com/category-pcg-games

[14] N. Sorenson and P. Pasquier,
*Towards a generic framework for automated video game level creation*,
In Proceedings of the European Conference on Applications of Evolutionary Computation (EvoApplications), volume 6024, pages 130-139. Springer LNCS, 2010.

[15] D. Ashlock, T. Manikas, and K. Ashenayi,
*Evolving a diverse collection of robot path planning problems*,
In Proceedings of the Congress On Evolutionary Computation, pages 6728-6735, 2006.

[16] *Procedural Content Generation Wiki*
http://pcg.wikidot.com

[17] Sean Luke,
*Essentials of Metaheuristics*, 2013, pag: 31-58

[18] Sean Luke,
*Essentials of Metaheuristics*, 2013, pag: 66

[19] Tracy Fullerton,
*Game Design Workshop: a play centric approach to creating innovative games*, Fullerton, 2nd edition, pag 286-306,

[20] Keith Burgun,
*Understanding Balance in Video Games Keith Burgun*, Gamasutra, 2011
http://www.gamasutra.com/view/feature/134768/understanding_balance_in_video_.php

[21] Pascal Luban,
*Multiplayer Level Design In-Depth*, Gamasutra,
http://www.gamasutra.com/view/feature/1795/multiplayer_level_design_indepth_.php

[22] Epic Games,
*Multiplayer Map Theory (Gears of War)*,
http://udn.epicgames.com/Three/GearsMultiplayerMapTheory.html

[23] Edge Staff,
*Secrets of Multiplayer Maps*,
http://www.edge-online.com/features/feature-secrets-multiplayer-maps/

[24] Dodger,
*Designing FPS Multiplayer Maps*,
http://dodger.uselessopinions.com/?p=194,
http://dodger.uselessopinions.com/?p=197

[25] Tommy Reddad, Clark Verbrugge,
*Geometric Analysis of Maps in Real-Time Strategy Games: Measuring Map Quality in a Competitive Setting*,
McGill University, 2012

[26] *Map generation, Diablo wiki*
http://diablo.gamepedia.com/Map_Generation

[27] *A Bestiary of Japanese FPS Player Types*
http://kotaku.com/a-bestiary-of-japanese-fps-player-types-512575576