Politecnico di Milano
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

Sviluppo di un'intelligenza artificiale tramite algoritmo genetico per il gioco del Tetris

Relatore: Prof. Andrea Bonarini

Tesi di laurea di:
Luca Stefano Di Leo Matricola 786122

Anno Accademico 2014–2015

L'unico limite all'intelligenza artificiale è quello che le viene imposto

Ringraziamenti

Desidero ringraziare tutti coloro che mi hanno aiutato nella realizzazione della mia Tesi.

In particolare tengo a ringraziare il mio relatore, Andrea Bonarini, per la disponibilità e la possibilità che mi ha offerto di lavorare a questo progetto.

Ringrazio la mia fidanzata Francesca, senza la quale non avrei mai avuto la serenità e la tranquillità necessaria per terminare un percorso così lungo e travagliato. Il suo supporto morale è stato molto spesso fondamentale nei momenti difficili. Posso dire di essere proprio un uomo fortunato ad averla accanto.

Ringrazio mio padre per tutto il tempo passato a discutere e ragionare su molti temi, la sua dedizione alla mia istruzione è stata sempre esemplare e ha contribuito notevolmente alla conclusione di questo ciclo di studi.

Ringrazio mia madre per il supporto morale e psicologico che non mi è mai mancato. Anche quando le motivazioni scarseggiavano ha sempre avuto la parola giusta per rimettermi in carreggiata.

Ringrazio mio fratello per tutto quello che abbiamo passato assieme tra casa e lavoro, la sua presenza è sempre stata rassicurante e senza dubbio uno stimolo per me.

Ringrazio Fulvio per il sostegno morale durante il percorso di studi, è sempre stato di grande aiuto.

Infine tengo particolarmente a ringraziare Marco Frigo per la possibilità lavorativa che mi ha dato in questi anni. Grazie a lui ho visto come si lavora in azienda e ho imparato tanto su molti aspetti che riguardano il mio lavoro. Ringrazio anche i miei colleghi Riccardo e Matteo, noi Associati siamo un grande team!

Sommario

La programmazione genetica applicata ai giochi permette di realizzare euristiche complesse che superano le prestazioni degli algoritmi tradizionali. In questo lavoro verrà sviluppata un'intelligenza artificiale per il gioco del Tetris. Inoltre verrà dimostrato come un'IA realizzata in questo modo è in grado di gestire in maniera disinvolta sia le non linearità sia la complessità intrinseca del problema. Analizzando i risultati ottenuti verrà stimata la difficoltà legata ad ogni tetromino e formulate delle ipotesi sulla strategia impiegata dall'intelligenza artificiale. Variando i parametri che governano il processo evolutivo verranno confrontate le prestazioni e la velocità di convergenza dell'algoritmo genetico utilizzato. I risultati finali mostreranno come l'approccio genetico sia efficace e applicabile a molti altri casi.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Questo capitolo introdurrà le tematiche trattate nel presente documento. Si partirà da una panoramica generale e verrà descritto il lavoro svolto. Infine si presenterà l'organizzazione dei capitoli seguenti per fornire al lettore una visione di insieme dell'elaborato.

1.1 Inquadramento generale

Il presente lavoro di tesi si svolge nell'ambito dell'intelligenza artificiale, in particolare nel campo delle euristiche di ricerca basate su principi evoluzionistici. Lo scopo della tesi è quello di sviluppare un'intelligenza artificiale per il gioco del Tetris tramite un algoritmo genetico appositamente realizzato. Combinando le capacità di un elaboratore con i principi dell'evoluzione si è in grado di ottenere risultati sorprendenti, soprattutto se il problema che è necessario risolvere non ha un modello matematico di riferimento o presenta forti non linearità e discontinuità.

L'intelligenza artificiale è un campo in continua evoluzione e gli algoritmi genetici hanno visto aumentare la loro popolarità negli ultimi anni. L'incremento delle prestazioni degli elaboratori sfruttando la moltiplicazione delle unità di elaborazione ha permesso agli algoritmi genetici di raggiungere velocità di esecuzione notevoli grazie alla facilità di parallelizzazione del processo evolutivo.

1.2 Descrizione del lavoro e obbiettivi

L'idea di sviluppare un algoritmo genetico in grado di giocare a Tetris non è un'idea nuova [6, 13, 15, 17, 20]. Online è presente molta letteratura al riguardo ma raramente si analizza nel dettaglio il comportamento dell'intelligenza artificiale e si formulano ipotesi sulla struttura del gioco. Lo spazio degli stati del Tetris è discreto e molto ampio, se si considera l'intero campo di gioco si contano 10^{60} stati

[1]. Utilizzare una ricerca esaustiva è impraticabile, per questo motivo si impiegano euristiche in grado di approssimare un comportamento ottimale.

In questo lavoro ci si soffermerà sul modello dei dati scelto per affrontare diversi tipi di obbiettivi e sulle caratteristiche necessarie della piattaforma software per soddisfare i requisiti del problema. Per rendere praticabile la strada degli algoritmi genetici è necessario che la simulazione del'ambiente di gioco sia molto veloce, per questo motivo verrà realizzata un'implementazione ad hoc che riduce al minimo il tempo di esecuzione.

Sono stati individuati due obbiettivi principali nel gioco del Tetris: il primo riguarda la durata assoluta della partita e il secondo la quantità di punti che l'intelligenza artificiale è in grado di ottenere a parità di tetramini giocati. Sono stati condotti molti esperimenti con entrambi gli obbiettivi e i risultati sono stati talvolta molto sorprendenti.

Il Tetris è difficile anche solo da approssimare [2], per questo motivo è stata condotta un'analisi di complessità sulle diverse tipologie di tetramini esaminado le strategie adottate dell'intelligenza artificiale.

Per assicurarsi che gli esperimenti forniscano risultati affidabili sono state condotte delle prove di ripetibilità del processo evolutivo. Un algoritmo genetico non è altro che una particolare tipologia di euristica di ricerca casuale guidata da una funzione obbiettivo. Come tutti i tipi di ottimizzazione potrebbe accadere che il processo rimanga bloccato in un massimo locale e non riesca ad esplorare in maniera efficiente lo spazio delle soluzioni. Per questo motivo, dopo ogni risultato sperimentale, viene condotta un'analisi volta a verificare la bontà del risultato.

1.3 Struttura della tesi

La tesi è strutturata nel modo seguente:

Nel capitolo 2 verranno trattati gli algoritmi genetici sotto diversi aspetti. Si parlerà della storia di questo tipo di euristiche di ricerca e del loro funzionamento, sottolineando i vantaggi e gli svantaggi che offrono.

Nel capitolo 3 verrà approfondito il Tetris a partire dalla sua storia, le sue caratteristiche principali e lo scopo del gioco. Si parlerà della complessità del gameplay e alla difficoltà di realizzare un modello matematico.

Nel capitolo 4 verrà mostrato il modello dei dati utilizzato negli esperimenti del capitolo 6. Si specificherà la struttura genetica del cromosoma e si parlerà dei diversi obbiettivi che un'intelligenza artificiale può raggiungere nel gioco del Tetris.

Nel capitolo 5 verrà approfondita l'architettura software alla base degli esperimenti del capitolo 6 con particolare attenzione alle ottimizzazioni implementate per rendere percorribile la strada degli algoritmi genetici.

Nel capitolo 6 verranno mostrati i risultati degli esperimenti condotti. Verranno trattati i diversi obbiettivi dell'intelligenza artificiale ovvero l'ottimizzazione per numero di tetramini e l'ottimizzazione per punteggio. In seguito verrà proposta un'estensione del cromosoma standard visto nel capitolo 4 al fine migliorare ulteriormente i risultati ottenuti. A seguire verrà condotta un'analisi di ripetibilità sull'evoluzione tramite algoritmo genetico e si cercherà di stabilire la complessità dei singoli tetramini variando la distribuzione di probabilità della sequenza di pezzi in uscita dal generatore di numeri casuali.

Capitolo 2

Algoritmi genetici

2.1 Introduzione

Prima di addentrarsi nella descrizione nei minimi dettagli della storia e del funzionamento degli algoritmi genetici è bene domandarsi cosa sia un algoritmo genetico e su quale principio si fonda. Nel campo dell'intelligenza artificiale gli algoritmi genetici fanno parte della classe degli algoritmi evolutivi. La caratteristica di questi ultimi è quella di trovare soluzioni ai problemi utilizzando tecniche mutuate dall'evoluzione naturale. La ricerca di una soluzione ad un problema è affidata ad un processo iterativo che seleziona e ricombina tra loro soluzioni sempre più raffinate fino al raggiungimento di un criterio di ottimalità. In un algoritmo genetico la popolazione di soluzioni viene spinta verso un dato obbiettivo dalla pressione evolutiva. Questa caratteristica è ottenuta tramite una particolare funzione, chiamata funzione di fitness, che è in grado di sintetizzare in un solo parametro la qualità della soluzione.

Ogni soluzione, o cromosoma, è costituito da un un insieme di geni. Questi geni prendono parte al processo di ricombinazione e mutazione al fine generare nuove soluzioni. Generalmente, l'evoluzione ha inizio da una popolazione costituita da individui dotati di materiale genetico casuale. Ad ogni iterazione dell'algoritmo, o generazione, la funzione di fitness viene valutata per ogni unità. Gli individui che si mostrano più adatti vengono selezionati e ricombinati tra loro con diverse tecniche che verranno esaminate in seguito. La generazione successiva conterrà la prole ottenuta delle unità migliori della precedente popolazione. Iterativamente le unità si muovono verso la soluzione, guidate dalla funzione di fitness. Generalmente l'algoritmo termina quando si raggiunge un numero massimo di generazioni o si è giunti ad un criterio di ottimalità.

2.2 Storia

I primi esempi di quelli che possono essere chiamati algoritmi genetici sono apparsi verso la fine degli anni '50 e inizio anni '60, e furono realizzati da biologi alla ricerca di modelli di evoluzione naturale. In quegli anni nessuno realizzò che tali strategie potessero essere applicabili non solo al caso biologico ma anche a moltissimi problemi di ottimizzazione [14, p.2]. Nel 1962, ricercatori come G.E.P. Box, G.J. Friedman, W.W. Bledsoe e H.J. Bremermann svilupparono indipendentemente gli uni dagli altri algoritmi ispirati all'evoluzione, applicandoli all'ottimizzazione funzionale e all'apprendimento automatico. I loro lavori, purtroppo, non ebbero molto successo. Fu diversa la ricezione del lavoro di Ingo Rechenberg, che nel 1965 introdusse una tecnica chiamata «strategia evolutiva», sebbene fosse più simile all'hill-climbing che ad un algoritmo genetico [8, p.146]. Questa tecnica non comprendeva nè una popolazione nè il crossover. Un genitore subiva una mutazione per generare dei figli, il migliore diventava il padre della generazione successiva e, solo in un secondo momento, si introdusse il concetto di popolazione.

Il successivo importante sviluppo arrivò nel 1966, quando L.J. Fogel, A.J. Owens e M.J. Walsh introdussero in USA una tecnica chiamata «programmazione evoluzionistica». In questo metodo, le soluzioni candidate per risolvere il problema venivano rappresentate come delle macchine a stati finite [14, p.2] [7, p.105]. Come nella strategia evolutiva di Rechenberg, il loro algoritmo funzionava mutando casualmente queste macchine conservando le migliori. Quello che ancora mancava a queste tecniche era il concetto di crossover, che è uno degli aspetti più importanti degli algoritmi genetici come li conosciamo oggi.

Nel 1962, John Holland pubblicò una ricerca sui sistemi adattivi nei quali delineava i fondamenti per gli sviluppi futuri. Tra tutti i ricercatori dell'epoca, Holland fu il primo a proporre esplicitamente l'utilizzo del crossover e di altri operatori di ricombinazione. Il punto di svolta nel campo degli algoritmi genetici arrivò nel 1975 con la pubblicazione del libro «Adaptation in Natural and Artificial Systems» [10]. Partendo dalle ricerche dello stesso Holland e dei suoi colleghi dell'università del Michigan, questo libro fu il primo che con sistematicità e rigore presentò il concetto di sistema digitale che utilizza mutazioni, selezione e crossover come strategia per la risoluzione di problemi. Il testo cerca anche di sostenere gli algoritmi genetici da un punto di vista teorico introducendo la nozione di schema [14, p.3] [8, p.147]. Quello stesso anno, Kenneth De Jong, affermò il grande potenziale degli algoritmi genetici mostrando che potevano essere impiegati con successo in molte funzioni di test, inclusi problemi caratterizzati da forte rumore e discontinuità [7, p.107].

Sulla base di questi lavori è partito l'interesse verso la programmazione evolutiva. Tra l'inizio e la metà degli anni '80, gli algoritmi genetici vennero applicati in moltissimi ambiti, da problemi matematici come quello dello zaino e la colorazione di mappe,

a problemi reali di ingegneria come il controllo del flusso delle tubature, il riconoscimento di schemi, la classificazione di oggetti e l'ottimizzazione strutturale [7, p.128]. Inizialmente queste applicazioni furono principalmente teoriche. In seguito la ricerca prese piede e gli algoritmi genetici si spostarono nel settore commerciale aiutati dalla crescita esponenziale della potenza computazionale e dallo sviluppo di internet.

Oggi, la computazione evolutiva è un campo in espansione e gli algoritmi genetici risolvono problemi riguardanti la vita di tutti i giorni in molte aree di studio, come ad esempio la predizione dei mercati finanziari, l'ingegneria aerospaziale, il design di microchip, la biochimica, la biologia molecolare, le tabelle degli orari degli aeroporti e le linee di montaggio [8, p.147]. La potenza dell'evoluzione ha toccato praticamente ogni campo degno di nota, dando forma al mondo intorno a noi in moltissimi modi e nuove applicazioni della computazione evolutiva vengono scoperte ogni giorno. Al centro di tutto questo non c'è altro che la semplice e potente intuizione di Charles Darwinn: le mutazioni casuali unite alle leggi di selezione costituiscono una tecnica di risoluzione dei problemi di incredibile potenza e con applicazioni virtualmente illimitate.

2.3 Funzionamento

L'implementazione di un algoritmo genetico non è una scienza esatta. La scelta dei parametri che verranno utilizzati è fondamentale per ottenere buoni risultati. Per raggiungere una soluzione ottimale in breve tempo sono necessarie diverse sperimentazioni sulla tipologia di algoritmo genetico da implementare. Nelle prossime sezioni verranno trattati gli aspetti più importanti che riguardano il funzionamento degli algoritmi genetici e particolare attenzione verrà posta sulle caratteristiche impiegate nel caso in esame.

2.3.1 Codifica

Prima di mettere al lavoro un algoritmo genetico è necessario un metodo per codificare le potenziali soluzioni in una forma comprensibile all'elaboratore. Un approccio comune è quello di codificare la soluzione sotto forma di stringa binaria: sequenze di 1 e di 0, dove ogni bit rappresenta il valore di un aspetto della soluzione. Un altro metodo simile è quello di codificare le soluzioni come una lista di numeri interi o reali. Quest'ultimo ha il vantaggio di garantire maggiore precisione e complessità rispetto al corrispettivo metodo binario. Un terzo approccio codifica gli individui come stringhe di lettere, nel quale ogni lettera rappresenta un preciso aspetto della soluzione. Il vantaggio di questi tre metodi di rappresentazione è la facilità di definizione degli operatori di mutazione: basta scambiare un 1 con uno 0 e vice versa, aggiungere o sottrarre un numero casuale, o cambiare una lettera con un'altra.

Un ulteriore metodologia, diversa dalle precedenti, consiste nel rappresentare la soluzione con strutture dati a forma di albero. In questo caso i cambiamenti casuali possono essere effettuati alterando un operatore, modificando il valore di un nodo dell'albero o sostituendo un sotto albero con un altro. È importante notare come gli algoritmi genetici non necessitano di rappresentare le soluzioni candidate con strutture dati di lunghezza fissa. Principalmente dipende dal problema che si intende risolvere, non esiste un rigoroso criterio di scelta in questo senso.

2.3.2 Funzione di fitness

Una funzione di fitness è un particolare tipo di funzione obbiettivo che riassume, in una singola misura di merito, quanto una soluzione è adatta a risolvere un problema. In un algoritmo genetico, dove il raggiungimento della soluzione è delegato all'elaborazione del computer, è compito del programmatore sviluppare una buona funzione di fitness. Mancanze in questo senso possono portare a difficoltà di convergenza o condurre verso soluzioni sbagliate. Non è infatti strano vedere il processo evolutivo sfruttare bug e incoerenze della funzione di fitness per raggiungere soluzioni eccezionalmente buone che in seguito si riveleranno prive di ogni fondamento. Per questo motivo è necessario porre molta attenzione durante lo sviluppo della funzione di fitness in quanto costituirà il landscape all'interno del quale si muoverà l'algoritmo. Inoltre la funzione di fitness ha dei requisiti in fatto di velocità di esecuzione. Il procedimento iterativo dell'algoritmo genetico per risultare efficace ha bisogno che il costo di valutazione della fitness sia il minore possibile. Per questo motivo vengono spesso impiegate approssimazioni matematiche di problemi più complessi.

La funzione di fitness ha il compito di stimare la bontà di una soluzione e, a seconda del tipo di problema, può essere mutabile o immutabile. Nel primo caso la funzione di fitness cambia di generazione in generazione, come nel caso di un algoritmo genetico per la previsione dell'andamento borsistico. Nel secondo caso, ogni valutazione della funzione nello stesso punto avrà sempre lo stesso valore. Questo tipo di funzione di fitness viene spesso utilizzata per l'ottimizzazione di una funzione multi variabile.

Nel caso in esame del Tetris la funzione di fitness è mutabile perché dipende dalla sequenza di pezzi del generatore di numeri casuali. Si vedranno in seguito gli accorgimenti impiegati per ottenere una misurazione affidabile in questo specifico caso.

2.3.3 Selezione

La selezione è una fase importante dell'algoritmo genetico, in quanto in essa vengono scelte le unità da ricombinare per formare la prossima generazione. Esistono

molti metodi di selezione, il più comune è quello proporzionale o roulette che consiste nei seguenti step:

1. La funzione di fitness viene valutata per ogni unità, e il valore viene normalizzato. Per normalizzazione si intende la divisione del valore per la somma di tutti gli altri, in modo che la somma dei valori risultanti sia uno.
2. La popolazione viene ordinata in ordine decrescente per il valore della funzione di fitness.
3. Vengono calcolati i valori cumulati normalizzati della fitness, ovvero si somma la fitness di ogni unità con quella di tutte le quelle che la precedono.
4. Utilizzando un generatore di numeri casuali si estrae un numero compreso tra 0 e 1.
5. Viene selezionata l'unita che ha il valore della fitness cumulata normalizzata maggiore del numero estratto.

Il valore della fitness dell'unità viene utilizzato per associare una probabilità di selezione, che risulta pari a

$$p_i = \frac{f_i}{\sum_{j=0}^{N} f_j}$$

con N il numero di individui nella popolazione.

Evoluzione del metodo di selezione proporzionale è il campionamento universale stocastico, che utilizza un singolo valore casuale per scegliere tutte le soluzioni in intervalli equamente distribuiti. Questo metodo dà alle unità più deboli appartenenti alla popolazione la possibilità di essere scelte. Così il processo di evoluzione ha la capacità di uscire agilmente dai massimi locali e di esplorare in maniera più completa lo spazio delle soluzioni.

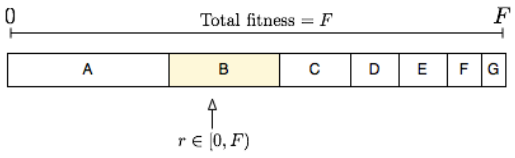


Figura 2.1: Selezione proporzionale

Un altro metodo per selezionare le unità è l'utilizzo di un sistema a tornei. Si costituiscono diversi gruppi di unità scelte casualmente dalla popolazione e il vincitore è l'individuo che ha maggior fitness. In questo modo è anche possibile regolare la pressione evolutiva variando la dimensione dei tornei, infatti più è grande il gruppo, meno probabilità hanno le unità più deboli di venir scelte.

In ultimo è possibile utilizzare il metodo di selezione per troncamento: le unità vengono ordinate per fitness e solo una percentuale di loro viene utilizzata per ottenere la generazione successiva. Questa tecnica è meno sofisticata rispetto alle altre, ma ha il vantaggio di essere veloce e di facile implementazione.

Un'altra strategia di selezione molto importante è l'etilismo. Per evitare che i cromosomi migliori vengano perduti a causa della ricombinazione, viene mantenuta una piccola percentuale di unità della generazione precedente in corrispondenza dei massimi valori di fitness. In questo modo si aumentano le probabilità di avere un andamento monotono crescente sia della massima fitness sia della fitness media della popolazione.

2.3.4 Crossover

Il crossover gioca un ruolo fondamentale nella fase di ricombinazione. Alcune parti dei geni delle soluzioni candidate all'evoluzione vengono mescolate per ricavare nuove soluzioni. Gli operatori più comunemente utilizzati sono:

- Crossover ad un punto: prese due soluzioni si sceglie un punto del cromosoma nel quale tagliare. La prima nuova soluzione ottenuta sarà data dalla combinazione della parte iniziale della prima e della parte finale della seconda, mentre la seconda nuova soluzione sarà data dalla parte finale della prima soluzione unita alla parte iniziale della seconda.

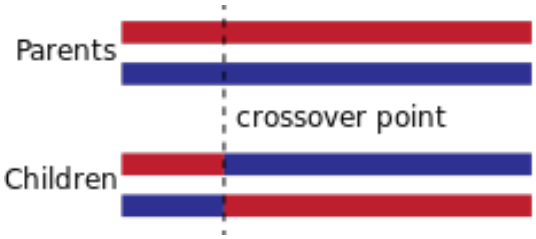


Figura 2.2: Crossover ad un punto

- Crossover in due punti: partendo da due soluzioni si scelgono due punti del cromosoma nei quali effettuare i tagli. Come nel caso del crossover ad un punto, le unità si scambiano alternativamente i segmenti di materiale genetico compresi tra un taglio e l'altro. Si individuano tre distinte sezioni: iniziale, centrale e finale. Rispettivamente la prima unità otterrà la parte iniziale della prima, la parte centrale della seconda e la parte finale della prima. La seconda unità, invece, otterrà la parte iniziale della seconda, la parte centrale della prima e la parte finale della seconda.

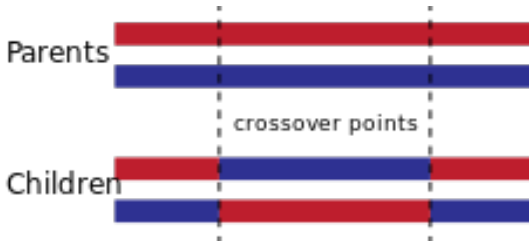


Figura 2.3: Crossover in due punti

- Crossover uniforme: consiste nello scambiare casualmente il materiale genetico tra le soluzioni candidate all'evoluzione. Si può regolare la percentuale di scambi da effettuare per aumentare o diminuire il tasso di ricombinazione.

- Crossover aritmetico: al posto di scambiare materiale genetico come nei casi precedenti, si procede ad applicare un operatore matematico ai due genomi per ottenere una nuova soluzione.

2.3.5 Mutazione

La mutazione è un operatore utilizzato per mantenere la diversità genetica da una generazione all'altra. Funziona alterando il valore di uno o più geni ed è governata da un parametro che specifica la probabilità di mutazione. La mutazione deve permettere all'algoritmo genetico di uscire dai minimi locali evitando che il pool genetico diventi troppo uniforme. Per questo motivo la probabilità di mutazione non deve essere troppo bassa ma neppure troppo alta, perché trasformerebbe l'algoritmo genetico in una semplice ricerca casuale.

Nel caso di codifica binaria del cromosoma, la mutazione consiste nell'invertire uno o più bit del cromosoma. Se il cromosoma è costituito da valori reali si possono utilizzare diverse tecniche. Ad esempio si può prevedere l'aggiunta di un delta casuale al gene, oppure la sostituzione completa con un numero casuale.

2.4 Vantaggi

Il primo e più importante vantaggio di un algoritmo genetico è la sua parallelizzabilità. Molti altri algoritmi di ottimizzazione multi variabile vengono eseguiti in maniera seriale e possono esplorare lo spazio delle soluzioni una direzione alla volta, e se la soluzione che trovano si rivela sub ottimale abbandonano completamente il ramo di ricerca. Gli algoritmi genetici, avendo una popolazione ampia, riescono ad esplorare lo spazio delle soluzioni più direzioni alla volta. Se un percorso si rivela un vicolo cieco viene eliminato e il lavoro continua su strade più profittevoli, ovvero che hanno maggiore probabilità di trovare buone soluzioni.

Grazie al parallelismo è possibile valutare più unità contemporaneamente, per questo motivo gli algoritmi genetici sono particolarmente adatti a risolvere problemi dove lo spazio delle soluzioni è troppo ampio per effettuare una ricerca esaustiva. La maggior parte dei problemi che ricadono in questa categoria sono quelli non lineari. In un problema lineare, la fitness di ogni componente è indipendente, così ogni miglioramento di una parte si traduce in un miglioramento dell'intero sistema. Nel mondo reale questo genere di problemi è molto raro, infatti la non linearità è la norma. Un problema è non lineare se il cambiamento di un componente influenza l'intero sistema. Spesso la somma di più cambiamenti peggiorativi si traduce nel miglioramento complessivo del sistema.

Un altro vantaggio degli algoritmi genetici è che sono in grado di ottenere buoni risultati in caso di funzione di fitness complessa, in presenza di discontinuità, rumore, variabilità nel tempo o molti massimi locali. La maggior parte dei problemi reali hanno uno spazio delle soluzioni molto vasto, impossibile da esplorare esaustivamente.

L'elemento chiave che distingue gli algoritmi genetici da tutti gli altri tipi di ricerca è il crossover. Senza di esso, ogni individuo della popolazione è solo ad esplorare lo spazio delle soluzioni e non ha alcuna informazione su ciò che le altre unità hanno scoperto. Tuttavia, tramite il crossover, si effettua lo scambio di informazioni tra buoni candidati al successo. Le unità possono beneficiare da quello che hanno appreso le altre. I cromosomi vengono mischiati e ricombinati con il possibilità di ottenere una prole che ha i punti di forza dei genitori e le debolezze di nessuno di loro.

In ultimo, una delle più importanti qualità di un algoritmo genetico è che non conosce nulla del problema che si accinge a risolvere. Al posto di utilizzare informazioni mutuate dalla ricerca precedente e guidate da una chiara volontà di risolvere il problema in un certo modo, gli algoritmi genetici non fanno assunzioni sul problema, sono liberi di esplorare in ogni direzione, anche in quella che, agli occhi di un supervisore, potrebbe sembrare la più sbagliata. I cambiamenti delle unità avvengono in maniera casuale, è compito della funzione di fitness determinare se questi cambiamenti siano positivi o negativi.

2.5 Svantaggi

Sebbene gli algoritmi genetici abbiano dimostrato di essere una strategia di risoluzione di problemi potente ed efficiente, hanno alcune limitazioni importanti. La prima e più importante considerazione da fare riguarda la rappresentazione del problema. La definizione di cromosoma deve essere robusta, non deve produrre errori o risultati senza senso in seguito a modifiche casuali.

Un altro problema riguarda la scrittura di una funzione di fitness consistente: a valori maggiori devono corrispondere soluzioni migliori. Se la funzione di fitness è scelta in maniera sbagliata, l'algoritmo genetico potrebbe non essere in grado di trovare una soluzione al problema o finire per risolvere il problema sbagliato. Quest'ultimo caso si manifesta nella tendenza dell'algoritmo genetico ad «imbrogliare» sfruttando le inconsistenze della funzione di fitness al fine di massimizzare il suo valore.

In aggiunta alla scelta di una buona funzione di fitness, gli altri parametri dell'algoritmo genetico, come la dimensione della popolazione, il tasso di mutazione e crossover, e il tipo di selezione da applicare, devono essere scelti con altrettanta cura. Ad esempio, se la popolazione è poco numerosa non riuscirà ad esplorare a sufficienza lo spazio delle soluzioni. Se la probabilità di mutazione è troppo alta, l'algoritmo genetico non riuscirà a convergere stabilmente verso una soluzione.

Un problema molto conosciuto che può verificarsi è la convergenza prematura. Se un individuo molto adatto emerge nelle prime fasi dell'evoluzione può riprodursi in maniera significativa riducendo di molto la diversità della popolazione. Questo porta l'algoritmo genetico a convergere in un minimo locale al posto di esplorare il più possibile lo spazio delle soluzioni alla ricerca del massimo globale. La soluzione a questo problema è non dare un grande vantaggio riproduttivo ad individui molto adatti.

Capitolo 3

Il gioco del Tetris

3.1 Storia

Il Tetris ha riscontrato un grande successo di pubblico fin dalla sua pubblicazione ed è classificato nel mondo come uno dei classici «puzzle game» [9, 19]. Il gioco fu realizzato da Alexey Pajitnov [16] mentre lavorava per l'Accademia delle Scienze dell'URSS di Mosca, e fu rilasciato il 6 Giugno 1984. Inizialmente l'interesse di Pajinov si soffermò sul suo puzzle game preferito: i pentamini, una versione embrionale di Tetris con pezzi composti da 5 blocchi al posto di 4. Durante lo sviluppo Pajinov sostituì i pentamini con i tetramini per ridurre la difficoltà e il gioco divenne subito molto popolare tra i suoi amici. Uno di loro, Vadim Gerasimov, realizzò il porting per PC IBM che fece diffondere a macchia d'olio il gioco all'interno dell'accademia. Poco dopo, nel 1985, Alexey distribuì una prima versione a colori del Tetris ai suoi amici al di fuori dell'università, i quali a loro volta lo diedero ai loro amici, e presto il gioco si diffuse per tutta Mosca. Poco tempo dopo, il Tetris varcò i confini dell'URSS e approdò in Europa. All'epoca non esisteva il concetto di proprietà intellettuale, e tutto quello che realizzò Pajitnov era di proprietà dello Stato, per questo motivo non ricevette nulla in cambio del suo lavoro.

La versione PC di Tetris si diffuse anche in Ungheria, in particolare a Budapest, dove venne realizzato il porting per diverse piattaforme. Successivamente, il gioco venne scoperto dalla software house inglese Andromeda che provò a contattare Pajitnov per assicurarsi i diritti per la versione PC, ma prima di riuscire nel loro intento, i diritti furono venduti alla Spectrum Holobyte. La trattativa con Pajitnov fallì e Andromeda cercò di assicurarsi i diritti con i programmatori ungheresi che ne avevano realizzato il porting per PC. Nel 1987, prima della conclusione delle trattative, la versione del Tetris per lo Spectrum della HoloByte fu rilasciata negli Stati Uniti, e comprendeva contenuti provenienti dalla versione russa [12]. La popolarità del gioco fu incredibile, il giornale Computer Gaming World definì il gioco «ingannevolmente semplice e insidiosamente coinvolgente». A questo punto i dettagli sulla proprietà

intellettuale del gioco non erano chiari, ma Andromeda riuscì ad ottenere non solo la licenza di copyright per la versione PC IBM ma anche per tutti i sistemi consumer.

Indeciso su come pubblicare il gioco e temendo la risposta del regime Sovietico, Pajitnov colse al volo l'opportunità della Perestroika e conferì i suoi diritti intellettuali al governo Sovietico per dieci anni. Nel 1988, il governo iniziò a pubblicizzare il gioco tramite un tour promozionale con il campione del mondo di Tetris Gerald Hicks. L'associazione promotrice, chiamata Elektronorgtechnica, o «Elorg» per brevità, non riceveva ancora soldi da Andromeda, nonostante quest'ultima stesse concedendo la licenza di utilizzo alla Bullet Proof Software per il mercato asiatico.

Negli anni seguenti il Tetris venne pubblicato su piattaforma GameBoy e NES della Nintendo, su piattaforma Tengen dell'Atari Games e su piattaforma Sega. Nonostante le innumerevoli versioni pubblicate, nessuna di queste compagnie ne deteneva legalmente i diritti.

Nel 1996 i diritti del gioco tornarono nelle mani di Pajitnov, che dalla pubblicazione del puzzle game non aveva guadagnato praticamente nulla. A questo punto Pajitnov fondò la Tetris Company che registrò il copyright per i diritti sul gioco negli Stati Uniti e in tutti gli altri stati del mondo. Da quel momento in poi vennerò intentate molte cause contro società che utilizzavano il concept di Tetris nei loro giochi come ad esempio BioSocia che dovette rimuovere dal mercato «Blockes» e Xio che vide rimossa la sua app «Mino» dallo store di Android.

3.2 Caratteristiche e scopo del gioco

I «Tetramini» sono le forme geometriche composte da quattro blocchi quadrati scelte per il gioco del Tetris. Una sequenza casuale di tetramini cade in uno schema di gioco che ha solitamente le dimensioni di 10x20 blocchi. L'obbiettivo del gioco è quello di spostare e ruotare questi tetrimini per completare una o più linee orizzontali. Quando si raggiunge questo obbiettivo, le linee scompaiono e tutto quello che si trova sopra di loro si abbassa del numero di linee completate. Ogni dieci linee completate il gioco avanza di livello e i pezzi cadono più velocemente. Il gioco termina quando la pila di blocchi raggiunge il tetto dello schema e un nuovo tetramino non può farsi strada.

Il giocatore vede, oltre al pezzo presente nello schema di gioco, il tetramino successivo. In alcune implementazioni è possibile vedere anche più avanti nella catena di pezzi che arriveranno oppure è possibile conservare un tetramino per utilizzarlo in futuro.

3.2.1 I Tetramini

La versione originale di Pajitnov, realizzata per il computer Elektronika 60, utilizzava delle parentesi verdi per rappresentare i blocchi. Le versioni iniziali di Tetris

per il Game Boy della Nintendo utilizzavano grafica monocromatica o in scala di grigi. Dopo l'avvento del colore negli schermi delle console portatili, si diffusero diverse colorazioni. Nella figura 3.1 vengono mostrati i colori originali della versione Tetris Worlds.

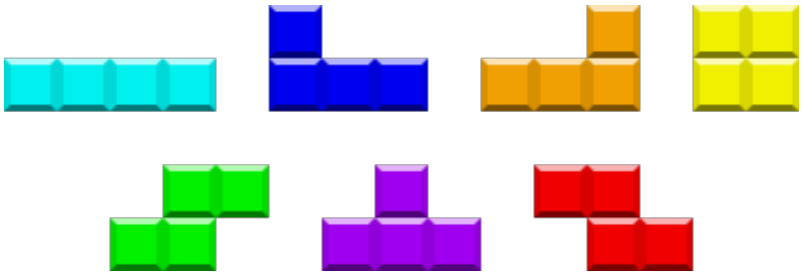


Figura 3.1: Le colorazioni ufficiali dei pezzi del Tetris

Le sette combinazioni di tetramini sono chiamate come le corrispondenti lettere dell'alfabeto che più assomigliano alla forma del pezzo. Le strutture dei tetramini sono il risultato di tutte le possibili combinazioni che si possono ottenere disponendo 4 quadrati, ciascuno dei quali ha almeno uno spigolo in comune con almeno uno degli altri tre quadrati.

Tutti i pezzi possono completare righe singole o doppie; I, L, e J possono completare anche righe triple e solo la I può completare quattro righe in una sola volta.

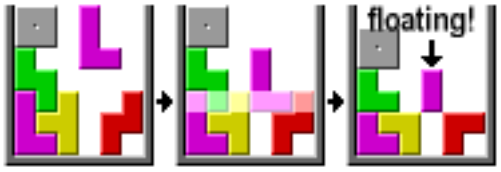
3.2.2 Punteggio

Il sistema di punteggio varia da versione a versione del Tetris [18]. Le prime implementazioni assegnavano punti al giocatore in funzione del numero di tetramini posizionati. La maggior parte delle versioni successive premiano il giocatore in base al numero di linee compleatate e al livello raggiunto. Molto spesso è disponibile anche la «discesa rapida», ovvero è possibile ottenere dei punti bonus facendo scendere il pezzo senza aspettare che cada da solo.

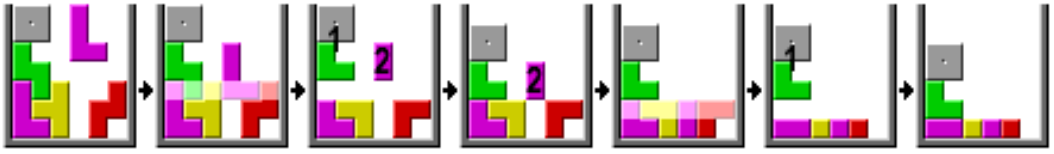
3.2.3 Varianti

Una prima caratteristica che contraddistingue le varianti del Tetris dall'originale è la gestione della gravità. Nella versione tradizionale, dopo il completamento di una linea, i blocchi possono rimanere sospesi al di sopra di spazi vuoti, contrariamente alle leggi della gravità. In molte varianti del Tetris i blocchi si comportano come corpi rigidi collegati tra loro: se uno non è sorretto da altri cade negli spazi obbedendo alle leggi della gravità. Questa regola apre il gioco a nuove strategie, ad esempio è possibile sfruttare la reazione a catena per riempire ulteriori spazi e aggiungere altre linee a quelle completate. Un esempio di questa regola è visibile nella figura 3.2 che

mette a confronto la versione originale a quella con gravità. Nell'implementazione di Tetris utilizzata come ambiente di evoluzione per l'intelligenza artificiale viene usata la versione senza gravità al fine di rimanere aderenti alla versione originale del gioco.



(a) Versione originale



(b) Versione con gravità

Figura 3.2: Gestione della gravità nel Tetris

3.3 Analisi complessità

Da quando il Tetris è stato pubblicato ad oggi, numerosi matematici [2, 3, 11] si sono dedicati all'interpretazione analitica del gioco e alla definizione della sua complessità in senso matematico. È stato dimostrato [4, 5] come una partita di Tetris si conclude certamente con una sconfitta del giocatore, in quanto esistono sequenze di tetramini che portano sicuramente alla perdita della partita.

Da questo risultato si evince che non è possibile giocare all'infinito, infatti esiste un N tale che, una sequenza di N pezzi S e Z, alternati, rende inevitabile la sconfitta. In una sequenza infinita casuale di tetramini verranno estratte tutte le combinazioni possibili di pezzi, quindi anche la sequenza che porta il giocatore a perdere sicuramente la partita. Per questo motivo si può ipotizzare che i pezzi S e Z siano più «difficili» da gestire, per questo, nei prossimi capitoli si cercherà di stabilire le relazioni di complessità tra i singoli pezzi. Inoltre si osserveranno gli adattamenti strategici dell'intelligenza artificiale in reazione alla variazione di distribuzione dei pezzi.

In senso matematico, si può dimostrare che i seguenti problemi sono NP-Completi

- Massimizzare il numero di righe completate a partire da una certa sequenza di pezzi
- Massimizzare il numero di pezzi giocati in una partita prima della sconfitta
- Massimizzare il numero di gruppi di quattro linee completati simultaneamente

- Minimizzare l'altezza dello schema riempito data una certa sequenza di pezzi

Per queste ragioni è interessante realizzare un'intelligenza artificiale che sia in grado di gestire l'intrinseca complessità del Tetris. Dove un algoritmo analitico non riesce nel suo intento, l'approccio genetico si rivela un'ottima strada percorribile come si vedrà nei prossimi capitoli.

Capitolo 4

Il modello

4.1 Introduzione

In questo capitolo si scenderà nel dettaglio del modello dei dati utilizzato nel processo evolutivo dell'intelligenza artificiale per il gioco del Tetris. Come specificato nella sezione 2.3, un algoritmo genetico ha bisogno di diversi ingredienti:

1. Codifica del cromosoma
2. Funzione di Fitness
3. Selezione e ricombinazione

Il cromosoma è codificato come un vettore di numeri reali, ogni elemento è un gene del cromosoma. I valori dei geni non sono limitati ad un intervallo ma possono assumere con continuità qualsiasi valore reale.

Nella sezione 4.2 si analizzerà la composizione del cromosoma e gli indicatori scelti per modellizzare il comportamento dell'intelligenza artificiale. Si concentrerà l'attenzione sul significato semantico di questi indicatori, infatti il loro valore può essere interpretato come segnale positivo o negativo riguardante l'azione da intraprendere.

4.2 Cromosoma e funzione di fitness

La modellizzazione del comportamento di un'intelligenza artificiale è uno degli aspetti più critici nella realizzazione di un algoritmo genetico. Nel caso del Tetris è necessario trovare un modo per stabilire una relazione di precedenza tra le mosse a disposizione. Ad ogni turno, per decidere la mossa da compiere, è opportuno valutare tutti i possibili posizionamenti e rotazioni del pezzo della mossa corrente. Per ogni combinazione di posizione/rotazione si calcolano alcuni indicatori che rappresentano l'effetto di quella mossa sulla partita. Lo score della mossa si può indicare come:

$$S_i = \sum_{j=0}^{N} p_j g_j \tag{4.1}$$

- p_j è il valore del parametro j-esimo calcolato sulla combinazione posizione/rotazione del pezzo corrente
- g_j è il valore del gene corrispondente al parametro j-esimo
- N è il numero di parametri da calcolare e anche il numero di geni presenti nel cromosoma

La mossa che compirà l'unità sarà $\max S_i$, ovvero il massimo della combinazione lineare del prodotto tra parametri e geni. Nella tabella 4.1 viene mostrato un esempio di questo procedimento e la mossa che l'intelligenza artificiale compirà sarà quella con lo score più alto (19;2):

	Height	ETP	ETW	Gaps	Lines
Gene	-0,405709453	0,616938309	0,783830552	-0,490775875	0,662098138

(a) Cromosoma dell'unità di esempio

Posizione	Rotazione	Height	ETP	ETW	Gaps	Lines	Score
0	0	2	3	1	0	0	1,8232265
0	1	1	2	1	1	0	1,1212218
...	...	...	...	...	...	...	...
19	**2**	**3**	**1**	**3**	**0**	**1**	**2,413399**

(b) Valori dei parametri di alcune mosse disponibili

Tabella 4.1: Esempio calcolo score di una mossa

Tutti i parametri sono calcolati sulle differenze tra configurazioni dello schema di gioco. La mossa in analisi è da immaginare come un segnale inviato nel sistema per ottenere una perturbazione. I parametri cercano di misurare le caratteristiche di questa variazione, fornendo all'intelligenza artificiale gli strumenti per gestire la partita.

Riprendendo quanto detto nella sezione 4.1 è possibile interpretare i valori dei parametri p_j come segnali positivi e negativi che guidano l'intelligenza artificiale nella scelta della migliore mossa. Nelle sezioni seguenti verranno analizzati nel dettaglio questi parametri, e verrà prestata particolare attenzione al loro significato semantico all'interno del processo decisionale dell'intelligenza artificale.

4.2.1 Height

Il parametro Height rappresenta l'incremento dell'altezza dello schema. Nella seguente formula, quando si parla di altezza della colonna si intende il numero di blocchi riempiti da pezzi già posizionati:

$$h_t = h_{t-1} - \max_{i=0}^{C} c_i$$

- t rappresenta il singolo pezzo nella sequenza della partita
- h_{t-1}è l'altezza dello schema di gioco, intesa come l'ordinata della colonna più alta, precedente al posizionamento del pezzo corrente
- c_i è l'altezza della colonna i

Il gene che governa questo parametro è generalmente negativo, infatti per prolungare la durata della partita è necessario tenere sotto controllo l'altezza dello schema evitando di impilare un numero eccessivo di pezzi.

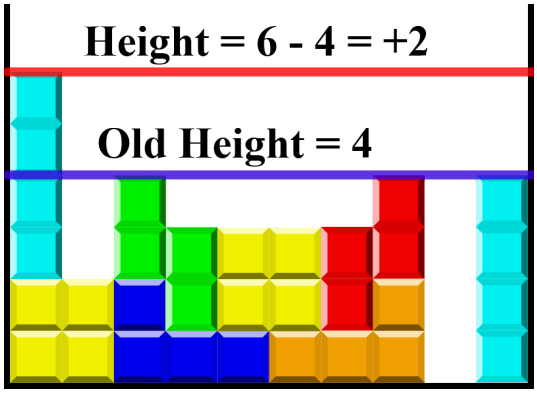


Figura 4.1: Visualizzazione parametro Height del pezzo I (colonna 1)

4.2.2 ETP e ETW

I parametri ETP (Edges Touching Pieces) ed ETW (Edges Touching Walls) si possono definire come parametri di contatto o di adiacenza. Sono utili sia per definire la tipologia di contatto del pezzo con i blocchi sottostanti, sia per regolare la forma che assume lo schema. Se ETP > ETW l'intelligenza artificiale tenderà a preferire l'aggiunta di pezzi nella parte centrale dello schema, viceversa se ETW > ETP prediligerà le colonne adiacenti alle pareti, facendo assumere allo schema una forma paraboloide. Il rapporto trà ETW ed ETW controlla la distribuzione dei pezzi e, come si vedrà nei risultati degli esperimenti, nella maggior parte dei casi, l'intelligenza artificiale preferisce massimizzare il contatto con le pareti al fine di lasciare più libertà di movimento al centro dello schema di gioco.

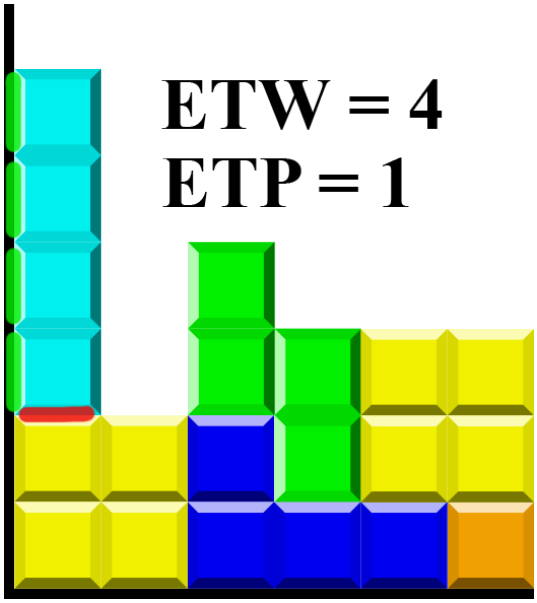


Figura 4.2: Visualizzazione dei parametri ETP ed ETW del pezzo I (colonna 1)

4.2.3 Gaps

Il parametro Gaps indica il numero di «spazi» o «buchi» che si vengono a creare con la mossa corrente. Come si vedrà in seguito è una metrica molto importante ed è tra i geni dominanti in quasi tutti gli esperimenti condotti. Il suo valore è sempre negativo, infatti è buona norma ridurre al minimo gli spazi vuoti per favorire il completamento delle linee.

Differentemente da quello che si potrebbe pensare, lasciare degli spazi vuoti nello schema non è tabù. Non è necessario impilare i pezzi senza lasciare alcuno spazio per completare un grande numero di mosse. L'analisi di questo tipo di strategia verrà condotta nei prossimi capitoli, nei quali si scoprirà che l'intelligenza artificiale non impara ad evitare gli spazi ma a gestirli nella maniera più efficiente.

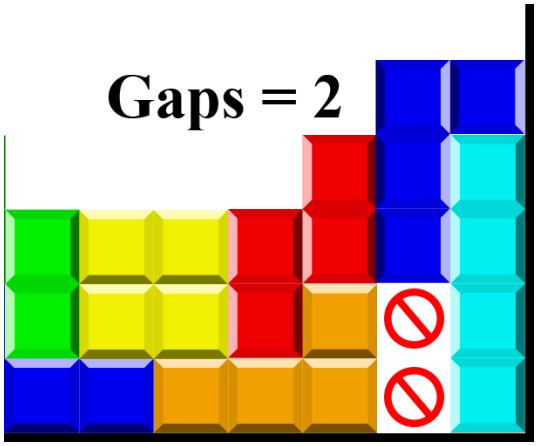


Figura 4.3: Visualizzazione del parametro Gaps del pezzo J (colonna 6)

4.2.4 Lines

La metrica Lines rappresenta il numero di linee completate. Specularmente al parametro Gaps, il valore di Lines è sempre grande e positivo. Questo indica che massimizzare il numero di linee completate per pezzo è un sotto obbiettivo molto importante per l'intelligenza artificiale e generalmente è il segnale più positivo riguardante la correttezza di una mossa.

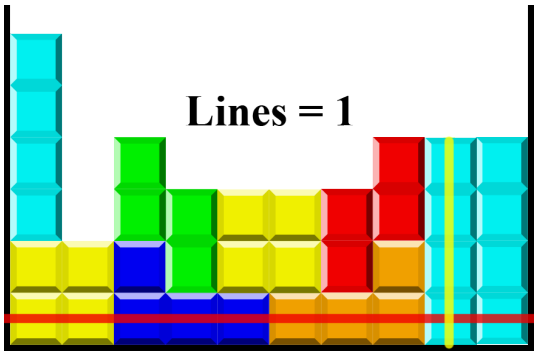


Figura 4.4: Visualizzazione del parametro Lines del pezzo I (colonna 9)

4.3 Selezione e ricombinazione

Il processo di evoluzione dell'intelligenza artificiale comprende diversi passaggi. La popolazione iniziale è costituita da 100 unità e per ogni posto disponibile viene selezionata l'unità più forte prelevata da un pool di 10 cromosomi con materiale genetico casuale. In questo modo il punto di partenza dell'evoluzione comprende geni di qualità migliore rispetto ad insieme totalmente casuale. Per ogni generazione vengono eseguiti i seguenti passi

1. Valutazione funzione di fitness per ogni unità: per moderare l'effetto dell'intrinseca variabilità della sequenza di pezzi vengono giocate 3 partite. I valori minimo, massimo e medio vengono registrati. Dopodiché le unità vengono ordinate per fitness media in ordine discendente.

2. Selezione popolazione elitaria: viene selezionato il miglior 5%. Queste unità faranno parte della prossima generazione senza subire nessuna ricombinazione o mutazione.

3. Selezione popolazione per ricombinazione: viene selezionato il miglior 20%. Queste unità si ricombineranno tra loro al fine di ottenere il nuovo 95% della popolazione.

   (a) La scelta delle unità da accoppiare è effettuata scegliendo casualmente la prima unità tra quelle selezionate e la seconda unità tra quelle con fitness

maggiore della sua. In questo modo le unità con fitness migliore diffondono i loro geni ma senza intaccare la diversità genetica della popolazione.

(b) La probabilità di crossover è fissata al 100% in modo da garantire che tutte le unità selezionate vengano ricombinate. Dopo la ricombinazione le unità tornano nella popolazione, questo permette alle unità di accoppiarsi eventualmente molteplici volte.

(c) Il tipo di crossover è importato a SwapBlend, ovvero c'è una probabilità pari al 10% che i geni vengano scambiati e del 10% che venga selezionata la media algebrica. Maggiori dettagli sull'implementazione di SwapBlend sono presenti nella sottosezione 5.3.4.1.

(d) La mutazione è di tipo DeltaAbsolute e ha un'incidenza pari al 2%. Questo tipo di mutazione aggiunge algebricamente un delta casuale compreso da -0.5 e 0.5 al gene in questione.

(e) Per garantire che non ci siano doppioni tra le unità della nuova generazione è disattivato il flag AllowDuplicates. Se un cromosoma risultato della ricombinazione è già presente nella popolazione significa che il suo materiale genetico non fornisce ulteriori informazioni rispetto a quelle già esistenti. Per questo motivo viene imposta una mutazione casuale al cromosoma duplicato al fine di incrementare la diversità genetica della popolazione.

4. La popolazione della nuova generazione è stata costituita. A questo punto il processo si compie nuovamente partendo dal punto 1 fino al raggiungimento della condizione di terminazione.

4.3.1 Obbiettivi degli esperimenti

Il primo obbiettivo di questo lavoro di tesi è stato stabilire per quanto tempo è in grado di sopravvivere l'intelligenza artificiale se lasciata libera di giocare senza un limite di pezzi. Questo tipo di ottimizzazione è molto indicativa perché permette di capire se le unità sono in grado di gestire la grande diversità di sequenze di pezzi che offre il gioco del Tetris. Il secondo obbiettivo è quello di massimizzare il rapporto tra il numero di pezzi giocati e il punteggio. Il terzo obbiettivo è quello di valutare la complessità dei tetramini variando la frequenza di estrazione del generatore di numeri casuali.

4.3.1.1 Ottimizzazione per numero di pezzi

Nel primo esperimento sono state condotte una serie di prove con lo scopo di stabilire quanto a lungo l'intelligenza artificiale è in grado di sopravvivere. Il limite a questo tipo di evoluzione è dato dal numero massimo di generazioni, infatti per ren-

dere l'esperimento praticabile è necessario fissare per quante volte ripetere il processo evolutivo.

L'ottimizzazione per numero di pezzi è tra le computazioni che richiedono più tempo, in quanto le unità riescono molto presto a sopravvivere diversi milioni di pezzi. Come si vedrà nel capitolo 5 sono state attuate una serie di ottimizzazioni per rendere il processo di simulazione sufficientemente veloce per rientrare nella finestra temporale a disposizione. Nonostante questo, il tempo medio di simulazione di una generazione per questo esperimento è di circa 8 ore.

4.3.1.2 Ottimizzazione per punteggio

Un altro obbiettivo per l'evoluzione dell'intelligenza artificiale è quello di massimizzare i punti per pezzo. Dato un numero finito di pezzi a disposizione si cerca di ottenere il punteggio maggiore possibile. Come si vedrà nel capitolo 6 la strategia di gioco con questo tipo di funzione obbiettivo sarà sensibilmente differente dall'ottimizzazione per numero di pezzi.

Nel gioco del Tetris si ottengono punti in base al numero di linee completate con il posizionamento di un pezzo. Il rapporto tra i punti guadagnati e le linee completate non è lineare:

Linee	1	2	3	4
Punti	10	30	60	100
Incremento		20	30	40

Tabella 4.2: Punti per linee completate

Questo comporta che l'utilità marginale è sempre crescente e l'intelligenza artificiale otterrà una ricompensa maggiore più linee riuscirà a completare.

4.3.1.3 Valutazione complessità Tetramini

L'analisi di complessità dei tetramini è un obbiettivo molto difficile, come si è visto nella sezione 3.3 il Tetris è difficile anche solo da approssimare. La valutazione che si intende ottenere riguarda la difficoltà che l'intelligenza artificiale incontra nella gestione di un determinato pezzo. Per raggiungere questo obbiettivo la distribuzione di probabilità dei tetramini viene modificata in modo da valutare l'impatto sulla strategia dell'intelligenza artificiale.

Dall'analisi dei risultati di questo esperimento sarà possibile evidenziare analogie e differenze tra le strategie messe in atto dalle unità per gestire le diverse combinazioni di pezzi. A partire da queste informazioni si possono formulare congetture sui rapporti di complessità tra i diversi tetramini.

Capitolo 5

Architettura software

5.1 Introduzione

La piattaforma software scelta per questo progetto è la tecnologia .NET di Microsoft. Questa scelta ha permesso lo sviluppo dell'implementazione del Tetris in maniera rapida ed efficiente traendo vantaggio dalle classi che il framework mette a disposizione. Grande attenzione è stata posta sull'implementazione parallela del processo evolutivo, in quanto caratteristica necessaria per rendere praticabile il tipo di ricerca in esame. L'algoritmo genetico è stato sviluppato ad hoc, in quanto per ogni problema da risolvere è necessario implementare delle caratteristiche diverse. Nella sezione 5.2 si vedrà nel dettaglio l'implementazione software del Tetris e si metteranno a confronto due diversi modelli di rappresentazione dello schema di gioco in ottica di ottimizzazione. Inoltre nella sezione 5.3 verranno evidenziati i punti importanti dell'implementazione dell'algoritmo genetico con particolare attenzione alla scelta dei parametri evolutivi.

5.2 Implementazione Tetris

L'implementazione software del gioco del Tetris non è particolarmente complessa. Il requisito più importante per il caso d'uso in esame è senza dubbio la velocità di esecuzione, perchè da essa dipende il tempo di valutazione della funzione di fitness. Il problema principale che è necessario affrontare è la rappresentazione dello schema di gioco, infatti da questo dipendono tutte le procedure che si occupano di posizionare i tetramini e calcolare i parametri Height, ETP, ETW, Gaps e Lines.

Prima di effettuare ogni mossa è necessario valutare tutti i possibili posizionamenti e rotazioni del tetramino. In realtà nel Tetris il giocatore è a conoscenza non solo del pezzo corrente ma anche del pezzo successivo. Sfruttando questa caratteristica l'intelligenza artificiale è in grado di pianificare anche la mossa futura. Per implementare la valutazione della mossa è stata realizzata la classe RollbackableTe-

trisGame che deriva da FixedHistory che incapsula uno stack di dimensione fissata. Gli step per valutare quale mossa compiere sono elencati di seguito:

1. Per ogni combinazione di posizione / rotazione del tetramino
2. Push nello stack di una copia dello stato corrente dello schema
3. Posiziona il tetramino nello schema
4. Lo score della mossa è la combinazione lineare tra i parametri (Height, ETP, ETW, Gaps, Lines) e i valori dei geni corrispondenti
5. Pop dallo stack dello stato dello schema in modo da ripristinare lo stato del punto 2
6. Ripeti da 2 con la successiva combinazione

Per estendere il processo sopra descritto per gestire anche il pezzo successivo è necessario moltiplicare le combinazioni considerando anche il secondo pezzo. In questo modo si valuteranno gli effetti combinati del posizionamento di due pezzi.

Al fine di ottenere una velocità ancora maggiore di valutazione della fitness sono state implementate delle classi gemelle a quelle standard ma con il prefisso Fast, come ad esempio FastRollbackableTetrisGame che si comportano allo stesso modo delle classi originali ma senza controlli di integrità al fine di aumentare ulteriormente la velocità di esecuzione.

Sono state realizzate due versioni del Tetris: la prima gestisce lo schema di gioco come una matrice bidimensionale di variabili booleane e la seconda come array di numeri interi. Nella prossima sezione si analizzeranno nel dettaglio queste due implementazioni sottolineando le differenze prestazionali.

5.2.1 Schema booleano

La prima implementazione del Tetris modellizzava lo schema di gioco come un array bidimensionale di variabili booleane. La realizzazione di questa implementazione ha richiesto minor tempo ma ha mostrato subito i suoi limiti in fatto di prestazioni. Un esempio di quanto detto è evidente nella procedura PlacePiece che si occupa di posizionare un pezzo nello schema:

Listing 5.1: Funzione PlacePiece

```
ï»¿public void PlacePiece(int iPosition, int iRotation)
{
    _iPosition = iPosition;
    _iRotation = iRotation;
```

```
    _CurrentPiece = StandardPieces.Pieces[_lstNextPieces.
        First.Value][_iRotation];

    FindLowestHeight();

    if (Height == -1)
    {
        Status = Status.Ended;
        IsPlaying = false;

        return;
    }

    GetEdgesTouching();

    PutPiece();
    RemoveCompletedLines();

    GetGaps();

    _lstNextPieces.AddLast(rand.Next(StandardPieces.Pieces.
        Count));
    _lstNextPieces.RemoveFirst();

    this.Moves++;

    return;

}
```

Questa funzione ha il compito di effettuare tutte le operazioni necessarie per posizionare un pezzo in una colonna con una certa rotazione. Come primo step calcola l'altezza alla quale posizionare il pezzo utilizzando la funzione FindLowestHeight dopodiché aggiunge il tetramino allo schema e rimuove le eventuali linee completate. Le funzioni GetEdgesTouching e Gaps calcolano ETP, ETW e Gaps, mentre FindLowestHeight e RemoveCompletedLines calcolano Height e Lines.

Nelle prossime sottosezioni si andrà nel dettaglio dell'implementazione di tutte le funzioni che vengono chiamate in PlacePiece.

5.2.1.1 FindLowestHeight

La procedura FindLowestHeight, che si occupa di ottenere l'altezza alla quale è possibile posizionare il pezzo, è critica dal punto di vista delle performances:

Listing 5.2: Funzione FindLowestHeight e CanPlacePiece

```
ï»¿protected void FindLowestHeight()
{
    int iMaxHeight = StaticSettings.TetrisHeight -
        _CurrentPiece.Height;

    if (!CanPlacePiece(iMaxHeight - 1))
    {
        Height = -1;
        return;
    }

    for (Height = iMaxHeight - 2; Height >= 0 &&
        CanPlacePiece(Height); Height--) ;

    Height++;
}

private bool CanPlacePiece(int iHeight)
{
    int iLengthX = _CurrentPiece.Width;
    int iLengthY = _CurrentPiece.Height;

    for (var x = 0; x < iLengthX; x++)
    {
        for (var y = 0; y < iLengthY; y++)
        {
            if (_CurrentPiece.Content[x, y] && Field[x +
                _iPosition, y + iHeight])
            return false;
        }
    }

    return true;
}
```

Partendo dall'alto si controlla se è possibile posizionare il pezzo a quella determinata altezza, quando si incontra un ostacolo significa che la minima altezza disponibile era quella dell'iterazione precedente. Il numero di cicli for annidati aumenta la complessità computazionale della procedura rendendola inefficiente.

5.2.1.2 RemoveCompletedLines

La procedura RemoveCompletedLines si occupa di rimuovere dallo schema di gioco le linee completate:

Listing 5.3: Funzione RemoveCompletedLines

```
ï»¿private void RemoveCompletedLines()
{
    int x, y, n, h;
    bool bOk;
    bool bMinimum;

    DeltaLinesCompleted = 0;

    do
    {
        bMinimum = true;

        for (y = 0; y < StaticSettings.TetrisHeight; y++)
        {
            bOk = true;

            for (x = 0; x < StaticSettings.TetrisWidth; x++)
            {
                if (!Field[x, y])
                {
                    bOk = false;
                    break;
                }
            }

            if (bOk)
            {

                for (h = y + 1; h < StaticSettings.TetrisHeight
                    + 1; h++)
```

```
            {
                if (h == StaticSettings.TetrisHeight)
                {
                    for (n = 0; n < StaticSettings.TetrisWidth
                        ; n++)
                    {
                        Field[n, h - 1] = false;
                    }
                }
                else
                {
                    for (n = 0; n < StaticSettings.TetrisWidth
                        ; n++)
                    {
                        Field[n, h - 1] = Field[n, h];
                    }
                }
            }

            bMinimum = false;
            DeltaLinesCompleted++;

            break;
        }
    }
} while (!bMinimum);

LinesCompleted += DeltaLinesCompleted;
}
```

Questa procedura è particolarmente onerosa perchè dopo essersi assicurati del completamento della linea è necessario abbassare tutto lo schema soprastante. L'implementazione con matrice di variabili booleane necessita di diversi cicli for annidati che si ripetono fino a che non sono state rimosse tutte le linee completate. Si verà nella sottosezione 5.2.2 come è possibile riscrivere questo codice in maniera di gran lunga più efficiente.

5.2.1.3 GetEdgesTouching

La procedura GetEdgesTouching ha il compito di calcolare i parametri ETP ed ETW. La scelta di non separare il calcolo dei due parametri in due funzioni distinte

deriva dal fatto che la procedura è analoga, è semplicemente necessario distinguere gli spigoli adiacenti se sono a contatto con altri blocchi o con il bordo dello schema.

Listing 5.4: Funzione GetEdgesTouching

```
ï»¿private void GetEdgesTouching()
{
    int x, y;

    int iLengthX = _CurrentPiece.Width;
    int iLengthY = _CurrentPiece.Height;

    for (x = 0; x < iLengthX; x++)
    {
        for (y = 0; y < iLengthY; y++)
        {
            if (_CurrentPiece.Content[x, y])
            {
                int XCoord = x + _iPosition;
                int YCoord = y + Height;

                switch (XCoord)
                {
                    case 0:
                        EdgesTouchingWalls++;
                    break;

                    case StaticSettings.TetrisWidth - 1:
                        EdgesTouchingWalls++;
                    break;
                }

                if (YCoord == 0) EdgesTouchingWalls++;

                // X
                if (XCoord > 0 && Field[XCoord - 1, YCoord])
                    EdgesTouchingPieces++;
                if ((XCoord < StaticSettings.TetrisWidth - 1) &&
                     Field[XCoord + 1, YCoord])
                    EdgesTouchingPieces++;
```

```
                // Y
                if (YCoord > 0 && Field[XCoord, YCoord - 1])
                    EdgesTouchingPieces++;
                if ((YCoord < StaticSettings.TetrisHeight - 1)
                   && Field[XCoord, YCoord + 1])
                    EdgesTouchingPieces++;

            }
        }
    }
}
```

La procedura scorre tutti i blocchi del pezzo corrente e ne conta le adiacenze incrementando rispettivamente ETP ed ETW a seconda della colonna:

- Se la colonna è la prima o l'ultima e si tratta di uno spigolo adiacente ad una parete, si incrementa ETW
- Altrimenti se lo spigolo è a contatto con un blocco si incrementa ETP

5.2.1.4 GetGaps

La funzione che si occupa di contare i blocchi vuoti è GetGaps:

Listing 5.5: Funzione GetGaps

```
ï»¿private void GetGaps()
{
    int iTemp = Gaps;
    int iWidth = _CurrentPiece.Width;
    int iMaxY = StaticSettings.TetrisHeight - 1;
    bool bFlag;

    Gaps = 0;

    for (var x = 0; x < StaticSettings.TetrisWidth; x++)
    {
        bFlag = false;

        for (var y = iMaxY; y >= 0; y--)
        {
            if (Field[x, y])
                bFlag = true;
```

```
            else
                if (bFlag) Gaps++;

        }
    }

    Gaps = iTemp - Gaps;

}
```

Per contare il numero di Gaps dello schema è sufficiente scorrere le colonne da sinistra verso destra e le righe dall'alto verso il basso contando gli spazi che sono sormontati da blocchi pieni.

5.2.1.5 PutPiece

Per posizionare un pezzo nello schema viene utilizzata la funzione PutPiece:

Listing 5.6: Funzione PutPiece

```
ï»¿private void PutPiece()
{
    int iPiece = _lstNextPieces.First.Value;

    int x, y;
    int iLengthX = _CurrentPiece.Width;
    int iLengthY = _CurrentPiece.Height;

    for (x = 0; x < iLengthX; x++)
    {
        for (y = 0; y < iLengthY; y++)
        {
            if (_CurrentPiece.Content[x, y])
            {
                Field[x + _iPosition, y + Height] = true;
            }
        }
    }
}
```

La funzione PutPiece viene chiamata una volta che l'altezza del pezzo è stata calcolata con FindLowestHeight. Il funzionamento è relativamente semplice: si scorrono i blocchi del pezzo e si copiano nello schema di gioco con coordinate traslate.

Nella sottosezione 5.2.2 si vedrà un'implementazione molto più efficiente di questa funzione.

5.2.2 Schema binario

Per colmare la mancanze della prima implementazione del Tetris è stato necessario sviluppare un differente modello dei dati. Lo schema binario colma le mancanze dello schema booleano in fatto di velocità di esecuzione, rendendo percorribile la strada degli algoritmi genetici. Lo schema di gioco è organizzato come un vettore di numeri interi, ogni numero rappresenta una colonna e i singoli bit del numero rappresentano le righe.Utilizzando operatori binari per manipolare i numeri naturali che costituiscono righe e colonne è possibile ottenere un sensibile aumento della velocità di esecuzione. Nella sottosezione 5.2.3 verranno mostrate nel dettaglio le prestazioni dei due modelli a confronto sui tempi di esecuzione di diversi esperimenti.

5.2.2.1 Funzioni accessorie

Il framework .NET offre molte funzioni già pronte per manipolare i numeri binari. Il problema che è sorto durante la scrittura dell'implementazione è stata la necessità di avere una funzione Math.Log con base 2 che non diventasse un collo di bottiglia. Per questo motivo sono state impiegate delle funzioni ad hoc sia per il logaritmo in base 2 sia per contare il numero di bit di un numero intero:

Listing 5.7: Funzioni accessorie ad hoc

```
ï»¿
// Utility da http://graphics.stanford.edu/~seander/bithacks
    .html

[MethodImpl(MethodImplOptions.AggressiveInlining)]
private static int CountBits(uint number)
{
    number = number - ((number >> 1) & 0x55555555);
    number = (number & 0x33333333) + ((number >> 2) & 0
        x33333333);
    return (int)(((number + (number >> 4) & 0xF0F0F0F) * 0
        x1010101) >> 24);
}

static readonly int[] tab32 = {
      0,  9,  1, 10, 13, 21,  2, 29,
     11, 14, 16, 18, 22, 25,  3, 30,
      8, 12, 20, 28, 15, 17, 24,  7,
```

```
    19, 27, 23,  6, 26,  5,  4, 31};

[MethodImpl(MethodImplOptions.AggressiveInlining)]
static int log2(uint value)
{
    value |= value >> 1;
    value |= value >> 2;
    value |= value >> 4;
    value |= value >> 8;
    value |= value >> 16;
    return tab32[value * 0x07C4ACDD >> 27];
}
```

Queste funzioni fanno uso di array e operazioni binarie per ottenere le massime prestazioni possibili, inoltre l'attributo AggressiveInlining consiglia al compilatore di effettuare l'inlining della funzione espandendola nel luogo di chiamata.

5.2.2.2 Ottimizzazioni

Per ottenere prestazioni elevate è buona pratica ridurre al minimo i conti eseguiti durante la fase di runtime. Per questo motivo nell'implementazione binaria del Tetris si fa largo uso di parametri precalcolati. La struttura dati che ospita i tetramini comprende anche informazioni riguardanti altezza, larghezza, adiacenze e impronta della base. Nel listato 5.8 viene mostrato uno spaccato di queste informazioni riguardanti il tetramino T:

Listing 5.8: Informazioni sulle configurazioni del pezzo T

```
ï»¿[...]

new List<PieceRotation>()
{
    new PieceRotation
    {
        Width = 2,
        Height = 3,
        Values = new uint[] {1 + 2 + 4, 0 + 2 + 0},
        LowestBits = new[] {0, 1},
        BoundaryCheckItems = new[]
      {
          new BoundaryCheckItem {x = 1, y = 0, count = 2},
          new BoundaryCheckItem {x = 2, y = 1, count = 1},
```

```
            new BoundaryCheckItem {x = -1, y = 2, count = 1},
            new BoundaryCheckItem {x = -1, y = 1, count = 1},
            new BoundaryCheckItem {x = -1, y = 0, count = 1},
            new BoundaryCheckItem {x = 0, y = -1, count = 1}
        }
    },
    new PieceRotation
    {
        Width = 2,
        Height = 3,
        Values = new uint[] {0 + 2 + 0, 1 + 2 + 4},
        LowestBits = new[] {1, 0},
        BoundaryCheckItems = new[]
        {
            new BoundaryCheckItem {x = 0, y = 0, count = 2},
            new BoundaryCheckItem {x = 1, y = -1, count = 1},
            new BoundaryCheckItem {x = 2, y = 0, count = 1},
            new BoundaryCheckItem {x = 2, y = 1, count = 1},
            new BoundaryCheckItem {x = 2, y = 2, count = 1},
            new BoundaryCheckItem {x = -1, y = 1, count = 1}
        }
    },
    new PieceRotation
    {
        Width = 3,
        Height = 2,
        Values = new uint[] {1 + 0, 1 + 2, 1 + 0},
        LowestBits = new[] {0, 0, 0},
        BoundaryCheckItems = new[]
        {
            new BoundaryCheckItem { x = 0, y = -1, count = 1},
            new BoundaryCheckItem { x = 1, y = -1, count = 1},
            new BoundaryCheckItem { x = 2, y = -1, count = 1},
            new BoundaryCheckItem { x = 3, y = 0, count = 1},
            new BoundaryCheckItem { x = -1, y = 0, count = 1}
        }
    },
    new PieceRotation
    {
        Width = 3,
```

```
            Height = 2,
            Values = new uint[] {0 + 2, 1 + 2, 0 + 2},
            LowestBits = new[] {1, 0, 1},
            BoundaryCheckItems = new[]
        {
            new BoundaryCheckItem { x = 0, y = 0, count = 2},
            new BoundaryCheckItem { x = 1, y = -1, count = 1},
            new BoundaryCheckItem { x = 2, y = 0, count = 2},
            new BoundaryCheckItem { x = 3, y = 1, count = 1},
            new BoundaryCheckItem { x = -1, y = 1, count = 1}
        }
    },
},

[...]
```

L'oggetto PieceRotation rappresenta una possibile rotazione di un certo pezzo, i campi che lo compongono hanno il seguente significato:

- Width: Larghezza del pezzo
- Height: Altezza del pezzo
- Values: Vettore che rappresenta i blocchi del pezzo, con lo stesso metodo di rappresentazione binaria dello schema di gioco
- LowestBits: Vettore che fornisce informazioni sull'impronta della base. Ogni valore contenuto indica l'altezza del primo blocco in quella colonna
- BoundaryCheckItems: Array di oggetti BoundaryCheckItem che incapsula le informazioni sull'adiacenza.

Nell'immagine 5.1 sono riassunte le informazioni del listato 5.8 per tutti i tetramini:

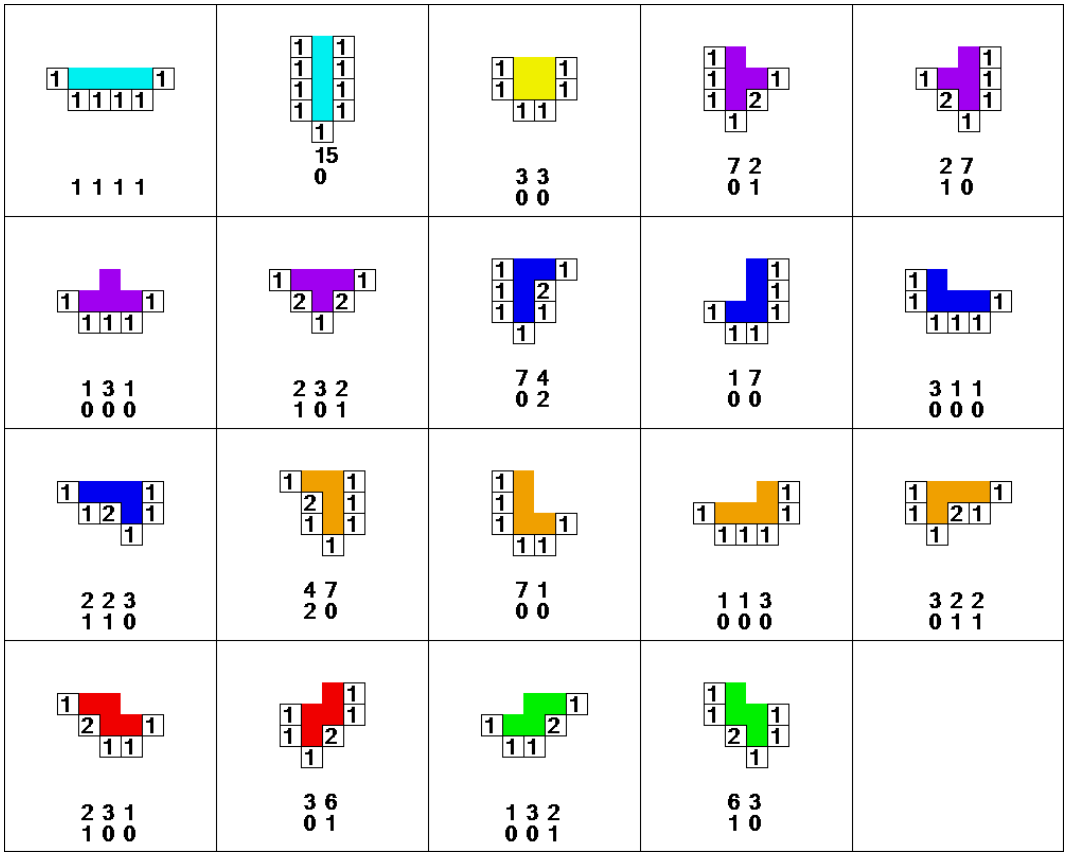


Figura 5.1: Informazioni precalcolate dei tetramini

Le strutture dati sopra mostrate vengono utilizzate nelle funzioni che verranno trattate nelle prossime sottosezioni.

5.2.2.3 PutPiece

La funzione PutPiece, analogamente al caso booleano della sottosezione 5.2.1.5, si occupa di posizionare un pezzo in una certa colonna:

Listing 5.9: La funzione PutPiece

```
ï»¿private static int PutPiece(uint[] field, PieceRotation
    piece, int column)
{
    var height = 0;
    var values = piece.Values;
    var width = piece.Width;
    var lowestBits = piece.LowestBits;

    // Scorro in altezza le colonne presenti sotto il pezzo
    for (var n = 0; n < width; n++)
```

```
    {
        // Calcolo l'altezza
        var max = log2(field[column + n]);

        max -= lowestBits[n];

        if (max > height) height = max;
    }

    height++;

    for (var n = 0; n < width; n++)
    {
        field[column + n] |= values[n] << height;
    }

    return height - 1;
}
```

Per calcolare l'altezza alla quale posizionare il pezzo scorre le colonne del pezzo cercando la posizione più bassa alla quale può essere posizionato. Utilizzando i LowestBits del pezzo è nota a priori la mappa dell'altezza dei blocchi pieni, questo permette con una semplice sottrazione di ottenere la riga di posizionamento. Il valore di ritorno della funzione è l'altezza alla quale è stato posizionato il pezzo, questo permette di eliminare la funzione FindLowestHeight, vista nella sottosezione 5.2.1.1.

5.2.2.4 CalculateContacts

La funzione CalculateContacts fa le veci della funzione GetEdgesTouching della sottosezione 5.2.1.3:

Listing 5.10: Funzione CalculateContacts

```
ï»¿private static void CalculateContacts(uint[] field,
    PieceRotation piece, int column, int height, out int etp,
     out int etw)
{
    var boundaries = piece.BoundaryCheckItems;

    etw = 0;
    etp = 0;
```

```
    foreach (var boundary in boundaries)
    {
        var col = column + boundary.x;

        if (col < 0 || col >= FieldWidth)
        {
            etw++;
            continue;
        }

        var blockHeight = height + boundary.y;

        if ((field[col] & (1 << (blockHeight + 1))) != 0)
        {
            etp += boundary.count;
        }
    }
}
```

Sfruttando i BoundaryCheckItems del pezzo è sufficiente scorrere la lista e controllare se esiste un blocco in quella posizione dello schema. I parametri ETP ed ETW vengono differenziati verificando se la posizione del boundary è sul bordo dello schema di gioco.

5.2.2.5 ClearLines

La funzione ClearLines è senza dubbio la funzione più complicata dell'intera implementazione binaria del Tetris. Analogamente alla funzione RemoveCompletedLines, vista nella sottosezione 5.2.1.2, il suo compito è quello di rimuovere le linee completate e restituire il parametro Lines:

Listing 5.11: Funzione ClearLines

```
ï»¿private const uint MaxValue = 0xFFFFF;

private static int ClearLines(uint[] field)
{

    int linesCompleted = 0;

    // Scorro in altezza a partire dal basso
```

```
for (var n = 0; n < FieldHeight; n++)
{
    // Maschera col bit corrispondente all'altezza
       corrente
    // Lo schema parte dal secondo bit perchÃš col primo
       abbiamo
    // dei problemi col log2(0)
    uint ind = 1u << (n + 1);
    var complete = ind;

    // Controllo se la linea Ãš completa, ovvero se tutte
       le colonne
    // hanno il bit settato all'altezza corrente
    for (var h = 0; h < FieldWidth; h++)
        complete &= field[h] & ind;

    if (complete != 0)
    {
        linesCompleted++;

        // Dobbiamo rimuovere la linea abbassando tutto
           quello che c'Ãš sopra
        // Mi preparo due maschere: inferiore e superiore
        // La maschera inferiore serve a preservare "quello
             che c'Ãš sotto" la linea completata
        uint low = MaxValue >> (FieldHeight - n - 1);

        // La maschera superiore estrae "quello che c'Ãš
           sopra"
        uint high = ~low << 1;

        for (var h = 0; h < FieldWidth; h++)
        {
            // La colonna Ãš la somma tra la maschera
               inferiore e la superiore abbassata di 1
            field[h] = (field[h] & low) | ((field[h] & high)
                >> 1);
        }

        // Dopo aver rimosso la linea devo ripetere il
```

```
                check sulla linea corrente perchÃš potrebbe
            // essere completa anch'essa :)
            n--;
        }
    }

    return linesCompleted;
}
```

La maggior parte dei commenti presenti nel codice sono autoesplicativi. Il funzionamento è il seguente:

1. Per ogni riga dello schema partendo dal basso
2. Se la linea è completa
3. Si preparano due maschere di bit per estrarre quello che sta sopra la linea completa e quello che sta sotto
4. Per ogni colonna dello schema
5. Si applicano le due maschere di bit avendo cura di abbassare di 1 i bit che stanno sopra la linea completata
6. Si ripete da 1 con la linea corrente in quanto potrebbe essere completa

5.2.2.6 CountGaps

La funzione CountGaps si occupa di contare gli spazi presenti nello schema di gioco in maniera analoga a GetGaps vista nella sottosezione 5.2.1.4. La sua implementazione è molto semplice:

Listing 5.12: Funzione CountGaps

```
ï»¿private static int CountGaps(uint[] field)
{
    int count = 0;

    for (var n = 0; n < FieldWidth; n++)
    {
        var col = field[n];

        count += log2(col) - CountBits(col);
    }
```

```
    return count;
}
```

Scorrendo lo schema per colonna, il numero di gaps è pari alla differenza tra l'altezza della colonna (highest bit) e il numero di bit 1. Infatti, se una colonna fosse piena, ovvero non avesse buchi, l'highest bit e il numero di bit 1 coinciderebbe. Ripetendo questa operazione per tutte le colonne e sommando i conteggi parziali si ottiene il valore globale del parametro Gaps.

5.2.3 Confronto prestazioni

Il confronto prestazionale tra le diverse implementazioni del Tetris è molto importante per tenere sempre presente la qualità del lavoro svolto. La piattaforma sulla quale sono stati effettuati i test è la seguente:

- CPU: Intel Core i7-3610QM
- RAM: G.Skill 16GB DDR3-1330
- HDD: OCZ-Vertex 3
- OS: Windows 8.1 Pro x64

Sono state implementate due versioni dello schema di gioco: booleano e binario. Nonostante questo ci sono altri parametri che influiscono sulle prestazioni, uno fra tutti il generatore di numeri casuali. Negli esperimenti che seguiranno sono stati utilizzati i seguenti generatori di numeri casuali:

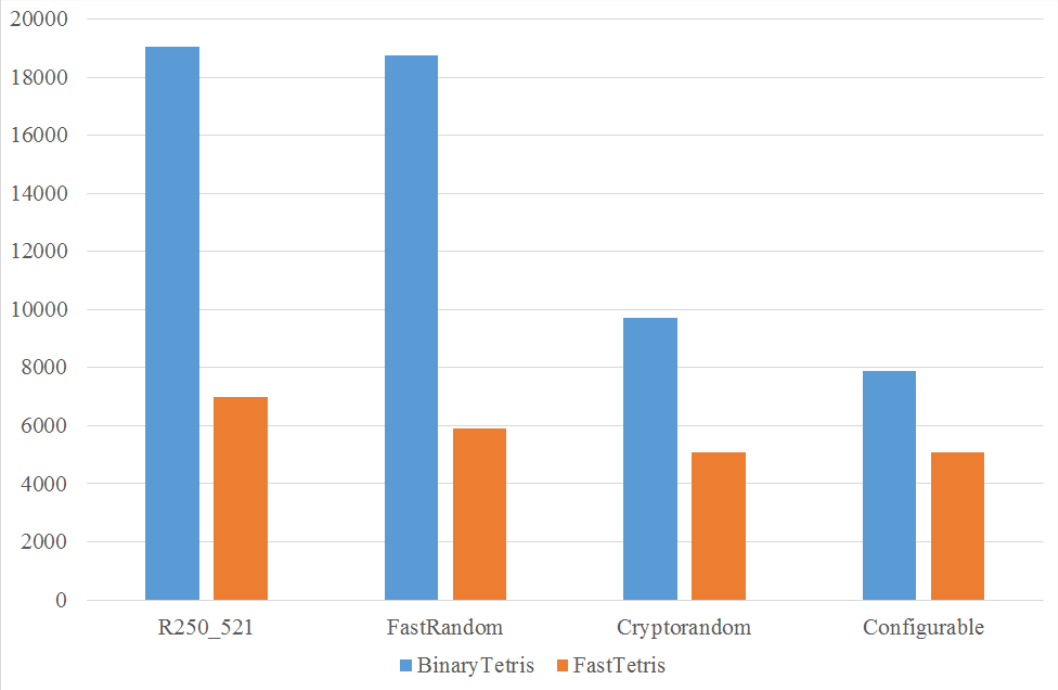
- R250/521: Implementazione di un algoritmo a registro di scorrimento ad alte prestazioni (http://en.literateprograms.org/R250/521_(C))
- FastRandom: Implementazione di System.Random senza controlli
- CryptoRandom: Tramite la classe crittografica RNGCryptoServiceProvider genera numeri casuali basati su entropia
- ConfigurableRandom: Implementazione distribuzione non uniforme intera per alterare le probabilità dei singoli pezzi:

Listing 5.13: Funzione Next di ConfigurableRandom

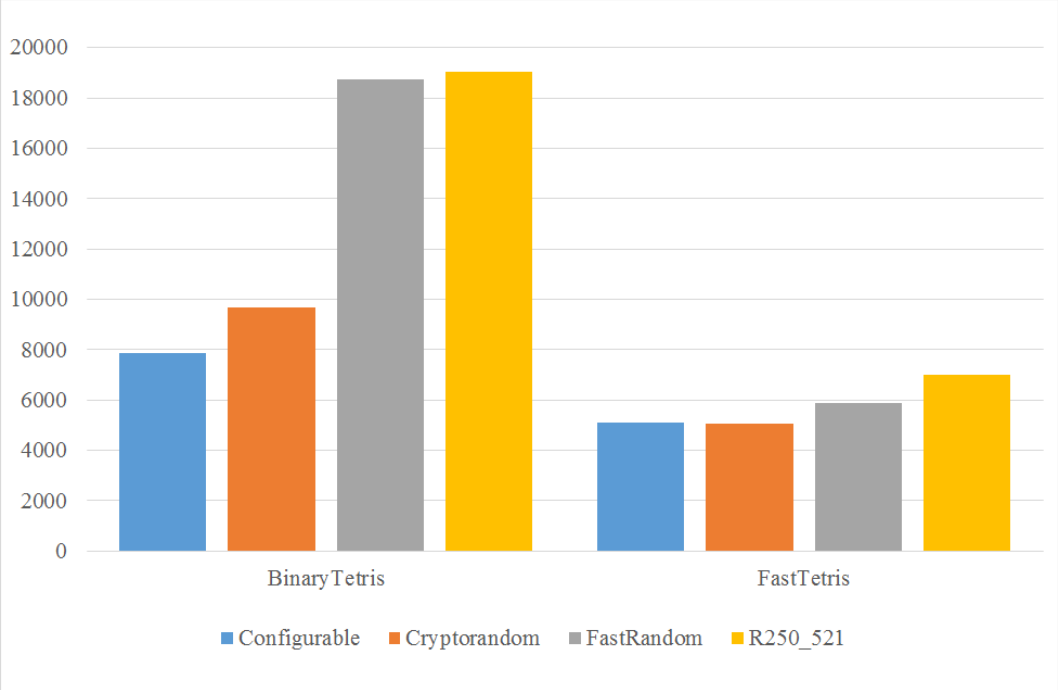
```
ï»¿public int Next()
{
    var totalSum = Probabilities.Sum();
    var index = rand.NextDouble() * totalSum;
    double sum = 0;
```

```
    int i = 0;

    while (sum < index)
    {
        sum = sum + Probabilities[i++];
    }

    return i - 1;
}
```

Nei seguenti grafici sono messe a confronto le due implementazioni FastTetris (schema booleano) e BinaryTetris:



(a) Confronto prestazionale raggruppato per generatore di numeri casuali



(b) Confronto prestazionale raggruppato per implementazione

Figura 5.2: Confronto prestazionale

Come è possibile vedere nel grafico 5.2a raggruppato per generatore di numeri casuali, l'implementazione binaria del tetris ha sempre un consistente margine su quella booleana. Le differenze prestazionali si riducono impiegando un generatore di

numeri casuali più lento, come ad esempio quello crittografico, ma rimangono sempre nettamente a favore dell'implementazione binaria. Nel grafico 5.2b viene messo in evidenza come l'utilizzo di un generatore di numeri pseudocasuali molto leggero come l'R250/521 garantisca prestazioni nettamente maggiori nel caso dell'implementazione binaria. Invece, per quanto riguarda l'implementazione booleana, le differenze non sono così marcate, segno che il tempo maggiore di esecuzione è impiegato nella manipolazione dello schema e non alla generazione casuale dei pezzi. Di seguito la tabella con i valori sorgente della figura 5.2:

	R250_521	FastRandom	Cryptorandom	Configurable
BinaryTetris	19043	18752	9693	7875
FastTetris	6986	5895	5062	5087

Tabella 5.1: Velocità di esecuzione espresse in pezzi al secondo

5.3 Implementazione algoritmo genetico

Il framework genetico impiegato nella realizzazione dell'intelligenza artificiale è stato realizzato ad hoc per questo particolare caso d'uso. La caratteristica sulla quale è stata posta più attenzione è stata la parallelizzazione del processo evolutivo. Grazie alla TPL (Task Parallel Library) offerta dal framework .NET è stato relativamente semplice soddisfare questo requisito, ottenendo l'utilizzo contemporaneo di tutti e 8 i core logici della piattaforma di test. Gli algoritmi genetici si prestano molto bene all'esecuzione parallela perché è possibile eseguire multiple valutazioni di fitness contemporaneamente dividendo il carico sulle unità di elaborazione.

L'algoritmo genetico implementato fa uso di geni rappresentati da numeri reali, per questo motivo le tecniche di mutazione differiscono dal caso binario introducendo il blending, trattato nella sottosezione 5.3.4.1.

5.3.1 Evolve

La procedura Evolve fa partire l'evoluzione dell'algoritmo genetico occupandosi di costituire la popolazione iniziale e di ripetere il ciclo di evoluzione fino al raggiungimento della condizione di terminazione:

Listing 5.14: Funzione Evolve

```
ï»¿public void Evolve()
{
    // Prerequisite
    if (Generation == 0)
    {
```

```
        _genomes = (from index in ParallelEnumerable.Range(0,
            Settings.PopulationSize)
                    let bestUnit = (from unit in Enumerable.Range
                        (0, Settings.TournamentPrerequisite)
                                let genome =
                                    _createGenomeFunction()
                                let fitness = FitnessWrapper(
                                    genome)
                                orderby fitness descending
                                select genome
                        ).First()
                    select bestUnit).ToList();

        Generation = 0;
    }

    do
    {
        Generation++;

        EvaluateFitness();

        var bestOfGeneration = Genomes.Aggregate((i1, i2) =>
            i1.AverageFitness > i2.AverageFitness ? i1 : i2);

        if (BestGenome == null || bestOfGeneration.
            AverageFitness > BestGenome.AverageFitness)
            BestGenome = bestOfGeneration;

        var stop = _stopFunction(this);

        if (stop) break;

        NaturalSelection();
        ExpandPopulation();

    } while (true);

}
```

Da notare l'utilizzo del metodo statico della TPL ParallelEnumerable.Range che restituisce un oggetto enumerabile parallelamente per distribuire il carico su tutte le unità di elaborazione. Il codice fa largo uso di LINQ (Language INtegrated Query) per rendere il codice estremamente compatto ed autoesplicativo.

5.3.2 NaturalSelection

La procedura NaturalSelection seleziona i migliori cromosomi della popolazione ed è governata dal parametro NewGenerationScaleFactor che indica la percentuale di unità coinvolte nei meccanismi di accoppiamento e ricombinazione:

Listing 5.15: Procedura NaturalSelection

```
ï»¿private void NaturalSelection()
{
    var survivors = (int)(Settings.NewGenerationScaleFactor *
        Settings.PopulationSize);

    _genomes = _genomes.OrderByDescending(item => item.
        AverageFitness)
            .Take(survivors)
            .ToList();
}
```

Grazie ai metodi di estensione OrderByDescending e Take è possibile ritagliare la porzione di popolazione con valore di fitness più elevato.

5.3.3 ExpandPopulation

La procedura ExpandPopulation si occupa di espandere la popolazione rimasta fino a ricostituire l'intero pool genetico:

Listing 5.16: Procedura ExpandPopulation

```
ï»¿private void ExpandPopulation()
{
    var eliteCount = (int)(Settings.ElitismPercentage *
        Settings.PopulationSize);

    var newUnitsCount = Settings.PopulationSize - eliteCount;

    var genomeLength = _genomes[0].Genes.Length;

    _genomes.RemoveRange(eliteCount, _genomes.Count -
        eliteCount);
```

```
    for (var n = 0; n < newUnitsCount; n++)
    {
        var first = _rand.Next(_genomes.Count);
        var second = _rand.Next(first);
        var child = MakeChildOf(_genomes[first], _genomes[
            second]);

        // Se c'Ãš almeno un genoma che ha tutti i geni
            identici a questo genoma
        if (!Settings.AllowDuplicates &&
            _genomes.Any(item =>
                Enumerable.Range(0, genomeLength).All(index =>
                    Math.Abs(item.Genes[index] - child.Genes[
                        index]) < CompareTolerance)))
        {
            // Mutiamo i duplicati!
            MutateGenome(child, true);
        }

        _genomes.Add(child);

    }
}
```

Il parametro ElitismPercentage indica quante unità faranno parte di quella élite che rimarrà inalterata nella nuova generazione. Tutto il resto della popolazione è ottenuto tramite ricombinazione.

Il metodo di selezione delle unità fa uso di due numeri casuali: il primo viene estratto tra l'intera popolazione mentre il secondo solo tra le unità con valore di fitness più elevato della prima unità selezionata. In questo modo le unità con maggiore fitness avranno l'opportunità di accoppiarsi con maggiore frequenza rispetto alle altre.

5.3.4 MakeChildOf

La funzione MakeChildOf permette di ricombinare tra loro due unità e di restituire un loro possibile figlio:

Listing 5.17: Funzione MakeChildOf

```
ï»¿private Genome MakeChildOf(Genome first, Genome second)
```

```
{

    var genomeLength = first.Genes.Length;

    var genome = new Genome
    {
        Genes = new double[genomeLength]
    };

     if (_rand.NextDouble() <= Settings.
         CrossingOverProbability)
     {

        switch (Settings.CrossingOverType)
        {
            case CrossingOverType.SinglePoint:

                var point = _rand.Next(genomeLength);

                for (var n = 0; n < point; n++)
                {
                    genome.Genes[n] = first.Genes[n];
                }

                for (var n = point; n < genomeLength; n++)
                {
                    genome.Genes[n] = second.Genes[n];
                }

                break;

            case CrossingOverType.DoublePoint:

                var firstPoint = _rand.Next(genomeLength);
                var secondPoint = _rand.Next(genomeLength);

                if (firstPoint > secondPoint)
                {
                    var temp = firstPoint;
                    firstPoint = secondPoint;
```

```
            secondPoint = temp;
        }

        for (var n = 0; n < firstPoint; n++)
        {
            genome.Genes[n] = first.Genes[n];
        }

        for (var n = firstPoint; n < secondPoint; n++)
        {
            genome.Genes[n] = second.Genes[n];
        }

        for (var n = secondPoint; n < genomeLength; n++)
        {
            genome.Genes[n] = first.Genes[n];
        }

        break;

    case CrossingOverType.SwapBlend:

        for (var n = 0; n < genomeLength; n++)
        {
            genome.Genes[n] = _rand.NextDouble() <
                Settings.SwapProbability ? second.Genes[n]
                 : first.Genes[n];

            if (_rand.NextDouble() < Settings.
                BlendProbability)
                genome.Genes[n] = (first.Genes[n] + second
                    .Genes[n]) / 2;

        }

        break;

    default:
        throw new ArgumentOutOfRangeException();
}
```

```
    }
    else
    {
      for (var n = 0; n < genomeLength; n++)
      {
          genome.Genes[n] = first.Genes[n];
      }
    }

  MutateGenome(genome);

  return genome;
}
```

In questa funzione sono implementati 3 diversi tipi di crossover: SinglePoint, DoublePoint e SwapBlend. Mentre i primi due sono trattati nella sottosezione 2.3.4, il terzo è stato realizzato ad hoc per questa implementazione dell'algoritmo genetico.

5.3.4.1 SwapBlend

La tipologia di crossover SwapBlend unisce lo scambio di materiale genetico con la fusione dei geni. I parametri che governano la procedura di ricombinazione sono:

- SwapProbability (10%): Probabilità che i geni vengano scambiati
- BlendProbability (10%): Probabilità che i geni vengano fusi

Dai test condotti in sede preliminare agli esperimenti di questa tesi è emerso che il crossover SwapBlend garantisce velocità di convergenza e prestazioni maggiori dell'algoritmo genetico rispetto a SinglePoint e DoublePoint.

Il vantaggio di questo nuovo metodo di ricombinazione è garantito dalla continuità del dominio dei valori dei geni. Nel problema in esame i valori ricombinati sono assimilabili a dei segnali analogici, il che permette all'operazione «media» di dare un contributo utile e positivo.

Capitolo 6

Risultati sperimentali

6.1 Introduzione

In questo capitolo sono riassunti i risultati ottenuti nelle diverse prove effettuate. La struttura con la quale vengono presentati segue il procedimento logico con il quale sono stati condotti gli esperimenti. Nella sezione 6.2 sono presenti le prove condotte con i cromosomi di lunghezza standard, ovvero costituiti dai geni trattati nella sezione 4.2. In particolare viene posta l'attenzione sulle strategie di ottimizzazione messe in campo dall'intelligenza artificiale al variare dell'obbiettivo.

La sezione 6.3 tratta di una possibile estensione del genoma di base per fornire all'intelligenza artificiale nuovi strumenti per raggiungere l'obbiettivo. Molto importante è la sezione 6.4 che tratta la ripetibilità degli esperimenti condotti. È infatti necessario verificare se i risultati di un processo evolutivo sono consistenti oppure se l'algoritmo genetico raggiunge valori diversi ad ogni esecuzione concentrandosi in massimi locali.

Nella sezione 6.5 viene condotta l'analisi presentata nella sezione 4.3.1.3. Le diverse tipologie di tetramini vengono messe a confronto per estrarre preziose informazioni riguardanti la loro difficoltà di gestione da parte dell'intelligenza artificiale.

Se non diversamente indicato questi sono i parametri di base della simulazione:

- AllowDuplicates = False
- BlendProbability = 10%
- CrossingOverProbability = 100%
- CrossingOverType = SwapBlend
- ElitismPercentage = 5%
- FitnessEvaluations = 3
- MutationProbability = 2%

- MutationType = DeltaAbsolute
- MaxMutationDelta = 1
- NewGenerationScaleFactor = 20%
- PopulationSize = 100
- SwapProbability = 10%
- TournamentPrerequisite = 10
- IRandom = CryptoRandom (RNGCryptoServiceProvider)

6.2 Cromosoma standard

Il cromosoma standard è composto dai geni trattati nella sezione 4.2: Height, ETP, ETW, Gaps e Lines. L'intelligenza artificiale ha a disposizione molte informazioni per comporre la sua strategia di gioco e durante il processo evolutivo è alla ricerca del modo migliore per interpretare i valori dei parametri. Più un gene è grande rispetto agli altri, maggiore è l'importanza che assume nel processo decisionale. Il tipo di feedback ottenuto da un certo parametro può essere positivo o negativo. Se il feedback è positivo significa che l'intelligenza artificiale ha come sotto obbiettivo la sua massimizzazione, se negativo la minimizzazione.

6.2.1 Ottimizzazione per numero di tetramini

L'ottimizzazione per numero di tetramini ha come obbiettivo la massimizzazione della durata della partita cercando di effettuare il maggior numero di mosse. Sono state completate 18 generazioni del processo evolutivo. Il motivo che non ha permesso di raggiungere le 20 generazioni è da ricercare nella velocità di simulazione. Il tempo medio di completamento di una generazione di questo esperimento è di circa 8 ore. In totale il tempo speso per ottenere questi risultati è stato di 5 giorni, 22 ore, 47 minuti e 43 secondi.

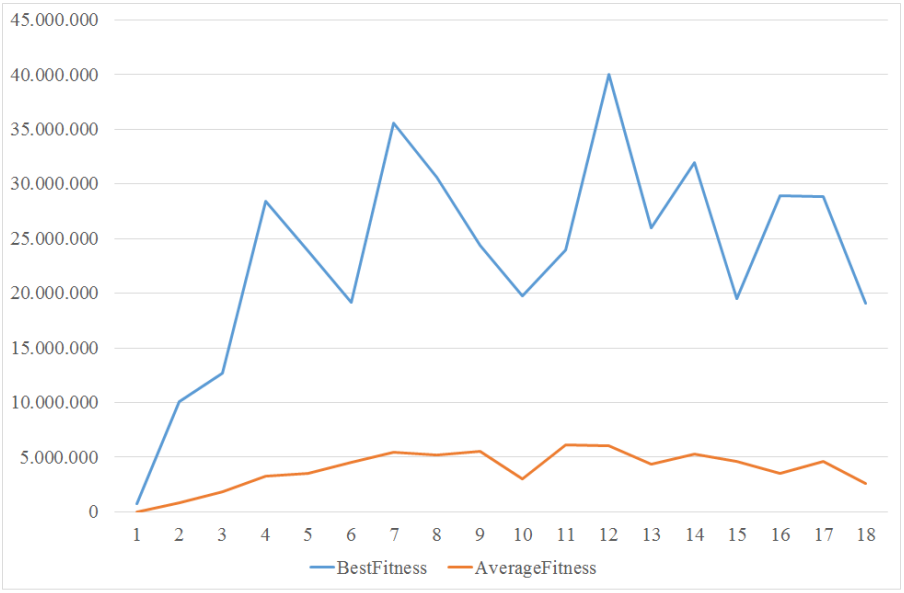


Figura 6.1: Fitness media e massima (ottimizzazione per numero di tetramini)

Nell'immagine 6.1 è possibile seguire l'andamento dell'evoluzione dell'intelligenza artificiale. Da notare l'andamento erratico della BestFitness da imputare alle sequenze casuali dei tetramini. Infatti, come specificato nella sezione 2.3.2, il processo di valutazione della fitness è mutabile, e quindi fortemente dipendente dal generatore di numeri casuali. Interessante è l'andamento della fitness media della popolazione che si stabilizza intorno ai 5 milioni di punti, questo indica che l'evoluzione ha raggiunto un plateau e probabilmente non è in grado di ottenere risultati migliori.

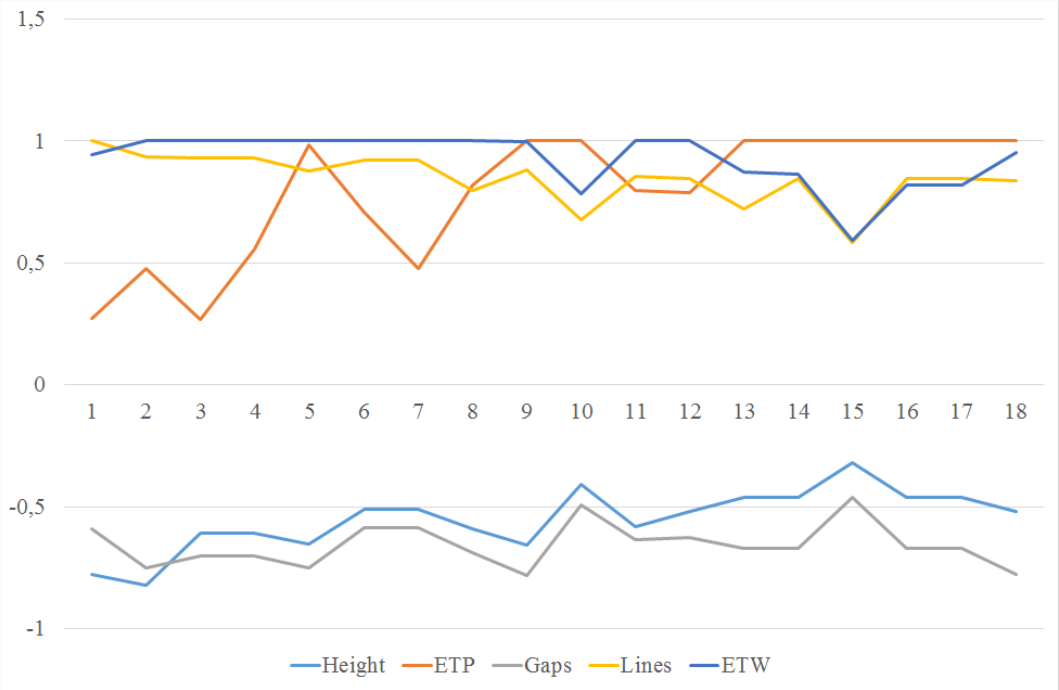


Figura 6.2: Valori dei geni migliori delle popolazioni (ottimizzazione per numero di tetramini)

L'andamento dei valori dei geni delle unità migliori di ciascuna popolazione è riassunto nella figura 6.2, è possibile normalizzare i valori dei geni in un intervallo [-1;1] per confrontare tra loro i risultati di diversi esperimenti evidenziando i rapporti di forza che intercorrono tra i valori. Nel grafico è possibile vedere come i parametri ETP, ETW e Lines siano forti segnali positivi riguardanti la correttezza della mossa, al contrario Gaps ed Height sono sostanzialmente negativi. La strategia che si è evoluta in questo esperimento delinea un'intelligenza artificiale che non ha preferenze su dove posizionare i pezzi (ETP ~ ETW), e cerca di limitare sia la creazione di spazi all'interno dello schema di gioco sia l'incremento dell'altezza.

Se si osservano le variazioni dei valori dei geni di generazione in generazione si nota che la maggior parte dei valori rimane all'interno di un certo range. A parte il gene ETP che aumenta di importanza, gli altri rimangono sostanzialmente invariati. Questo conferma quanto detto all'inizio di questa sezione: l'evoluzione ha raggiunto l'equilibrio. Per verificare se il massimo raggiunto è causato da una riduzione di diversità genetica è possibile prendere in esame la popolazione di una generazione. E' stata scelta la popolazione della generazione 12 perché contiene il massimo assoluto di fitness (40.002.790) e la media (6.081.965) è in linea con le altre generazioni.

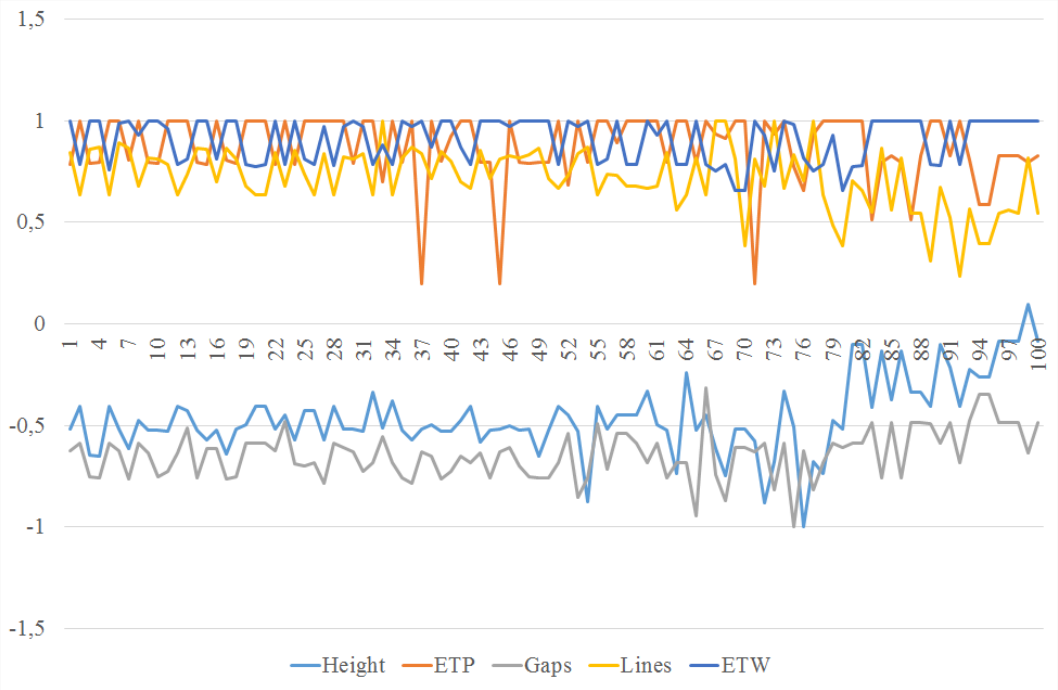


Figura 6.3: Valori dei geni delle unità della generazione 12 ordinati per AverageFitness

Le unità sono ordinate per AverageFitness che rappresenta la media delle 3 misurazioni della fitness. Nella parte sinistra abbiamo le unità con fitness maggiore e a destra quelle con fitness minore. Si può notare come la maggior parte delle unità è dotata di geni che oscillano intorno agli stessi valori, mentre esiste una percentuale, intorno al 30%, di unità con geni notevolmente diversi.

L'andamento della AverageFitness è di tipo esponenziale. Riprendendo quanto detto all'inizio del capitolo, il tipo di sequenza di tetramini in uscita dal generatore di numeri casuali determina le grandi differenze tra il valore massimo e minimo della fitness. Questo significa che anche una buona unità può incontrare una sequenza di pezzi molto difficile da gestire e perdere la partita di molto prima di quanto sarebbe stata in grado di fare in un altro scenario.

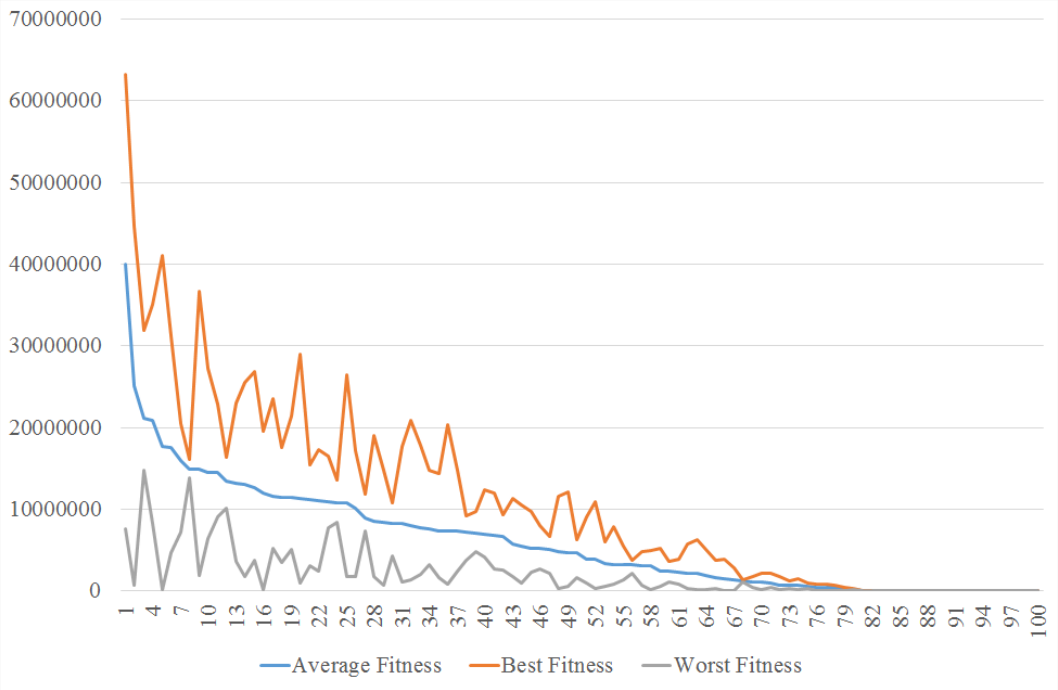


Figura 6.4: Fitness delle unità della generazione 12 ordinati per AverageFitness

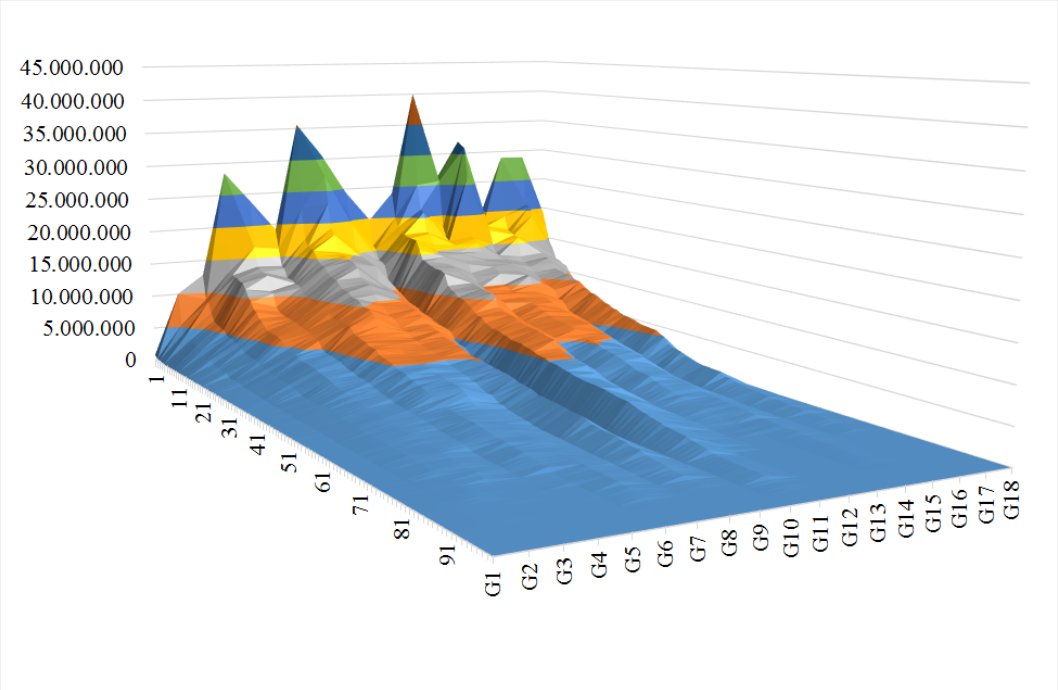


Figura 6.5: Andamento fitness media delle popolazioni

Per evidenziare la composizione delle popolazioni è possibile rappresentare la fitness media delle unità in un grafico ad area 3D. Mentre nel grafico 6.1 sono rappresentate solo le migliori unità per popolazione, nella figura 6.5 è possibile vedere

l'intera composizione di tutte le popolazioni. La maggior parte delle unità ha una fitness minore di 5.000.000 e solo circa il 30% raggiunge i livelli più alti. Di questo 30% solo un esiguo numero di unità dimostra di avere una fitness molto elevata, evidente dal grafico 6.4.

I risultati di questo esperimento sono molto interessanti, evidenziano quanto il generatore di numeri casuali influisca sulla fitness massima che si può raggiungere. Dal grafico 6.3 è evidente che il primo 30% di unità ha geni molto simili ma le differenze in fatto di fitness sono sostanziali, questo rafforza ancora di più la tesi secondo la quale la sequenza di tetramini in ingresso gioca un ruolo fondamentale. Nella tabella 6.1 sono messi a confronto i geni di alcune unità molto simili al campione della generazione 12. Nonostante la forte somiglianza con il migliore della generazione il valore della fitness media è notevolmente inferiore.

Fitness	Height	ETP	ETW	Gaps	Lines	Change
40.002.790	**-0,40571**	**0,61694**	**0,78383**	**-0,49078**	**0,66210**	
11.030.583	-0,40571	0,61694	0,78383	-0,49078	0,66210	-
8.271.557	-0,40571	0,61694	0,78169	-0,49078	0,63216	0,04127

Tabella 6.1: Unità molto simili al campione della generazione 12

Per riuscire ad ottenere risultati consistenti eliminando le oscillazioni date dal generatore di numeri casuali è necessario effettuare un maggior numero di valutazioni della fitness. Valutando 3 volte la fitness per unità, come nell'esperimento corrente, si riesce ad ottenere una valutazione poco stabile. Infatti sarebbe necessario incrementare il numero di valutazioni a circa 10-20 per garantire la validità statistica del dato. Il problema principale di questa strategia risiede nel tempo di valutazione: per ottenere 18 generazioni composte da 100 unità valutate 3 volte la simulazione è durata per poco meno di 6 giorni continuativi. La finestra temporale a disposizione dell'esperimento purtroppo non è stata sufficientemente ampia da poter estendere la simulazione ulteriormente.

6.2.2 Ottimizzazione per punteggio

L'ottimizzazione per punteggio ha come obbiettivo di massimizzare il punteggio a parità di numero di tetramini posizionati. Una misurazione che non è stata mostrata nella sezione 6.2.1 riguarda la velocità con la quale le unità ottengono i punti. La metrica è definita come Fitness / Iteration ovvero come «punteggio al pezzo». Durante l'evoluzione le unità arrivano facilmente a raggiungere il limite massimo di 100.000 pezzi, a quel punto l'unico modo per aumentare ulteriormente la fitness è quella di sviluppare una strategia che gli permetta di incrementare i punti per pezzo.

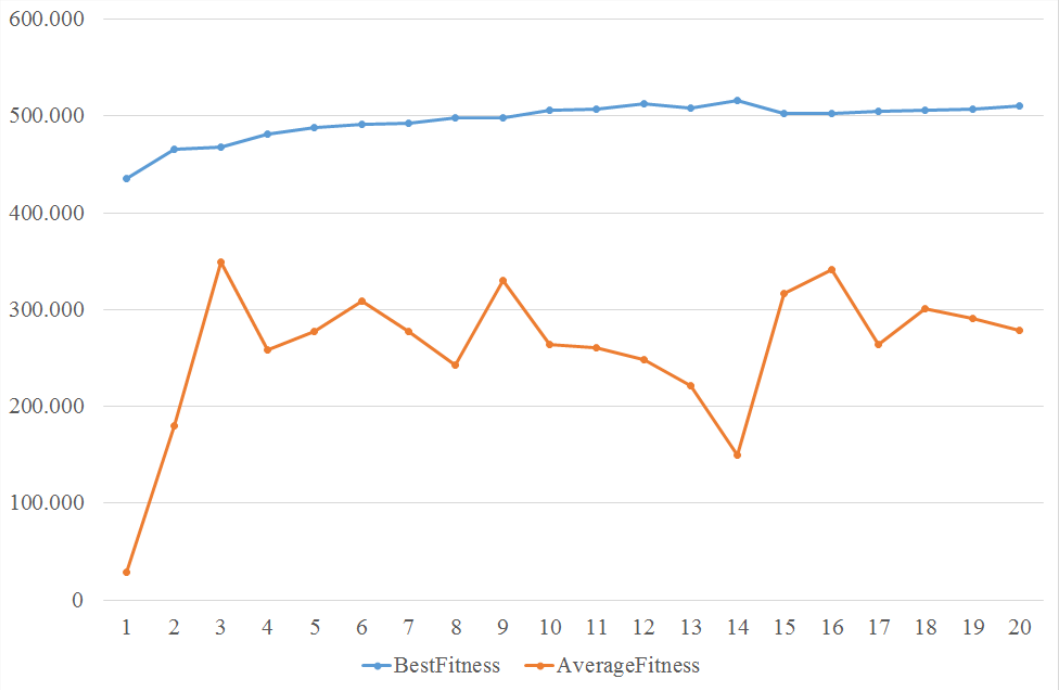


Figura 6.6: Fitness media e massima (ottimizzazione per punteggio)

Nel grafico 6.6 è possibile seguire l'evoluzione della popolazione durante le 20 generazioni. La fitness media oscilla intorno al valore 300.000 mentre la massima sale con bassa pendenza fino ad assestarsi poco sopra i 500.000 punti. Da notare come, fin dalle prime generazioni, la fitness massima e media siano subito molto vicine al valore finale. Questo andamento è indicativo di un processo evolutivo che ha raggiunto un massimo relativo o assoluto. Per capirlo è necessario analizzare i geni delle unità e verificare se è presente sufficiente diversità genetica.

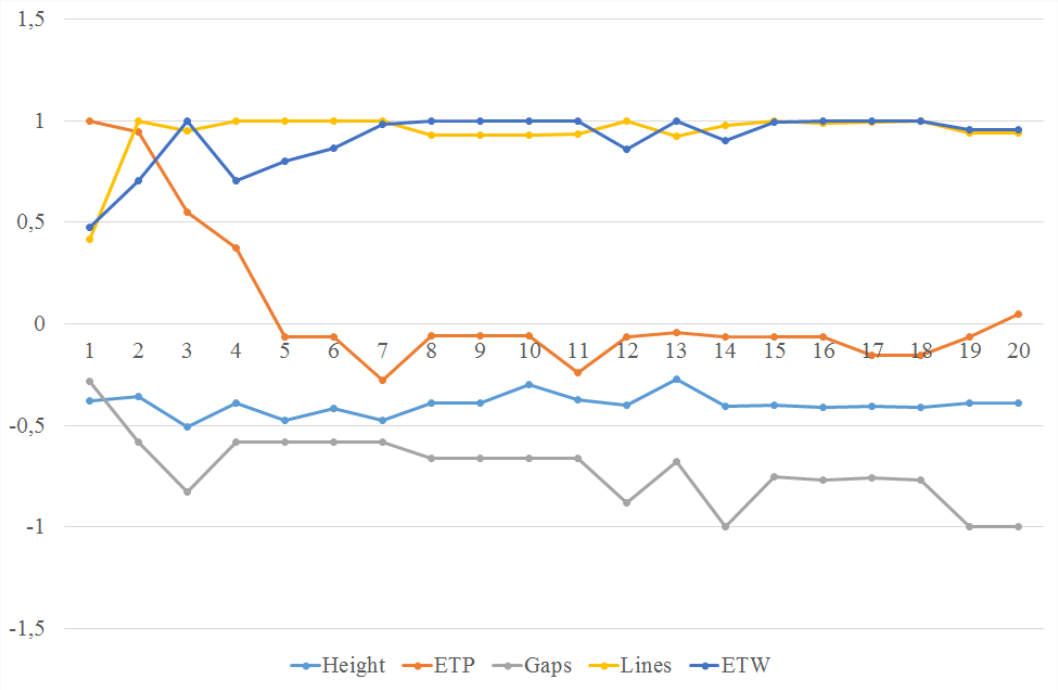


Figura 6.7: Valori dei geni migliori delle popolazioni (ottimizzazione per punteggio)

Dal grafico 6.7 è possibile seguire l'evoluzione dei geni delle unità migliori di ogni generazione. Per la prima volta incontriamo un caso in cui l'evoluzione scarta delle informazioni che gli vengono fornite. Il gene ETP assume un'importanza sempre minore fino alla generazione 5 dove il suo valore si assesta intorno allo zero. Questo significa che l'informazione contenuta nel parametro ETP è considerata superflua e non prende sostanzialmente parte al processo decisionale. Il rapporto tra ETP ed ETW vicino allo zero significa che l'intelligenza artificiale cerca il più possibile di posizionare i tetramini adiacenti alle pareti dello schema di gioco. Questo comportamento si può spiegare intuitivamente pensando all'utilizzo del tetramino I, che è il pezzo in grado di completare il maggior numero di linee contemporaneamente e quindi di ottenere molti punti con una sola mossa. Combinando l'effetto dei segnali positivi ETW e Lines, l'intelligenza artificiale punta a completare linee multiple inserendo il tetramino I al lati dello schema di gioco. Inoltre, a differenza di quanto visto nel grafico6.2, i geni Height e Gaps non hanno lo stesso valore. Gaps comunica un feedback molto più negativo rispetto a Height, questo significa che le unità cercheranno di evitare di alzare troppo lo schema di gioco ma in maniera meno marcata rispetto alla formazione di spazi. Anche i valori di questi parametri possono essere spiegati pensando alla strategia di gioco: se lo schema si alza senza avere spazi al suo interno è possibile sfruttare il tetramino I per completare più linee contemporaneamente.

Per verificare la diversità genetica viene presa in esame la generazione 14 che registra il valore massimo della fitness (516.327). Nel grafico 6.8 sono presenti tutte le unità della generazione 14 ordinate per fitness media crescente da destra verso

sinistra. Anche considerando le unità con maggiore fitness la diversità genetica è significativa, quindi è possibile concludere che il processo evolutivo non è rimasto bloccato in un massimo locale.

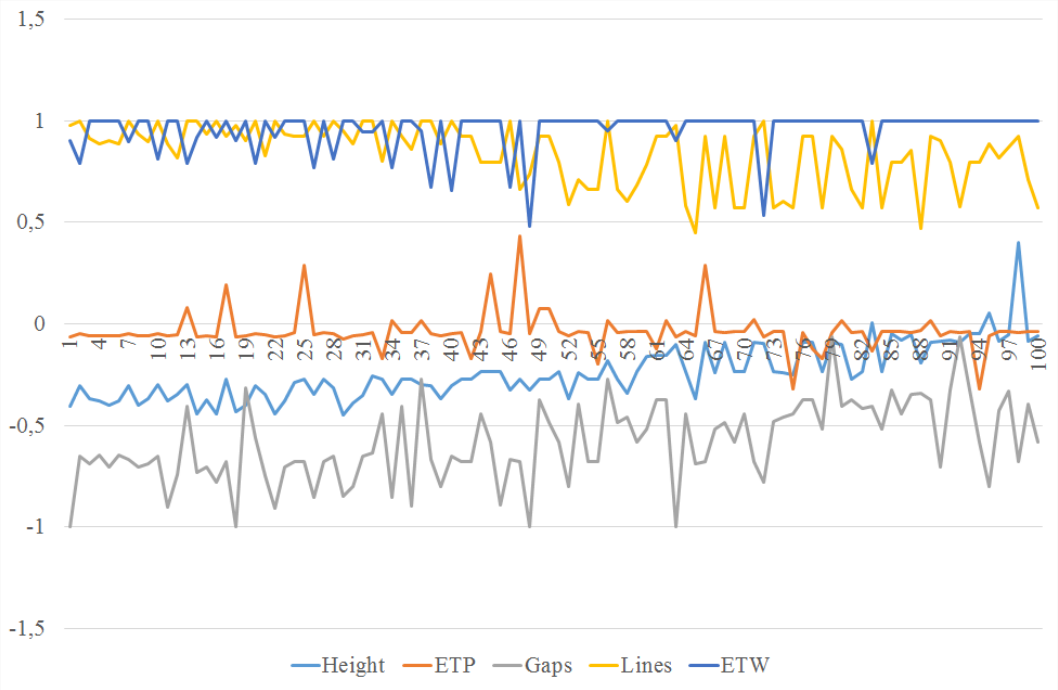


Figura 6.8: Valori dei geni delle unità della generazione 12 ordinati per AverageFitness

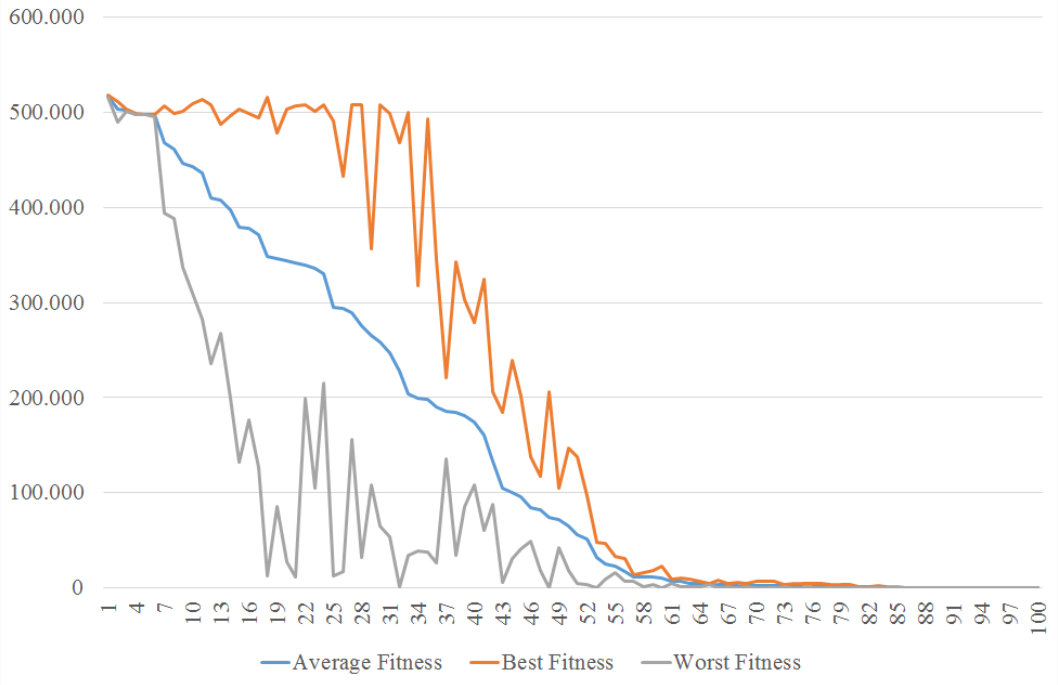


Figura 6.9: Fitness delle unità della generazione 14 ordinati per AverageFitness

Nell'immagine 6.9, allo stesso modo del grafico 6.4, la variabilità delle sequenze di pezzi in uscita dal generatore di numeri casuali comporta grandi variazioni tra la fitness massima e quella minima. Un altro punto da sottolineare è il fatto che il 40% delle unità della popolazione non è in grado di sopravvivere più di poche migliaia di pezzi e il 10% non riesce a fare nessun punto. Questa popolazione è sì in grado di ospitare l'unità con maggiore fitness di tutto il processo evolutivo ma è anche quella con fitness media più bassa dopo la prima generazione. La spiegazione di questo fenomeno non è di facile comprensione, essendo l'evoluzione basata su processi casuali potrebbe essere imputabile a diverse cause. Ad esempio la diffusione di geni mutati in maniera sfavorevole provenienti dalla generazione precedente.

L'immagine 6.10 mostra il confronto tra i punti per pezzo delle unità della generazione 12, trattata nella sezione 6.2.1, e i punti per pezzo delle unità della generazione 14 trattata in questa sezione. Dall'analisi dei dati emerge chiaramente che l'ottimizzazione per punteggio si sta effettivamente muovendo nella direzione indicata. Il numero di punti per pezzo è notevolmente superiore nelle unità che lo hanno come obiettivo.

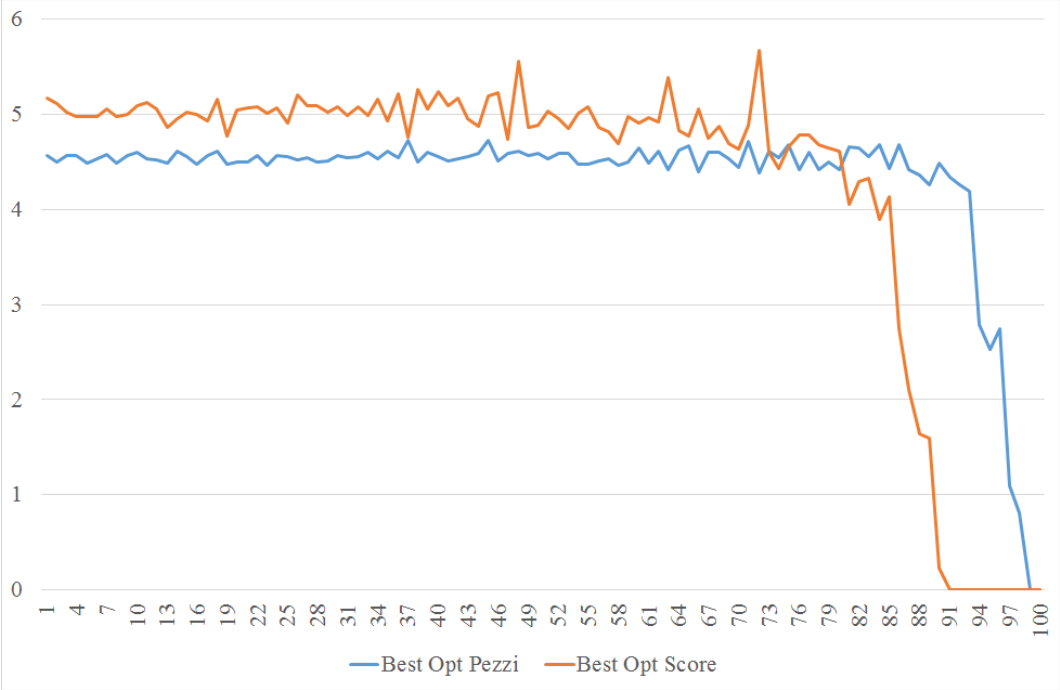


Figura 6.10: Confronto punti per pezzo tra i diversi esperimenti

6.3 Cromosoma esteso

Il cromosoma trattato fino ad ora è costituito da 5 geni: Height, ETP, ETW, Gaps e Lines. Il punto di partenza per questa estensione è stata una considerazione sulla linearità dei segnali dei parametri. Per questo motivo sono stati aggiunti ul-

teriori 5 geni al cromosoma calcolati sulla base di quelli esistenti: $Height^2$, ETP^2, ETW^2, $Gaps^2$ e $Lines^2$. La domanda principale è se questi geni che rispondono a dei parametri non lineari siano di effettiva utilità per l'intelligenza artificiale oppure non aggiungono informazioni utili. Il principio che ha portato a questa estensione è l'idea di mettere a disposizione delle unità un modo per reagire in maniera non lineare ai segnali che riceve. Se si prende per esempio il parametro Gaps = 1 si può immaginare che questo esprima un certo segnale negativo, ma Gaps = 2 è semplicemente il doppio più «grave» oppure molto di più? Aggiungendo questi parametri non lineari si cercano di fornire strumenti più sofisticati all'intelligenza artificiale per raggiungere il suo obbiettivo e nella sezione 6.3.1 verranno mostrati i risultati ottenuti.

6.3.1 Ottimizzazione per punteggio

Il tipo di esperimento è analogo a quello visto nella sezione 6.2.2 in cui le unità hanno a disposizione 100.000 pezzi per ottenere il punteggio maggiore possibile.

La prima cosa che è stata chiara fin dalle prime generazioni del processo evolutivo è stata la diversa velocità di convergenza. A differenza degli esperimenti visti nelle sezioni 6.2.1 e 6.2.2 in cui le unità erano in grado in poche generazioni di raggiungere risultati molto vicini a quelli finali, in questo caso è necessario molto più tempo.

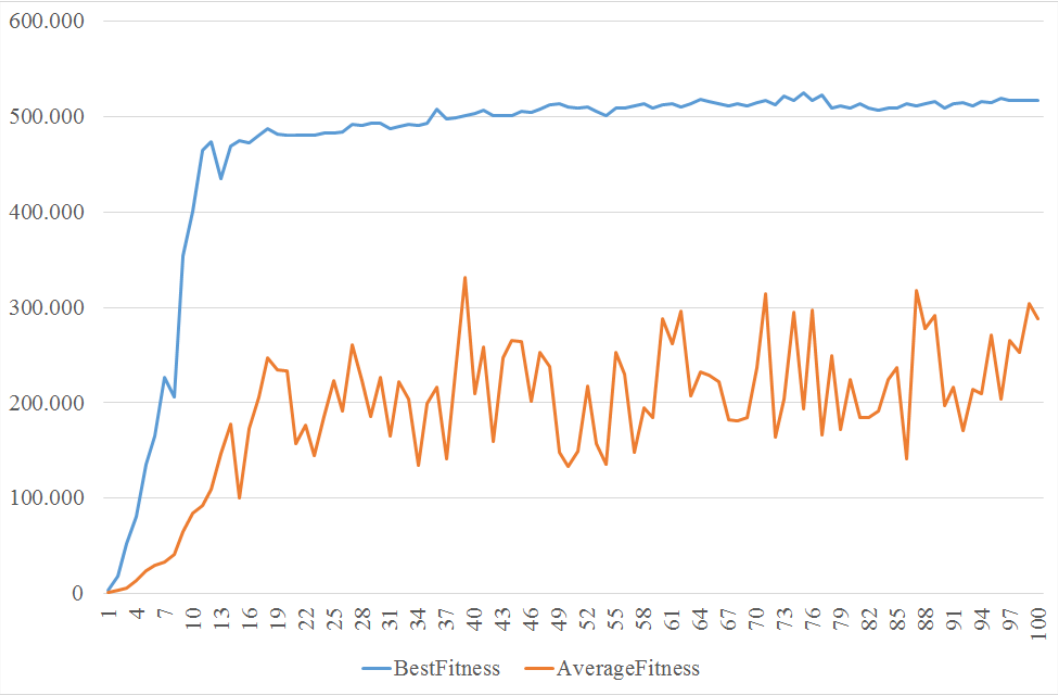


Figura 6.11: Fitness media e massima genoma esteso

Sono state completate 100 generazioni al posto delle 20 degli altri esperimenti per dare tempo al processo evolutivo di affinare l'utilizzo di tutti i parametri. La

velocità di convergenza cresce con l'aumentare del numero di geni nel cromosoma. Nel grafico 6.11 è possibile vedere che la fitness massima sale molto velocemente nelle prime 15 generazioni e da quel momento in poi inizia una lenta salita superando il tetto di 500.000 punti intorno alla generazione 39. La fitness massima raggiunta è pari a 525.363 punti nella generazione 75. Dalla generazione 60 in poi l'evoluzione si assesta intorno ai 515.000 punti e non cresce ulteriormente. La fitness media oscilla intorno ai 200.000 punti e cresce molto velocemente come la fitness massima nelle prime 15 generazioni e poi inizia un andamento oscillatorio per il resto del processo evolutivo.

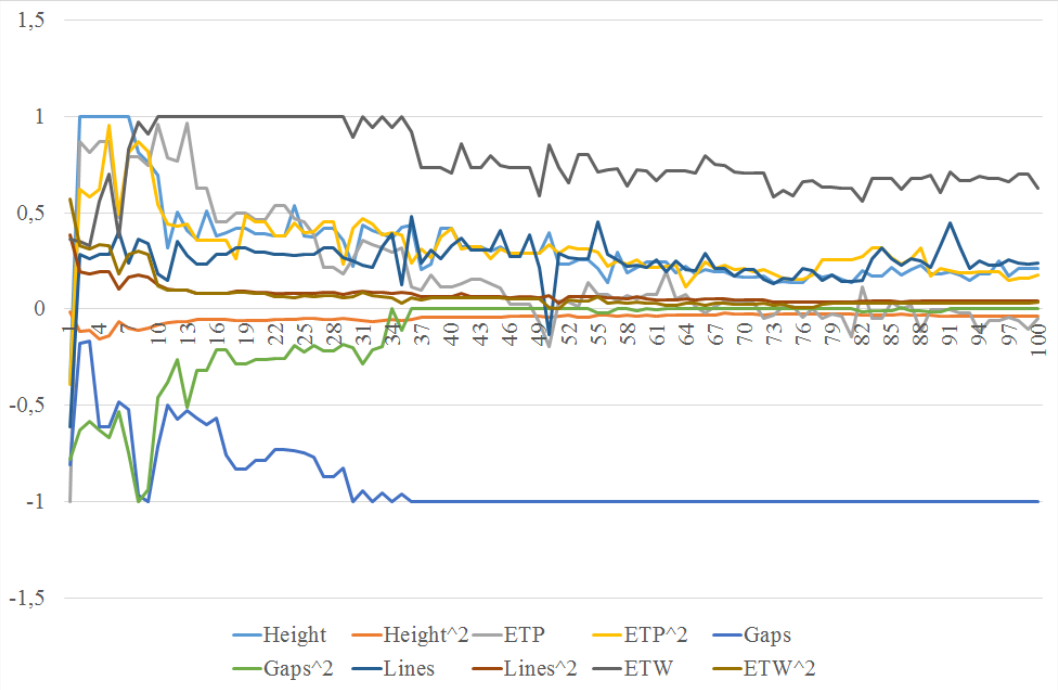


Figura 6.12: Valori dei geni migliori delle popolazioni con genoma esteso

Le informazioni contenute nell'immagine 6.12 sono preziose per capire la strategia di gioco dell'intelligenza artificiale. I geni sono disposti su 3 diversi livelli di importanza: i geni dominanti sono Gaps ed ETW, in linea con i risultati ottenuti nella sezione 6.2.2; un gruppo di geni composti da ETP^2, Lines ed Height hanno minore importanza ma sono significativi per il loro feedback positivo; in ultimo un folto gruppo che comprende $Lines^2$, ETP, ETW^2, $Gaps^2$ ed $Height^2$ non ha praticamente alcun ruolo nel processo decisionale.

Per quanto riguarda il gene ETP^2 è necessario fare una precisazione. Nonostante non sia tra quelli dominanti, il fatto che riguardi un parametro non lineare gli attribuisce comunque un feedback importante. Il parametro ETP corrisponde al numero di spigoli adiacenti ai blocchi sottostanti e assume un valore compreso tra 0 e 9. Il suo quadrato è compreso tra 0 e 81, il che gli conferisce un certo peso nel calcolo dello

score della mossa. Nella tabella 6.2 e nell'immagine 6.13 viene mostrato il genoma dell'unità migliore della generazione 75.

Height	$Height^2$	ETP	ETP^2	ETW	ETW^2	Gaps	$Gaps^2$	Lines	$Lines^2$
0,1396	-0,0254	0,0135	0,1552	0,5870	0,0072	-1,0000	0,0002	0,1545	0,0368

Tabella 6.2: Cromosoma dell'unità migliore della generazione 75

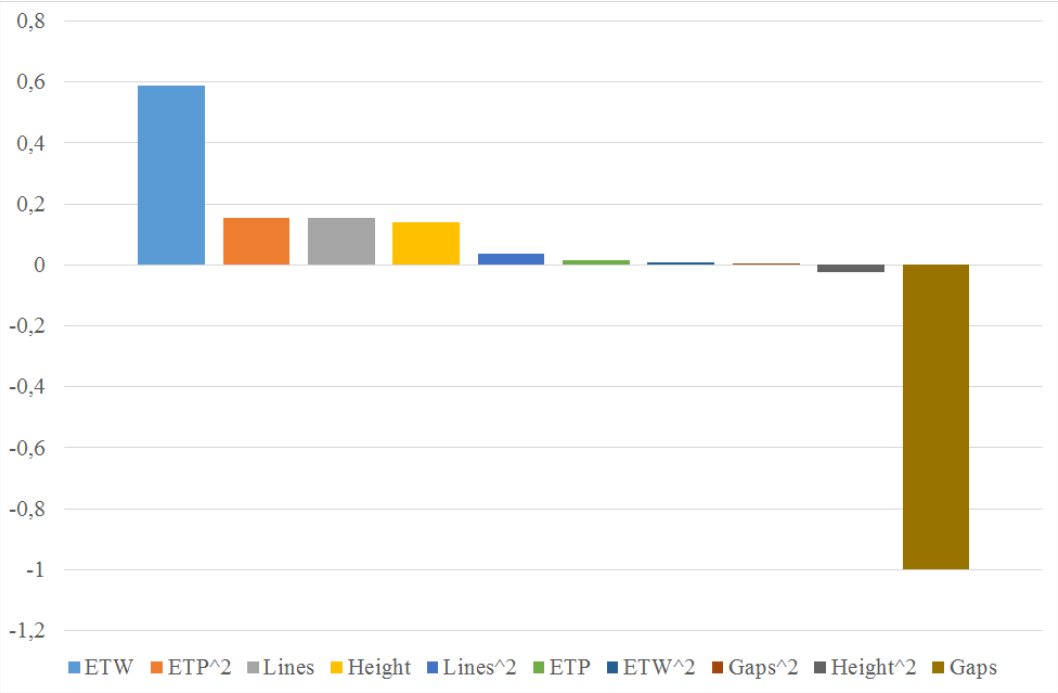


Figura 6.13: Cromosoma dell'unità migliore della generazione 75

E' evidente come il processo evolutivo abbia attuato un processo di selezione delle informazioni. Il 50% dei geni non viene utilizzato, tra questi la maggior parte sono geni non lineari. Questo fa capire come le unità facciano uso solo del valore ETP^2, ignorando gli altri geni aggiunti al cromosoma. Molto sorprendente è il valore del gene Height, che a differenza degli esperimenti con il cromosoma standard, è positivo.

Complessivamente i risultati del genoma esteso sono incoraggianti, nella sezione 6.3.2 lo si metterà a confronto con il genoma standard evidenziando le differenze sia nel comportamento delle unità sia nelle prestazioni.

6.3.2 Confronto cromosoma standard ed esteso

Per mettere a confronto il cromosoma standard con quello esteso è necessario focalizzare l'attenzione sulla metrica che governa l'ottimizzazione per punteggio: i punti per pezzo.

La figura 6.14 aggiunge al grafico 6.10 i punti per pezzo delle unità della generazione 75 ordinate per fitness media. Il cromosoma esteso si dimostra capace di superare i risultati del genoma standard con prestazioni mediamente superiori del 3%. Il margine non sembra particolarmente rilevante, ma se si mettono a confronto i grafici della fitness media delle due popolazioni risulta evidente la differenza che intercorre tra i valori.

Come risulta evidente dalla figura 6.15, le unità che fanno parte della popolazione con cromosoma esteso ottengono risultati mediamente migliori rispetto a quelle con il cromosoma standard. È possibile concludere che il genoma esteso, nonostante la velocità di convergenza sia 5 volte inferiore rispetto al genoma standard, è in grado di adattarsi meglio all'obbiettivo e ottenere un numero maggiore di punti per pezzo.

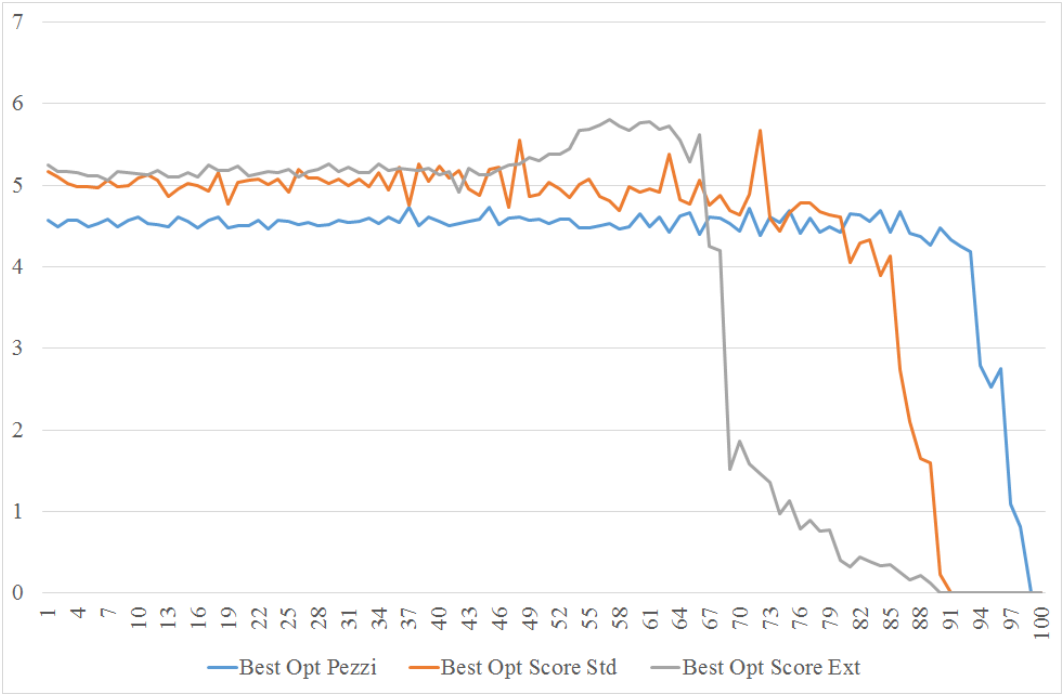


Figura 6.14: Confronto punti per pezzo tra cromosoma standard ed esteso

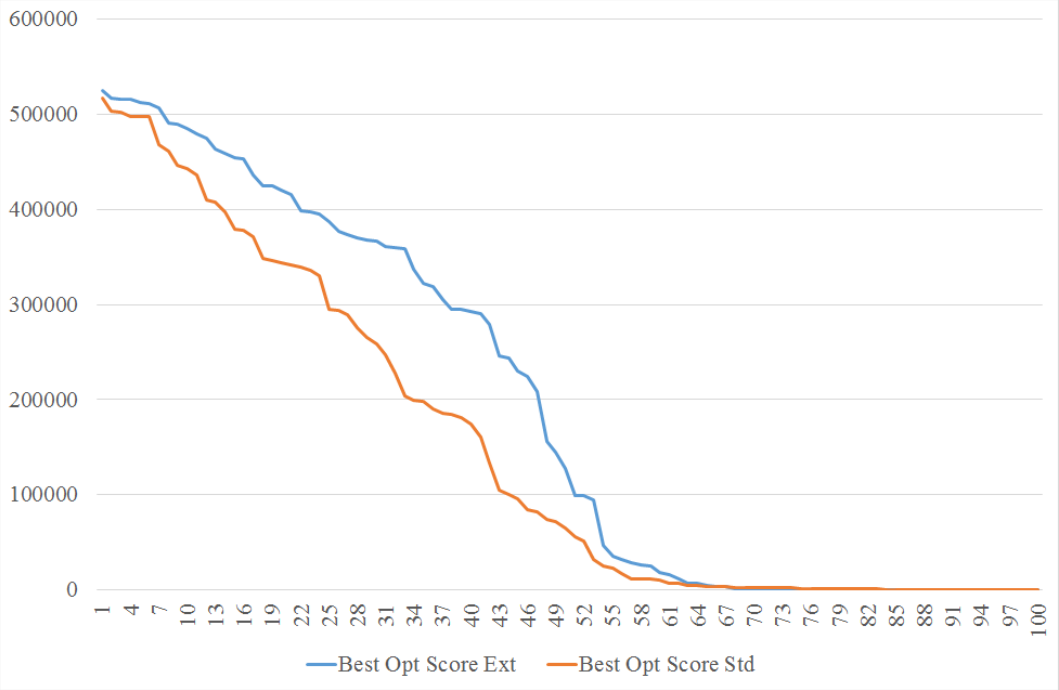


Figura 6.15: Confronto fitness media tra cromosoma standard ed esteso

6.4 Ripetibilità

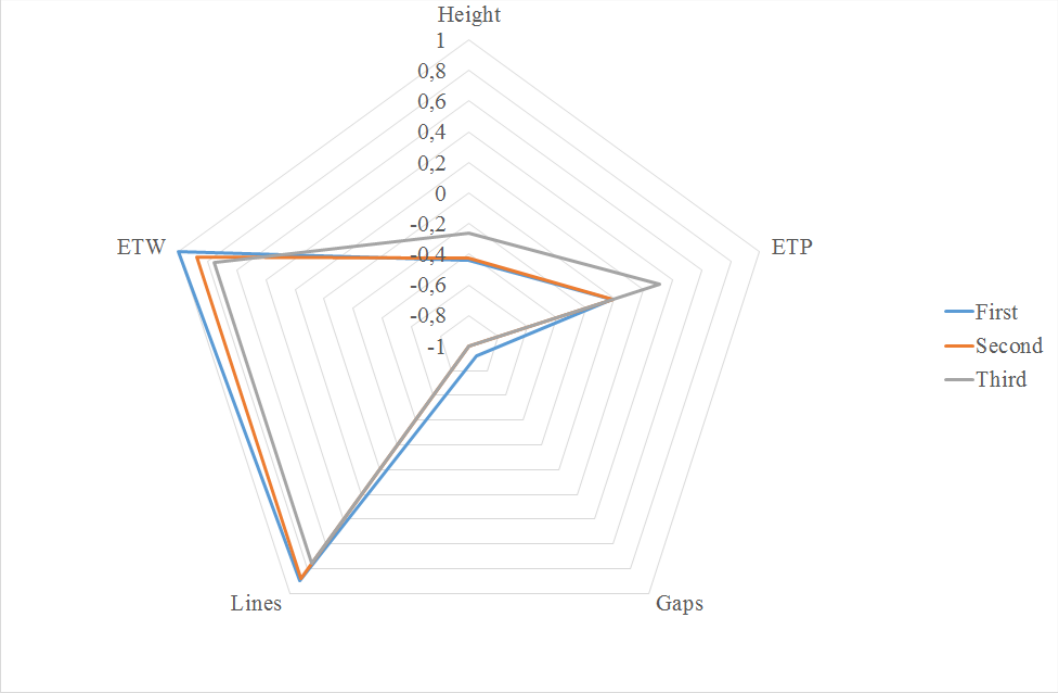
La ripetibilità di un esperimento è molto importante per verificare la sua validità. Nel caso in esame sono state condotte 3 prove analoghe di evoluzione su una popolazione casuale con lo stesso obbiettivo per scoprire se i risultati trovati sono consistenti tra loro. L'obbiettivo dell'evoluzione è l'ottimizzazione per numero di pezzi con un tetto massimo di 100.000 tetramini e per ogni esperimento sono state completate 6 generazioni.

	Height	ETP	ETW	Gaps	Lines
First	-0,4416	-0,0161	1,0000	-0,9154	0,8930
Second	-0,4264	-0,0117	0,8762	-1,0000	0,8762
Third	-0,2596	0,3110	0,7589	-1,0000	0,7544
%	18,20%	32,72%	24,11%	8,45%	13,86%

Tabella 6.3: Confronto geni unità migliori di tre prove diverse di evoluzione

Come è possibile vedere nei grafici presenti nell'immagine 6.16 e nella tabella 6.3, i tre cromosomi sono molto simili ma non identici. In particolare, confrontando i valori dei geni ETP, si nota una differenza di oltre il 32%. Tutte e tre le unità migliori hanno una fitness pari a circa 510.000 punti, con differenze che non superano mai il 3%.

(a) Grafico a linee



(b) Grafico radar

Figura 6.16: Confronto geni unità migliori di tre prove diverse di evoluzione

Per assicurarsi di avere un algoritmo genetico consistente sono stati effettuati ulteriori test comparativi con due popolazioni diverse. In questo caso è stato cambiato il metodo di mutazione da SwapBlend a DoublePoint, ma l'ottimizzazione è sempre per punteggio e sono state completate 20 generazioni.

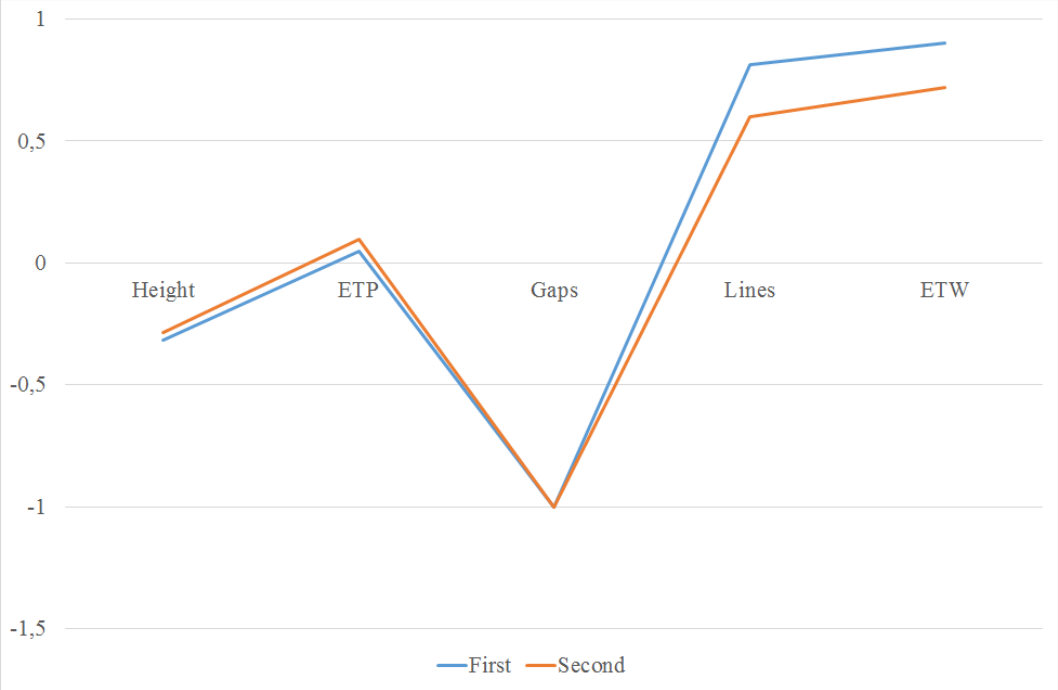
	Height	ETP	ETW	Gaps	Lines
First	-0,3182	0,0483	0,9027	-1,0000	0,8127
Second	-0,2847	0,0976	0,7198	-1,0000	0,6001
%	3,35%	4,93%	18,29%	0,00%	21,27%

Tabella 6.4: Confronto geni unità migliori di tre prove diverse di evoluzione

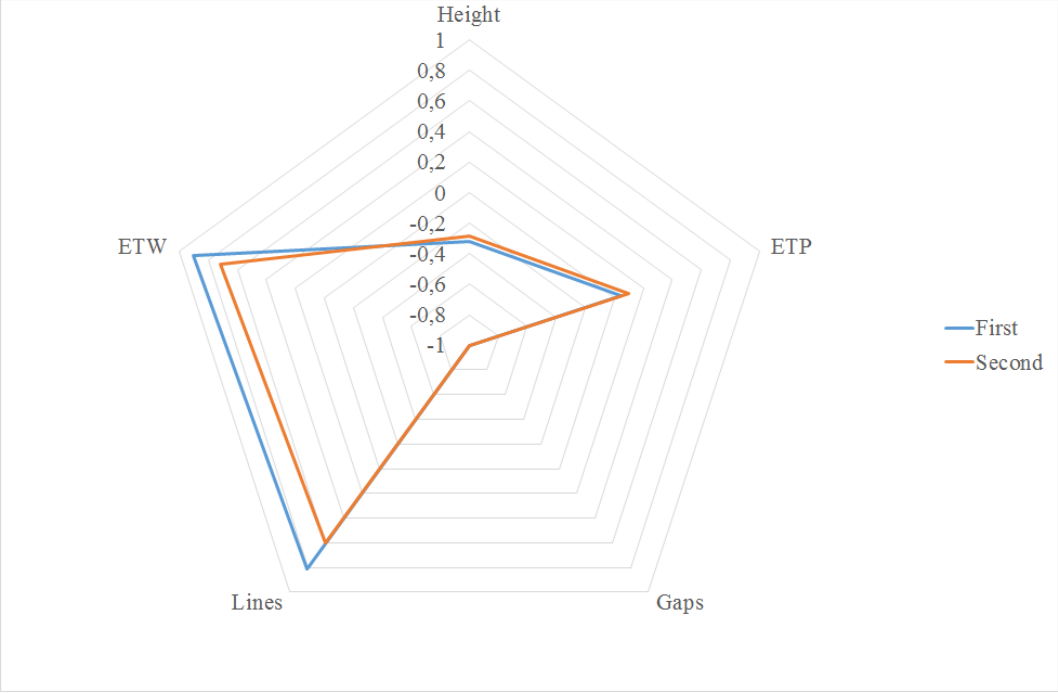
Analizzando i dati presenti nei grafici della figura 6.17 e nella tabella 6.4 è possibile notare che solo 2 geni su 5 (ETW e Lines) hanno differenze intorno al 20%. I valori dei geni Height ed ETW si possono considerare molto simili in quanto esiste una differenza inferiore al 5%. Per quanto riguarda il gene Gaps non ci sono differenze tra le due unità e questo significa che in entrambi i casi il valore feedback negativo del parametro è dominante su tutti gli altri. Per concludere al di là di ogni ragionevole dubbio che l'algoritmo genetico produce risultati totalmente riproducibili sarebbe necessario effettuare più prove in modo da stabilire la variabilità intrinseca dei parametri. Dagli esperimenti condotti è possibile affermare, con buona approssimazione, la consistenza del processo evolutivo. Si deve sempre considerare che gli algoritmi genetici sono un metodo di ricerca casuale ed è veramente improbabile che due popolazioni per quanto simili seguano lo stesso percorso evolutivo.

6.5 Analisi complessità tetramini

Nella sezione 3.3 sono stati presentati i risultati di una ricerca che conclude che il tetris è difficile anche solo da approssimare. Lo scopo dei prossimi esperimenti è quello di stimare la difficoltà dei tetramini osservando il comportamento dell'intelligenza artificiale. Collegandosi a quanto detto nella sezione 4.3.1.3, la distribuzione di probabilità dei tetramini viene alterata per modificare l'ambiente di evoluzione. Sono stati condotti 7 esperimenti diversi rimuovendo di volta in volta un tetramino diverso dal generatore di numeri casuali. Nelle prossime sezioni verranno analizzati i risultati ottenuti dalle prove effettuate e si cercherà d stabilire quali sono i pezzi più difficili da gestire e quali possono essere considerati «simili» dal punto di vista della complessità. I parametri di evoluzione sono gli stessi degli esperimenti condotti nella sezione 6.2.2.



(a) Grafico a linee



(b) Grafico radar

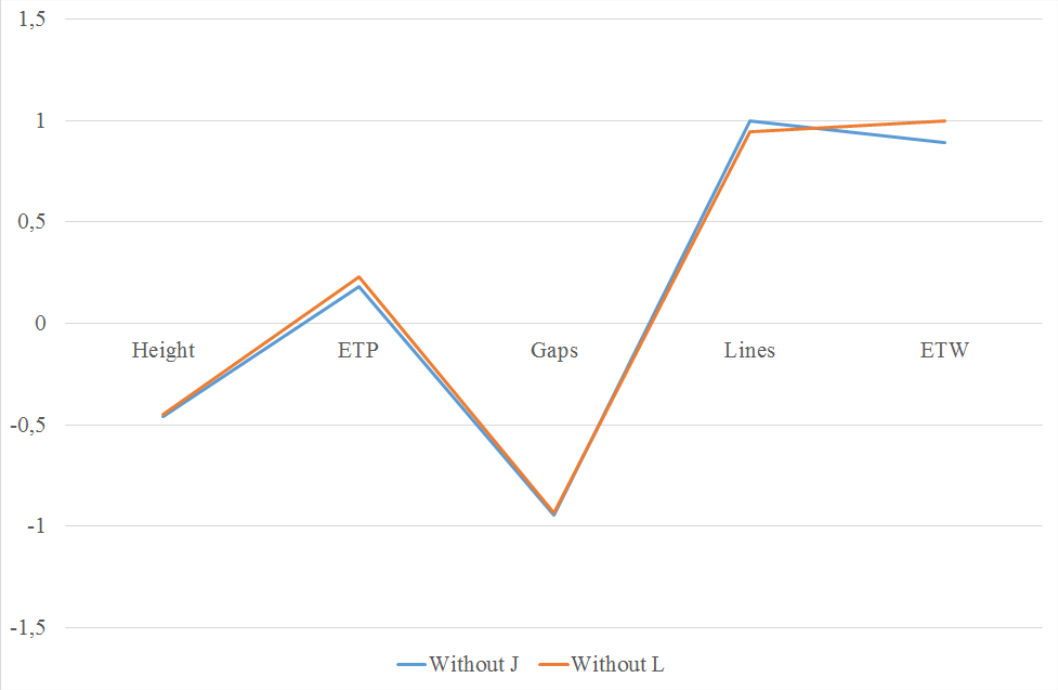
Figura 6.17: Confronto geni unità migliori di due prove diverse di evoluzione

6.5.1 Tetramini J ed L

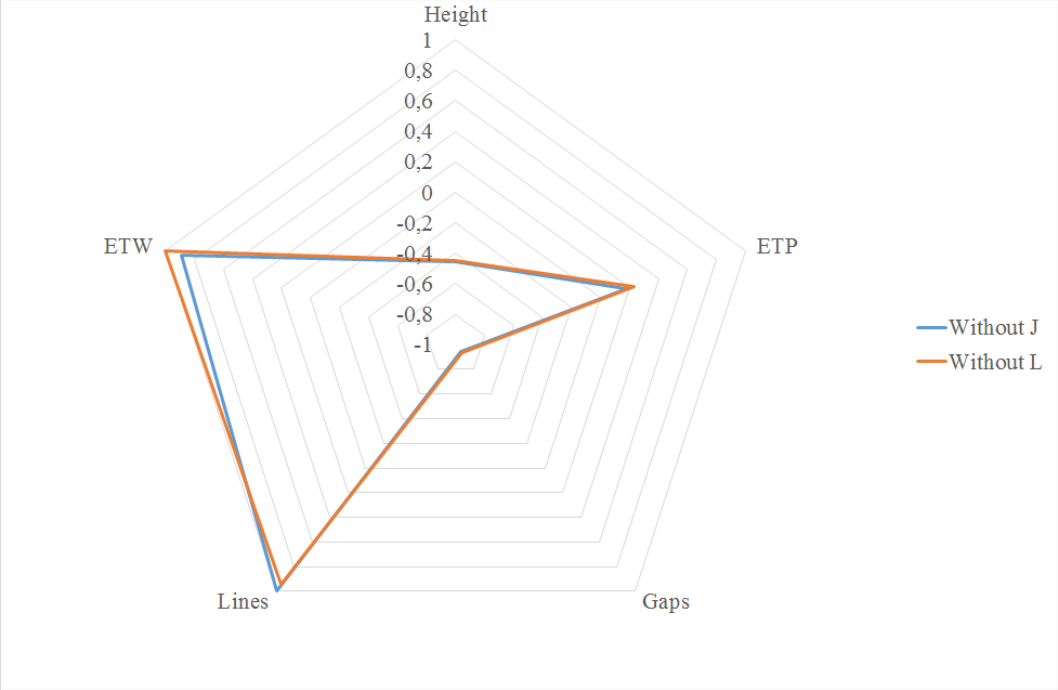
I tetramini L e J sono composti dagli stessi blocchi ma sono speculari tra loro. Sarebbe logico aspettarsi che, anche dal punto di vista della complessità, l'intelligenza artificiale li gestisca allo stesso modo. Le unità prese in esame per questo confronto sono le migliori della generazione 20 del processo evolutivo.

Nei grafici presenti nell'immagine 6.18 e nella tabella 6.5 è possibile notare come le differenze tra le prestazioni e la composizione dei cromosomi delle due unità siano molto contenute. A parte il valore del gene ETW che si differenzia di circa il 10% tutte le altre discrepanze sono trascurabili in quanto perfettamente assimilabili alla variabilità intrinseca del processo di evoluzione.

Da questi risultati si può concludere che il tetramino J e il tetramino L presentano lo stesso tipo di difficoltà di gestione e quindi dal punto di vista della complessità sono analoghi.



(a) Grafico a linee



(b) Grafico radar

Figura 6.18: Confronto cromosomi di unità evolute senza J e unità evolute senza L

	Fitness	Height	ETP	ETW	Gaps	Lines
No J	498.900	-0,4566	0,1816	0,8928	-0,9437	1,0000
No L	492.943	-0,4460	0,2286	1,0000	-0,9343	0,9441
%	1,19%	1,05%	4,69%	10,72%	0,95%	5,59%

Tabella 6.5: Confronto cromosomi di unità evolute senza J e unità evolute senza L

6.5.2 Tetramini Z ed S

Ripetendo la premessa fatta nella sezione 6.5.1, anche i tetramini Z ed S sono composti dagli stessi blocchi ma sono speculari tra loro. Si attendono dei risultati che confermino la tesi dell'equivalenza di complessità tra i due tetramini. Anche in questo esperimento sono state prese in considerazione le migliori unità della generazione 20.

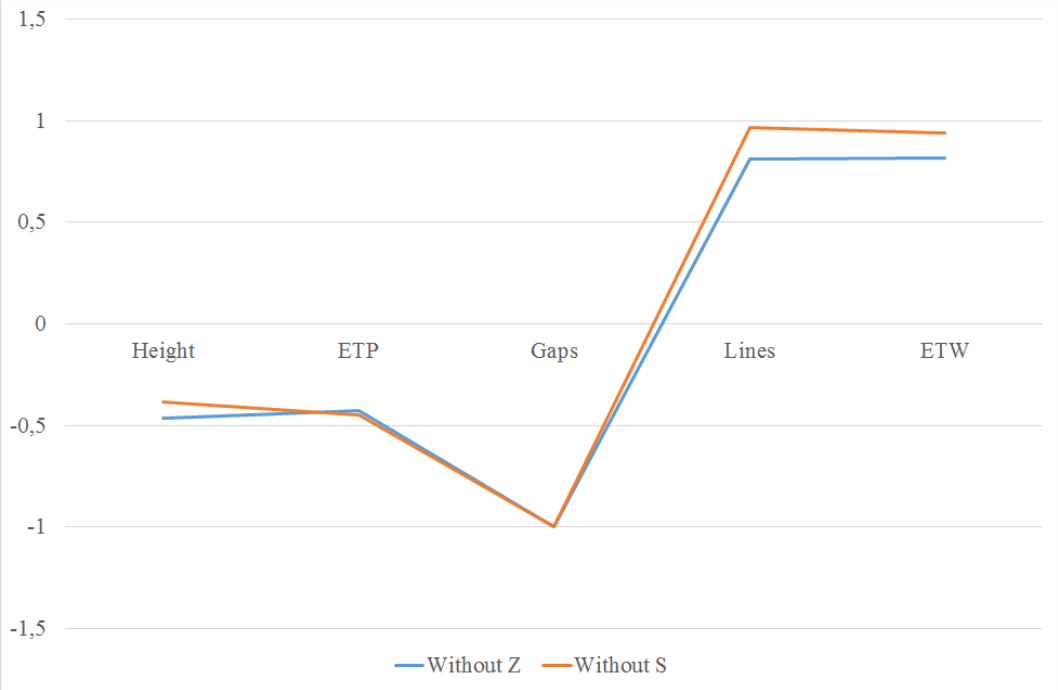
	Fitness	Height	ETP	ETW	Gaps	Lines
No Z	545.200	-0,4622	-0,4261	0,8154	-1,000	0,8136
No S	561.783	-0,3831	0,4495	0,9386	-1,000	0,9667
%	3,04%	7,91%	2,33%	12,32%	0,00%	15,31%

Tabella 6.6: Confronto cromosomi di unità evolute senza Z e unità evolute senza S

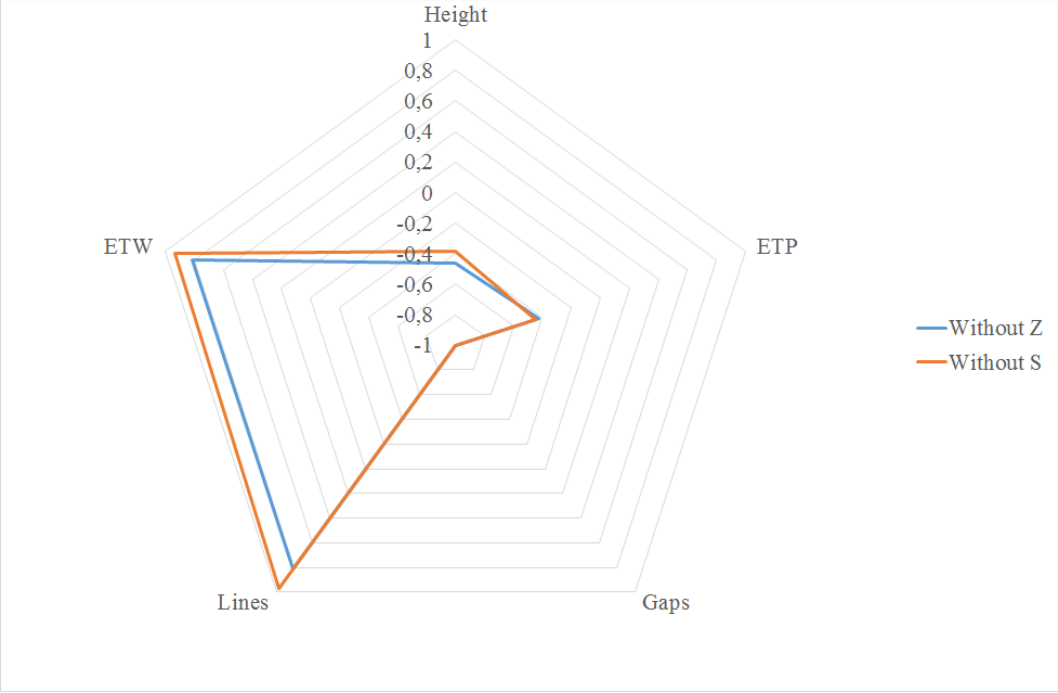
Nei grafici presenti nell'immagine 6.19 e nella tabella 6.6 è possibile evidenziare nessuna differenza significativa tra le prestazioni e la composizione dei cromosomi delle due unità. I geni che al confronto risultano più distanti sono ETW e Lines, ma con differenze contenute intorno al 12% e 15%, gli altri geni sono tutti molto simili con discrepanze massime del 7%

Da questi risultati si può concludere che il tetramino Z e il tetramino S presentano lo stesso tipo di difficoltà di gestione e quindi dal punto di vista della complessità sono analoghi. Inoltre il punteggio di fitness ottenuto eliminando anche solo uno di questi due tetramini è molto più alto del caso con tutti i pezzi visto nella sezione 6.2.2.

Collegandosi a quanto detto nella sezione 3.3, senza uno dei due tetramini della «snake sequence» che porta alla sconfitta certa del giocatore è possibile ottenere punteggi sensibilmente più elevati. Per verificare questa ipotesi è stato condotto un esperimento eliminando entrambi i tetramini S e Z, i risultati del processo evolutivo sono presenti nell'immagine 6.20



(a) Grafico a linee



(b) Grafico radar

Figura 6.19: Confronto cromosomi di unità evolute senza Z e unità evolute senza S

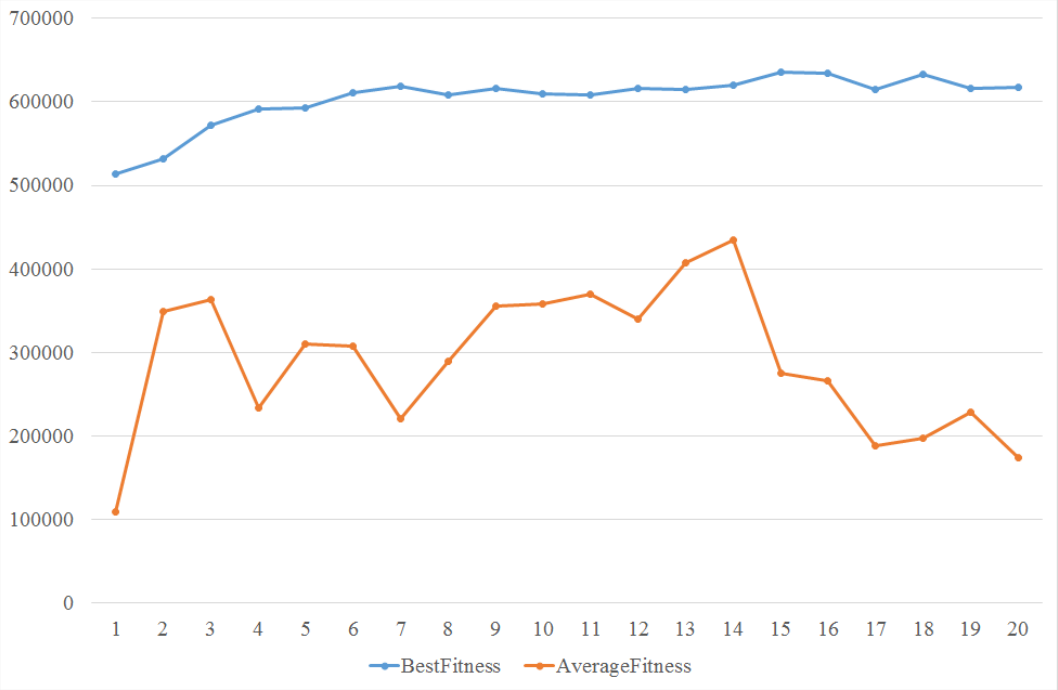


Figura 6.20: Fitness media e massima popolazione senza i tetramini S e Z

Senza i tetramini che fanno parte della «snake sequence» il processo evolutivo è in grado di ottenere risultati di gran lunga superiori rispetto a qualsiasi altro caso analizzato. Il valore della fitness massima si incontra nella generazione 15 ed è pari a 635.723 punti ovvero il 21% maggiore rispetto al caso con tutti i pezzi a disposizione (525.363 punti) visto nella sezione 6.2.2. Anche i punti per pezzo sono sorprendenti: 6,3572 contro 5,1633, oltre il 23% maggiore. I risultati di questo esperimento confermano senza ombra di dubbio la tesi esposta nel paragrafo precedente: senza i pezzi S e Z è possibile ottenere punteggi notevolmente più elevati.

6.5.3 Tetramino I

Il tetramino I è un pezzo molto importante del Tetris, infatti è l'unico che permette di completare 4 linee contemporaneamente e quindi ottenere molti punti con una sola mossa. La domanda principale riguarda le prestazioni dell'intelligenza artificiale senza questo tetramino così importante ai fini del punteggio. L'ipotesi è che il valore della fitness sarà notevolmente inferiore rispetto a tutti gli altri casi analizzati.

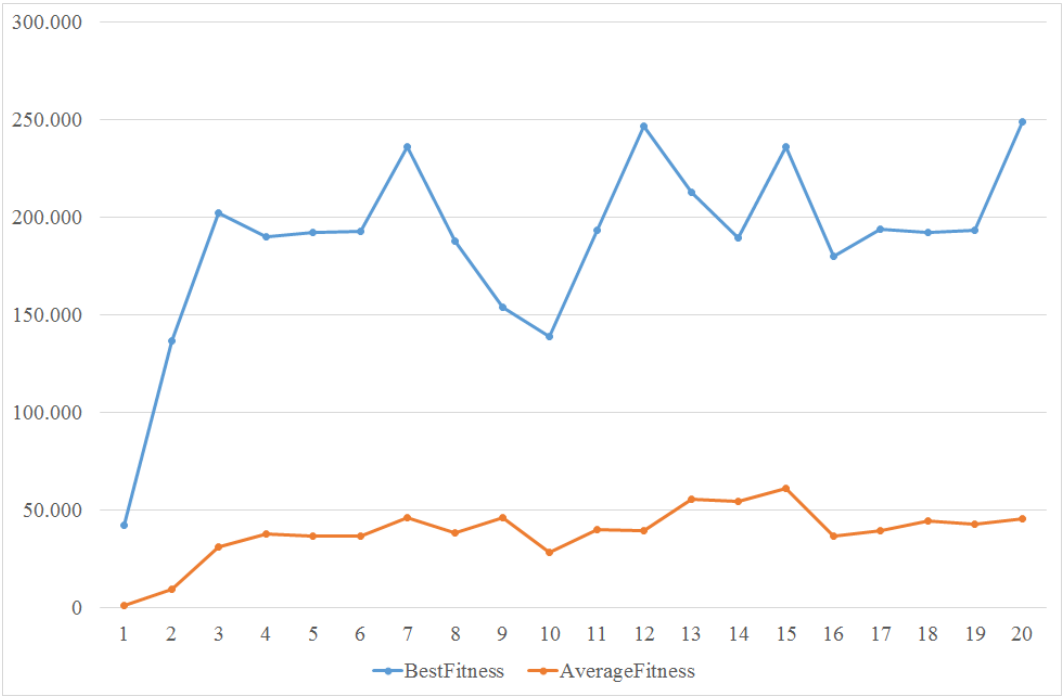


Figura 6.21: Fitness media e massima popolazione senza il tetramino I

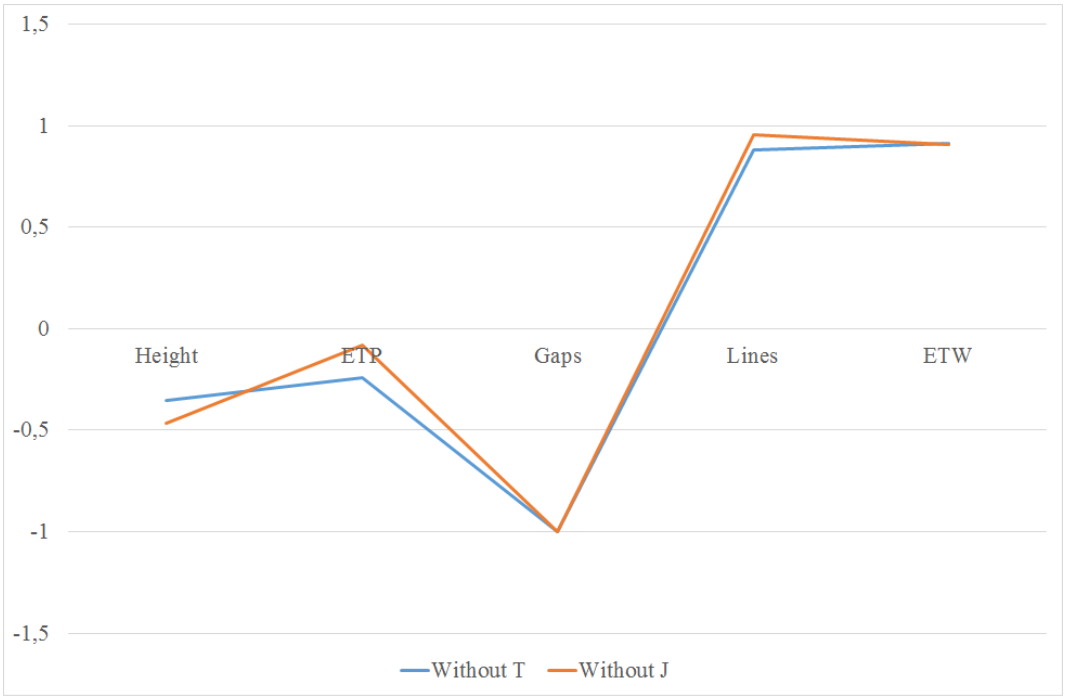
Nella figura 6.21 è possibile seguire l'andamento dell'evoluzione della popolazione se privata del tetramino I. La fitness media rimane molto bassa e allo stesso modo la fitness massima continua a oscillare intorno a valori vicini a 200.000, ovvero circa 2 volte e mezzo inferiore rispetto al caso con tutti i pezzi della sezione 6.2.2. Dati i risultati di questo esperimento la tesi riguardante le performance scarse delle unità se private del pezzo I è confermata.

6.5.4 Tetramini T e J

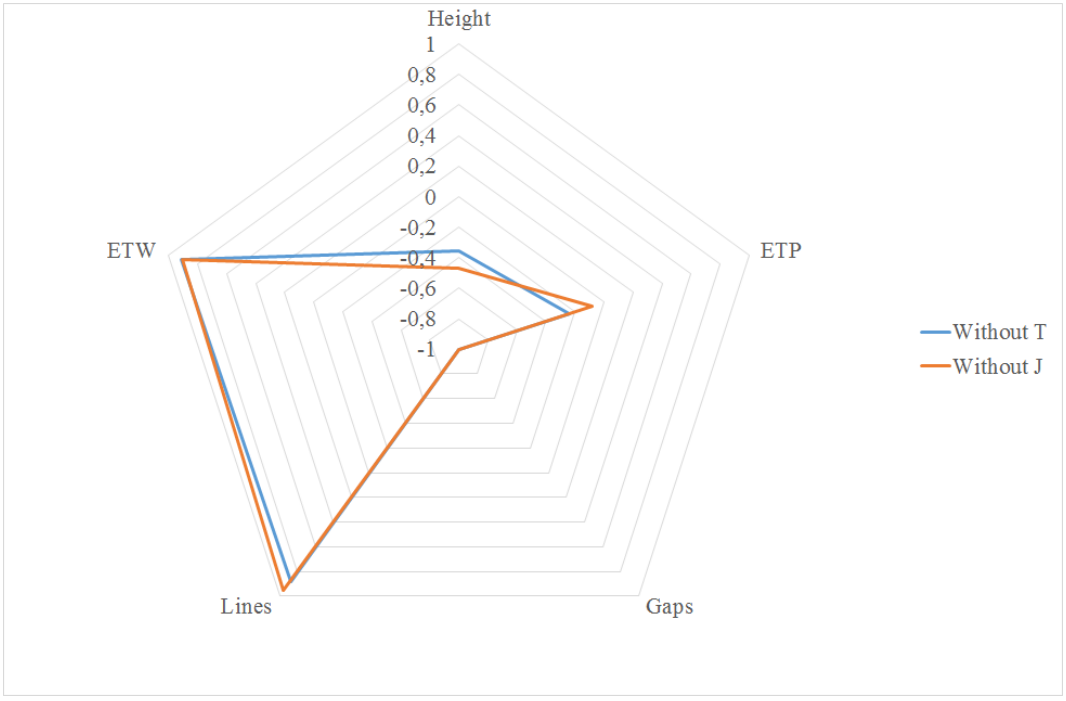
Se si confrontano i tetramini T e J si possono trovare molte analogie interessanti. Non essendo pezzi speculari il risultato di questo esperimento è tutt'altro che scontato. Se si pensa alla loro forma, sia T che J sono in grado di completare 3 linee contemporaneamente, quindi almeno sulla quantità di punti ottenibili potrebbero essere analoghi, infatti quello che li differenzia è la posizione di un solo blocco. Nel seguente esperimento vengono prese in esame le unità migliori di tutto il processo evolutivo senza J e senza T, entrambe presenti nella generazione 10.

	Fitness	Height	ETP	ETW	Gaps	Lines
No T	532.953	-0,3559	-0,2402	0,9113	-1,000	0,8802
No J	516.476	-0,4668	-0,0834	0,9053	-1,000	0,9551
%	3,19%	11,09%	15,68%	0,60%	0,00%	7,49%

Tabella 6.7: Confronto cromosomi di unità evolute senza J e unità evolute senza T



(a) Grafico a linee



(b) Grafico radar

Figura 6.22: Confronto cromosomi di unità evolute senza J e unità evolute senza T

Nei grafici presenti nell'immagine 6.22 e nella tabella 6.7 si può notare come non ci siano differenze sostanziali né tra i valori dei geni né tra le performance delle unità. Questo risultato è senza dubbio il più curioso trovato fino ad ora, infatti dai dati ottenuti è possibile concludere che il tetramino J, il tetramino T e quindi anche il tetramino L sono analoghi dal punto di vista della complessità.

Capitolo 7

Conclusioni

7.1 Riassunto risultati

L'architettura software vista nel capitolo 5 è stata di fondamentale importanza nella compilazione dei risultati degli esperimenti. L'implementazione binaria dello schema del tetris ha permesso di triplicare la velocità di esecuzione permettendo di completare tutti gli esperimenti nella finestra temporale a disposizione.

Dagli esperimenti effettuati nel capitolo 6 sono state tratte molte informazioni preziose riguardanti l'evoluzione dell'intelligenza artificiale per il gioco del Tetris. Nella sezione 6.2 sono stati condotti due diversi esperimenti per verificare il processo evolutivo il cui obbiettivo era l'ottimizzazione per numero di pezzi e l'ottimizzazione per punteggio. Nel primo caso l'evoluzione ha richiesto molto tempo a causa del grande numero di mosse che le unità sono in grado di effettuare prima di perdere la partita e le sequenze di tetromini in uscita dal generatore di numeri casuali hanno conferito al grafico della fitness massima un andamento erratico. Per mitigare l'influenza della casualità sul corso della partita è necessario aumentare il numero di valutazioni della funzione di fitness per avere dei dati statistici più affidabili.

Il secondo esperimento, esaminato nella sezione 6.2.2, riguarda l'ottimizzazione per punteggio. L'obbiettivo di questa evoluzione è aumentare il numero di punti per pezzo ottenuto dalle unità durante la partita. I risultati hanno mostrato come l'intelligenza artificiale ritenga il valore del parametro ETP di scarsa importanza e per questo motivo non prende sostanzialmente parte al processo decisionale. La strategia di gioco che si è evoluta è molto interessante in quanto utilizza il pezzo I per completare linee multiple inserendolo ai lati dello schema. Dal confronto tra l'unità migliore proveniente dall'ottimizzazione per numero di pezzi e il campione dell'ottimizzazione per punteggio si dimostra effettivamente che l'evoluzione ha preso la direzione indicata incrementando il numero di punti per pezzo.

Il cromosoma dei primi due esperimenti è composto dai geni Height, ETP, ETW, Gaps e Lines. Nell'esperimento successivo della sezione 6.3 si è operata un'estensione

del cromosoma standard aggiungendo ulteriori geni che controllano parametri non lineari. Lo scopo di questa estensione è di fornire all'intelligenza artificiale i mezzi per reagire non linearmente ai segnali in ingresso. L'obbiettivo della simulazione è l'ottimizzazione per punteggio e la prima osservazione che si evince dai risultati dell'esperimento è la diversa velocità di convergenza. Con un cromosoma composto da 10 geni al posto che 5 sono state necessarie 100 generazioni per stabilizzare il grafico della fitness. Dai risultati ottenuti è emerso che il processo evolutivo ha attuato una selezione delle informazioni eliminando dal processo decisionale il 50% dei parametri in ingresso. Tra i 5 geni non lineari $Height^2$, ETP^2, ETW^2, $Gaps^2$ e $Lines^2$ solo ETP^2 viene effettivamente utilizzato, gli altri hanno valori molto vicini allo zero. La fitness delle unità dotate di cromosoma esteso si è rivelata sensibilmente superiore rispetto a quelle con genoma standard. Questo significa che le informazioni contenute nei geni non lineari apportano un effettivo vantaggio rispetto al cromosoma composto da soli geni lineari.

Per verificare la ripetibilità del processo evolutivo sono state condotte diverse prove con le stesse condizioni iniziali. I dati mostrati nella sezione 6.4 hanno dimostrato la consistenza dei risultati dell'evoluzione anche se i percorsi seguiti per ottenere quei risultati sono stati fondamentalmente diversi a causa della natura di ricerca casuale dell'algoritmo genetico.

Nella sezione 6.5, per stabilire la complessità dei tetramini sono stati effettuati diversi esperimenti modificando la distribuzione di probabilità del generatore di numeri casuali. Il primo obbiettivo è stato dimostrare se i tetramini speculari, ovvero J - L e Z - S, presentano lo stesso tipo di difficoltà. I risultati ottenuti confermano questa ipotesi, inoltre è stato scoperto che il tetramino T appartiene alla stessa classe di complessità dei tetramini J ed L. Per quanto riguarda il tetramino I, è stato dimostrato che è il pezzo più importante per raggiungere l'obbiettivo dell'ottimizzazione per punteggio in quanto permette di completare 4 linee contemporaneamente con una sola mossa.

Il gioco del Tetris nasconde, nella sua apparente semplicità, molte caratteristiche di giochi di gran lunga più complessi. La realizzazione di un'intelligenza artificiale in grado di ottenere i punteggi visti nel capitolo 6 permette di approfondire tutti gli aspetti del gioco.

A seguire i risultati[1] delle unità migliori evolute durante gli esperimenti condotti:

[1] **std**: cromosoma standard, **ext**: cromosoma esteso. **all**: tutti i tetromini, **no S e Z**: senza i tetromini S e Z. **no limit**: nessun limite di pezzi, **max 100k**: massimo 100.000 pezzi. **t**: tetromini, **ppt**: punti per tetromino

Ottimizzazione	Condizioni	Risultato
Numero di pezzi	std, all, no limit	63.217.070 fitness e 8.753.484 t
Punteggio	std, all, max 100k	517.710 fitness e 5,177 ppt
Punteggio	std, no S e Z, max 100k	646.440 fitness e 6,457 ppt
Punteggio	ext, all, max 100k	525.363 fitness e 5,254 ppt

Tabella 7.1: Riassunto risultati migliori

7.2 Sviluppi futuri

La finestra temporale a disposizione degli esperimenti del capitolo 6 ha permesso di ottenere risultati molto interessanti e significativi. Il numero massimo di generazioni è stato fissato a 20 perchè la maggior parte dei processi evolutivi analizzati sono in grado di raggiungere l'equilibrio in questo lasso di tempo. Senza dubbio, avendo a disposizione una finestra temporale più ampia, è possibile incrementare il numero di generazioni simulate al fine di ottenere risultati migliori.

Per aumentare la validità statistica delle misurazioni della fitness sarebbe interessante incrementare il numero di valutazioni. Un numero compreso tra 10 e 20 misurazioni permetterebbe di stabilire con maggiore precisione la qualità del cromosoma, mitigando le grandi oscillazioni viste, ad esempio, nella figura 6.4.

Sono stati condotti esperimenti per stabilire la bontà dell'estensione del cromosoma solo nel caso dell'ottimizzazione per punteggio. Sarebbe interessante confrontare le prestazioni del cromosoma esteso nel caso dell'ottimizzazione per numero di pezzi, al fine di verificare se i risultati ottenuti sono migliori del cromosoma standard.

L'estensione del cromosoma è stata realizzata aggiungendo 5 geni non lineari: $Height^2$, ETP^2, ETW^2, $Gaps^2$ e $Lines^2$. Sarebbe interessante includere nel cromosoma altri geni, come ad esempio i cubi o i logaritmi dei geni standard. E' stato mostrato nella figura 6.13 come il processo evolutivo sia in grado di selezionare le informazioni che gli vengono fornite. Ma è da tenere in considerazione che la velocità di convergenza dell'algoritmo si riduce notevolmente al crescere delle dimensioni del cromosoma. Oltre a riutilizzare in forma non lineare i geni standard, sarebbe utile aggiungere geni di diversa tipologia. Ad esempio si potrebbe considerare l'aggiunta di un gene che indica il rapporto tra gli spazi presenti al di sotto del pezzo corrente e quelli del resto dello schema.

L'analisi di complessità dei tetramini ha permesso di evidenziare le strategie adottate dell'intelligenza artificiale al variare della distribuzione di probabilità. Tutti i pezzi speculari sono analoghi, e allo stesso modo anche L, J e T vengono affrontati con la stessa strategia. Un'estensione interessante di questo esperimento è ripetere l'analisi con tutte le combinazioni di tetramini. Quindi non solo rimuovendo completamente un singolo pezzo dal pool ma anche modificando con continuità la distribuzione di probabilità per tutti i tetramini.

Al di là delle estensioni dei singoli esperimenti, è possibile replicare lo stesso approccio evolutivo variando le caratteristiche del gioco del Tetris. Sarebbe interessante osservare le strategie impiegate dall'intelligenza artificiale se lo schema fosse di dimensione diversa, oppure se i pezzi fossero dei pentamini o esamini. Un altro possibile sviluppo è quello di far evolvere le unità in un ambiente tridimensionale come nel caso del Tetris 3D.

La programmazione evolutiva e in particolare gli algoritmi genetici aprono nuove strade nel campo dell'intelligenza artificiale. È stata mostrata la loro capacità di adattamento a problemi privi di un modello matematico di riferimento come il Tetris. Le prestazioni di un'intelligenza artificiale basata su modelli sono limitate dalla compresione del problema, al contrario un algoritmo genetico non parte da alcuna assunzione o conoscenza pregressa. Questa condizione gli consente di esplorare lo spazio delle soluzioni senza pregiudizi, riuscendo ad ottenere risultati tanto buoni quanto sorprendenti.

Bibliografia

[1] Amine Boumaza. How to design good tetris players. 2013.

[2] Ron Breukelaar, Erik D. Demaine, Susan Hohenberger, Hendrik Jan Hoogeboom, Walter A. Kosters, and David Liben-nowell. Tetris is hard, even to approximate. *International Journal of Computational Geometry and Applications*, 2004.

[3] Ron Breukelaar, Hendrik Jan Hoogeboom, and Walter A. Kosters. Tetris is hard, made easy. Technical report, Leiden Institute of Advanced Computer Science, Universiteit Leiden, 2003.

[4] John Brzustowski. Can you win at tetris? Master's thesis, Master's Thesis, University of British Columbia, 1992.

[5] Heidi Burgiel. How to lose at tetris. *Mathematical Gazette*, 81:194–200, 1997.

[6] Landon Flom and Cliff Robinson. Using a genetic algorithm to weight an evaluation function for tetris.

[7] David Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.

[8] Randy Haupt and Sue Ellen Haupt. *Practical Genetic Algorithms.* John Wiley & Sons, 1998.

[9] Tetris History. Wikia. `http://tetris.wikia.com/wiki/History`.

[10] John H. Holland. *Adaptation in Natural and Artificial Systems.* The MIT Press, 1975.

[11] Hendrik Jan Hoogeboom and Walter A. Kosters. The theory of tetris.

[12] Peter H. Lewis. New software game: It comes from soviet. *The New York Times*, 1988.

[13] Bai Li. Coding a tetris ai using a genetic algorithm, 2011. `https://luckytoilet.wordpress.com/2011/05/27/coding-a-tetris-ai-using-a-genetic-algorithm/`.

[14] Melanie Mitchell. An introduction to genetic algorithms. *MIT Press*, 1996.

[15] Stefan Mandl Niko Böhm, Gabriella Kókai. An evolutionary approach to tetris. The Sixth Metaheuristics International Conference, 2005.

[16] Aleksej Leonidovič Pažitnov. Wikipedia. `https://it.wikipedia.org/wiki/Aleksej_Leonidovi%C4%8D_Pa%C5%BEitnov`.

[17] David Rollinson and Glenn Wagner. Tetris ai generation using nelder-mead and genetic algorithms. 2010.

[18] Tetris Scoring. Wikia. `http://tetris.wikia.com/wiki/Scoring`.

[19] Tetris. Wikipedia. `https://it.wikipedia.org/wiki/Tetris`.

[20] Lee Yiyuan. Tetris ai - the (near) perfect bot. `https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/`, 2013.