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

DESIGN E SVILUPPO DI IRON DUKE: UN GIOCO GESTIONALE PER DISPOSITIVI MOBILI

Relatore: Prof. Pier Luca LANZI
Correlatore: Spartaco ALBERTARELLI

Tesi di Laurea di:
Gabriele Arnaboldi, matricola 765568

Anno Accademico 2013-2014

Sommario

Il mondo dei videogiochi ha da sempre offerto sfide interessanti nel campo dell'intelligenza artificiale. Sebbene quello dell'intelligenza artificiale "accademica" e quello della *Game AI* siano due settori che possano a volte sembrare di viaggiare su due binari separati, è indubbio come i punti in comune certamente esistano e come il settore dei videogiochi possa trarre vantaggio dalla ricerca e, in alcuni casi, anche viceversa. In questo elaborato viene quindi presentato lo stato dell'arte nel campo delle intelligenze artificiali nel settore videoludico, ne vengono mostrate le principali sfide e le principali tecniche utilizzate per affrontarle. Vengono poi descritte le modalità con cui è stato disegnato e sviluppato il videogioco *Iron Duke*: un gioco manageriale-strategico ambientato nell'inghilterra del diciannovesimo secolo basato sul trasporto ferroviario. Infine viene descritto in che modo è stato affrontato il problema di implementare un motore di intelligenza artificiale che fosse in grado di giocare ad una partita di *Iron Duke* in modo autonomo.

Abstract

Videogame industry has always offered compelling challenges to the developers that had to implement the behaviour of the actors partecipating in their games. Even though "*academic*" *AI* and "*gaming*" *AI* often share different goals and methods, it can be easily seen that the two worlds does have a lot of points in common and that one discipline can take benefits from the methods and the knowledge provided by the other. As an introduction of the work that has been done, a brief description of the current state of the art in the gaming AI field will be provided, with a detailed insight into all of its main goals and the techniques typically used to reach them. The thesis continues with a presentation of the videogame *Iron Duke*, a business-strategic game based on the railroad transport in the England of the 19th century and entirely developed in the context of this work. The thesis concludes with a detailed explaination of the work done to implement an artificial intelligence engine able to play one game of *Iron Duke* by its own, and the results of the tests that has been done to check its funtionality.

Ringraziamenti

Ringrazio innanzitutto il prof. Lanzi per avermi offerto l'opportunità di lavorare su un'argomento che amo, per avermi fatto da mentore in questi anni e per essere sempre stato un punto fisso di riferimento sia per me, sia per tutti i ragazzi che condividono la mia passione per lo sviluppo di videogames, settore molto spesso preso poco seriamente e snobbato dalla gente, ma che per fortuna sta emergendo ed assumendo la giusta importanza che merita anche grazie a persone come Pier Luca Lanzi che sono riuscite a diventare il punto di contatto tra industria, mondo accademico e appasionati del settore.
Ringrazio poi Spartaco Albertarelli con cui ho avuto il privilegio di collaborare nella realizzazione di questo lavoro e che reputo indubbiamente una delle persone più interessanti con cui ho avuto il piacere di parlare, una mente eccelsa, e con la fortuna di fare, a mio parere, uno dei lavori più belli del mondo. Ringrazio infine mio padre per avermi dato l'opportunità, non certo senza grossi sacrifici, di arrivare fin qui e di completare questo difficile percorso che spero mi ripagherà negli anni che verranno...

Indice

Sommario **i**

Abstract **iii**

Ringraziamenti **v**

Elenco delle figure **xi**

Elenco delle tabelle **xv**

1 Introduzione **1**
1.1 Scopo e struttura della tesi 2

2 Stato dell'arte **5**
2.1 L'Intelligenza Artificiale per giochi: l'arte dell'illusione 6
2.2 Un modello generico per l'Intelligenza Artificiale 8
2.2.1 Il problema del movimento 9
2.2.2 Il problema decisionale 11
2.2.3 Il problema strategico 14
2.3 Tecniche e strumenti 15
2.3.1 Espedienti e soluzioni ad-hoc 16
2.3.2 Euristiche 19
2.3.3 Algoritmi e modelli 21
2.4 I.A. specifica per board games 28
2.4.1 Il problema del movimento nei board games 29
2.4.2 Il problema decisionale-strategico nei board games . . 32
2.5 Conclusioni 36

3 Descrizione del gioco Iron Duke **37**
3.1 Descrizione del gioco 37
3.1.1 Obiettivo del gioco 38
3.1.2 Meccanica di gioco 38

3.2 Struttura ed elementi di gioco 41
3.2.1 Mappa 41
3.2.2 Città 43
3.2.3 Rete ferroviaria 47
3.2.4 Treni 48
3.3 Il modello economico 52
3.3.1 Modello implementativo 53
3.3.2 Il mercato delle merci 55
3.3.3 Passeggeri e posta 57
3.3.4 I costi di costruzione 59
3.3.5 I costi di manutenzione 61
3.3.6 Premi e punteggio 63

4 L'intelligenza artificiale in Iron Duke 65
4.1 Il modello di base 66
4.2 Il problema del movimento: l'algoritmo di pathfinding in Iron Duke 69
4.3 Il problema decisionale-strategico: la struttura modulare . . . 76
4.3.1 Il modulo 77
4.3.2 Il sistema dei goal 80
4.3.3 La macchina a stati 84
4.3.4 Il meccanismo a scambio di messaggi 85

5 Playtesting e simulazioni 91
5.1 Test 1: equilibrio di gioco e risposta della I.A. 91
5.1.1 Test 1.1: risposta e aggressività 92
5.2 Test 2: efficienza degli algoritmi di costruzione 94
5.2.1 Test 2.1: valutazione del punto iniziale di costruzione 94
5.2.2 Test 2.2: legame tra la scelta posizione iniziale e la profondità della ricerca 97
5.2.3 Test 2.3: costruzione di segmenti distaccati 99
5.3 Test 3: politiche di costruzione dei convogli 100
5.4 Test 4: livelli di abilità dell'I.A. 103
5.4.1 Test 4.1: partita a due giocatori 104
5.4.2 Test 4.2: partita a quattro giocatori 105

6 Conclusioni e sviluppi futuri 109
6.1 Conclusioni 109

Bibliografia 111

A Algoritmi e Pseudocodice **115**
A.1 Algoritmo A* 115
A.2 Dijkstra 116
A.3 Alpha-Beta Pruning 117

B Lista dei parametri utilizzati dall'intelligenza artificiale **119**

Indice analitico **122**

Elenco delle figure

2.1 Modello generico di implementazione di una I.A. 8
2.2 Waypoints e Mesh Navigazionali a confronto 11
2.3 Modello generico di decision making 11
2.4 Esempio di Behavior Tree 13
2.5 Analisi strategica basata su waypoints 14
2.6 Esempio di mappa di influenza 15
2.7 Muovere centinaia di unità non significa necessariamente utilizzare centinaia di algoritmi di pathfinding 18
2.8 Esempio di applicazione dell'euristica 20
2.9 Pathfinding con algoritmo A* 22
2.10 A* Gerarchico 23
2.11 Algoritmo di Dijkstra nella tecnica di map flooding 24
2.12 Esempio di Macchina a Stati 25
2.13 Task di selezione e sequenza in un Behavior Tree 26
2.14 Esempio di albero comportamentale complesso che utilizza i task di selezione e sequenza 27
2.15 Porzione dell'albero del gioco del tris 29
2.16 Matematica alla base di scacchiere e mappe esagonali 31
2.17 Esempio di riferimento alle celle adiacenti in un board game . 31
2.18 Albero del gioco dei tre politici 34
2.19 Esempio di gioco a somma zero 35

3.1 Diagramma UML del sistema di gestione dei turni 39
3.2 Schematizzazione della meccanica di gioco a turni in Iron Duke 40
3.3 Mappa di gioco di Iron Duke 41
3.4 Diagramma UML del modello di gestione della mappa 43
3.5 Schema del modello di domanda e offerta di una città 45
3.6 I diversi livelli di binario 47
3.7 Diagramma UML del modello di gestione dei treni in Iron Duke 49
3.8 Diagramma UML del modello di gestione del budget 53

3.9 Diagramma dei costi in Iron Duke 54
3.10 Analisi dei prezzi delle merci in giochi simili ad Iron Duke . . 55
3.11 Grafico dell'andamento del prezzo di una merce in funzione dei ritardi di consegna 57
3.12 Ripartizione dei bonus per passeggeri e posta 58
3.13 Grafici di utilità dei diversi livelli di binario 61
3.14 Ripartizione del valore dei premi in Iron Duke 64

4.1 Modello base dell'intelligenza artificiale in Iron Duke 66
4.2 Diagramma di flusso della gestione dell'esecuzione dell'intelligenza artificiale in Iron Duke 67
4.3 Diagramma UML del modello di IA utilizzato in Iron Duke . 68
4.4 Esempi di applicazione dell'algoritmo di pathfinding in Iron Duke con diverse opzioni: algoritmo puro, ottimizzato per non creare percorsi artificiosi, ottimizzato per attraversare le città vicine. 72
4.5 Esempi di applicazione dell'opzione passThroughNearbyCities nell'algoritmo di pathfinding in Iron Duke 73
4.6 Ulteriori esempi di applicazione dell'algoritmo di pathfinding in Iron Duke 76
4.7 Flusso di esecuzione di un modulo di I.A. 80
4.8 Visualizzazione grafica di una delle distribuzioni dei goal usata in Iron Duke 81
4.9 Esempio di collaborazione tra i moduli dell'intelligenza artificiale 85
4.10 Una possibile soluzione (esecuzione sequenziale) al problema della comunicazione tra moduli 86
4.11 L'esecuzione atomica e sequenziale delle azioni di I.A. potrebbe causare latenza 86
4.12 Una possibile implementazione (esecuzione parallela) del motore di intelligenza artificiale 87
4.13 Diagramma UML delle classi dei messaggi utilizzati dal motore di intelligenza artificiale in Iron Duke 89
4.14 Esempio di esecuzione di una azione nel motore di intelligenza artificiale in Iron Duke 90

5.1 Rappresentazione grafica del distacco tra i giocatori rilevato dal primo playtest 92
5.2 Distacco tra i giocatori e punteggio massimo al variare del parametro di aggressività 93

5.3 Differenza di costruzione al variare del parametro di aggressività 94
5.4 Rappresentazione grafica della zona iniziale di costruzione da parte del modulo di intelligenza artificiale e relativo risultato finale. 97
5.5 Grafici dei dati ottenuti incrociando i parametri di costruzione iniziale e probabilità di costruire reti distaccate. 100
5.6 Confronto tra le varie politiche di costruzione dei convogli adottate dalla I.A. 102
5.7 Efficienza di guadagno in relazione alle politiche di costruzione dei convogli adottate dalla I.A. 103
5.8 Risultati delle simulazioni a 2 giocatori artificiali con diversi livelli di abilità 105
5.9 Risultati delle simulazioni a 4 giocatori artificiali con diversi livelli di abilità (prima prova) 107
5.10 Risultati delle simulazioni a 4 giocatori artificiali con diversi livelli di abilità (seconda prova) 108

Elenco delle tabelle

3.1 Parametri descrittivi di una città in Iron Duke 43
3.2 Popolazione storica (1860) e livello iniziale delle città. 46
3.3 Tabella di efficienza delle locomotive in Iron Duke 49
3.4 Parametri principali delle locomotive in Iron Duke 50
3.5 Elenco di tutte le voci economiche di Iron Duke 52
3.6 Ripartizione teorica dei costi 53
3.7 Rapporto tra i prezzi delle merci in Iron Duke 55
3.8 Fasce di bonus per il trasporto di passeggeri e posta 57
3.9 Rapporto tra i prezzi di costruzione della rete ferroviaria in Iron Duke 61

5.1 Differenza media di punteggio finale tra giocatori comandati da I.A. 92
5.2 Differenza media di punteggio finale al variare della politica di scelta del punto iniziale di costruzione. 95
5.3 Vantaggio del giocatore che effettua la prima mossa utilizzando una politica intelligente di scelta della posizione iniziale di costruzione. 96
5.4 Differenza del vantaggio del primo giocatore al variare del parametro MAX_CONNECTION_CHAIN 98
5.5 Risultati ottenuti incrociando i parametri di costruzione iniziale e probabilità di costruire reti distaccate. 99
5.6 Confronto tra le varie politiche di costruzione dei convogli adottate dalla I.A. 101
5.7 Modifica dei parametri dell'I.A. al fine di equilibrare i livelli di difficoltà 108

B.1 Lista dei parametri utilizzati dall'intelligenza artificiale (Modulo di gestione della flotta) 119

B.2 Lista dei parametri utilizzati dall'intelligenza artificiale (Modulo di costruzione) 120
B.3 Lista dei parametri utilizzati dall'intelligenza artificiale (Modulo di business) 121

Non smettiamo di giocare perchè diventiamo vecchi,
diventiamo vecchi perchè smettiamo di giocare.

Anonimo [1]

[1] La frase viene spesso attribuita (probabilmente erroneamente) a figure quali Benjamin Franklin, Oliver Wendell Holmes, Herbert Spencer e George Bernard Shaw

Capitolo 1

Introduzione

Sono passati oramai diversi decenni da quando i primi videogiochi commerciali videro la luce, ma già allora risultò subito evidente come questi prodotti rappresentassero un campo fertile sul quale affrontare e sperimentare in modo pratico le sfide teoriche offerte dal mondo dell'intelligenza artificiale. Sebbene sia ancora argomento di discussione se e in che modo i due mondi dell'intelligenza artificiale cosiddetta "accademica" e quella "per giochi" (*Game AI*) siano effettivamente legati, lo studio di casi specifici dimostra come in effetti tale legame non possa essere del tutto ignorato. È vero anche, tuttavia, che entrando più nel dettaglio della ricerca, si nota come il progresso dell'intelligenza artificiale "per giochi" sia abbastanza stagnante e di come l'industria riutilizzi ormai da tempo tecniche di I.A. oramai ben note e consolidate. Il motivo di questo fatto va però attribuito ad un semplice fatto: i moderni giochi sono ormai sempre più veloci e frenetici, e il giocatore interagisce con agenti comandati da I.A. per frazioni di tempo molto brevi (un nemico deve comparire sullo schermo giusto per il tempo necessario ad essere sconfitto dal giocatore). Spesso non c'è quindi interesse da parte dello sviluppatore nell'implementare un'intelligenza artificiale innovativa o complessa, che paradossalmente potrebbe rivelarsi addirittura controproducente e rendere il prodotto finale meno interessante dal punto di vista commerciale

e meno giocabile. Esistono ovviamente anche casi in cui è il gioco stesso, che per via della sua particolare meccanica, richiede l'implementazione di una I.A. allo stato dell'arte, è il caso questo di quei giochi nei quali l'intelligenza artificiale è *costantemente* sotto agli occhi del giocatore, come per esempio i giochi strategici, i board games, o i giochi in cui l'I.A. gioca *con* il giocatore e non *contro* di esso.

1.1 Scopo e struttura della tesi

Lo scopo di questa tesi è quello di implementare un buon motore di intelligenza artificiale per un gioco completamente nuovo ed originale e che prende spunto da meccaniche tipiche dei giochi da tavolo, e che rientra quindi nella seconda categoria dei giochi appena elencati. Si tenterà quindi di utilizzare sia tecniche "standard" nel campo della game AI, sia, dove possibile, tecniche un po' più innovative in risposta alle varie sfide offerte dalla struttura del gioco. Il lavoro svolto consiste quindi nel design e nello sviluppo di un gioco partendo da zero e la successiva implementazione di un'intelligenza artificiale in grado di sfidare il giocatore umano. La tesi è strutturata nel modo seguente:

- Nel capitolo due si illustra lo stato dell'arte nel campo della game AI, si analizzano quali sono i problemi principali gli obiettivi di un motore di intelligenza artificiale, e se ne fornisce poi un modello generico. Si studia poi come l'industria affronta tali problemi e quali sono gli strumenti e le tecniche principali a disposizione dello sviluppatore. Per alcune di queste tecniche vengono poi forniti esempi pratici e spiegazioni dettagliate. Alla fine del capitolo vengono illustrate alcune tecniche specifiche utili nel caso particolare dei board games e che prendono spunto in gran parte dal campo di ricerca della teoria dei giochi .

- Nel capitolo tre viene presentato il gioco Iron Duke , ne vengono descritte le regole e ne viene presentata la struttura e le meccaniche. Vengono poi analizzati uno ad uno i diversi elementi che compongono il gioco e viene spiegato come essi siano stati implementati, per concludere infine con un'analisi matematica del modello economico alla base del gioco.

- Il capitolo quattro è dedicato interamente al motore di intelligenza artificiale: viene descritto il modello generico utilizzato e viene spiegato come i problemi posti dal gioco siano stati affrontati e risolti. In seguito vengono descritte dettagliatamente le componenti che formano il motore di I.A. e come esse siano state implementate. Vengono descritti il sistema basato su goal, la macchina a stati ed il meccanismo a scambio di messaggi.

- Nel capitolo cinque vengono riportati i risultati ottenuti in fase di playtest modificando i vari parametri che dettano il comportamento dell'intelligenza artificiale, al fine di validare il modello utilizzato e di trovare le configurazioni più competitive e che possano garantire l'implementazione di diversi livelli di abilità da parte dei giocatori comandati dalla CPU.

- Nelle conclusioni si riassumono gli scopi, le valutazioni di questi e le prospettive future.

- Nell'appendice A si riportano alcuni degli algoritmi più comunemente utilizzati nell'ambito dello sviluppo di un'intelligenza artificiale per giochi.

- Nell'appendice B si elencano tutti i parametri che descrivono il comportamento dell'intelligenza artificiale e se ne fornisce una breve descrizione.

Capitolo 2

Stato dell'arte

Si è sempre dibattuto molto su cosa il settore di ricerca sull'intelligenza artificiale possa fare per il mondo dei videogiochi ed il consenso a riguardo pare essere univoco: "*non molto*" [18]. In effetti, analizzando l'industria videoludica, ci si rende conto di come la maggior parte dei prodotti commerciali faccia ancora affidamento a tecniche ed algoritmi ormai superati, come per esempio le macchine a stati finiti, e che l'introduzione di tecniche più moderne e ricercate come reti neurali ed algoritmi genetici non sempre sono sinonimo di un buon prodotto finale. Affermare tuttavia che le due discipline viaggino su due strade completamente distinte è forse anch'essa un'affermazione troppo forte, in quanto il settore dei videogiochi può trarre indubbiamente vantaggio dalle tecniche e dai risultati di ricerca nel campo dell'intelligenza artificiale. Dal lato opposto è interessante invece notare come negli ultimi anni stia sempre più crescendo un interesse piuttosto inusuale, ovvero come proprio il mondo della ricerca utilizzi i videogiochi come strumento di studio per applicare ed analizzare nuove soluzioni nel campo dell'intelligenza artificiale.

In questo capitolo verranno analizzati i problemi principali che ci si trova a dover affrontare quando si vuole implementare un motore di intelligenza artificiale, ne verrà presentato un modello di risoluzione generico, e verranno infine analizzate più in dettaglio alcune delle tecniche più comunemente utilizzate dall'industria odierna.

2.1 L'Intelligenza Artificiale per giochi: l'arte dell'illusione

Uno dei punti chiave che sembra mettere d'accordo chiunque lavori sia nel campo dei videogiochi che in quello della ricerca [8, 19, 16], è che i concetti, le tecniche e soprattutto gli obiettivi finali dell'intelligenza artificiale per prodotti di tipo videoludico si discostano molto dalle loro controparti nel settore dell' intelligenza artificiale cosiddetta "tradizionale" o "accademica" , tant'è che si potrebbe addirittura parlare di due domini separati. Se si analizzano le produzioni commerciali correnti o degli ultimi 15 anni si ha l'idea che le sfide poste nel campo dell'intelligenza artificiale siano state in gran parte risolte tramite l'utilizzo di tecniche che non necessariamente seguono o sono ispirate dagli avanzamenti accademici in questo settore [34], e le tecnologie utilizzate sono spesso ormai vecchie e obsolete: il tutto si riduce molto spesso a voler ottenere dei comportamenti più "artificiali" che "intelligenti". L'obiettivo di una I.A. di tipo ludico si traduce quasi sempre nel voler emulare il comportamento di altri giocatori (I.A. che riveste in tutto e per tutto il ruolo di un avversario umano in un gioco competitivo) oppure nel controllare le entità di gioco non direttamente controllate da essi (I.A. che controlla i cosiddetti *non player character* o NPC) e, molto raramente lo sviluppatore di un intelligenza artificiale deve preoccuparsi di creare agenti realmente senzienti o dotati di auto-consapevolezza, e mai in nessun caso un' agente dovrà possedere o acquisire conoscenze che vanno al di fuori del contesto di gioco [19]. L'unica grande sfida per uno sviluppatore di intelligenza artificiale è quella di riuscire a creare una qualcosa che *simuli un comportamento in grado di dare al giocatore l'illusione di trovarsi di fronte ad un entità dotata di intelligenza*, e comportamenti ingiustificati da parte di tali entità possono facilmente rompere questa illusione ed immediatamente ridurre la partecipazione e l'immersione del giocatore [2]. Concetti interessanti da tenere in considerazione saranno quelli di *trasparenza psicologica* [18] e di *finestra percettiva* [16] del giocatore. Come esseri umani percepiamo un'entità come intelligente poichè siamo in grado di comprenderne i comportamenti e le emozioni, e una buona I.A. dovrà quindi essere

trasparente al giocatore in questo senso, ovvero non dovrà mai nascondergli il proprio stato d'animo o le proprie intenzioni. Inoltre, l'intelligenza artificiale dovrà unicamente rivestire il ruolo per il quale essa ha ragione di esistere all'interno del gioco e *per il solo tempo in cui essa riceverà attenzione da parte del giocatore*, dovrà quindi aver "vita" solamente all'interno della finestra percettiva di quest'ultimo. Non è quindi sempre scontato che intelligenze complesse, realistiche o correttamente implementate secondo i canoni accademici siano sinonimo di un miglior risultato finale in termini di gioco, ed inoltre è sempre buona regola distinguere attentamente intelligenze artificiali *ottime* da intelligenze artificiali *divertenti*: spesso, infatti, l'obiettivo di una buona intelligenza artificiale è quello di assicurare che la loro *sconfitta* sia percepita dal giocatore come un'esperienza divertente, e un'intelligenza artificiale che gioca per perdere non è certo sinonimo di una ottimalità nella sua implementazione. Al giocatore deve essere sufficiente credere che la sua vittoria sia frutto esclusivamente dei propri sforzi e delle proprie capacità, e non dovuta ad una "falla" (inserita volutamente) della I.A. Caso opposto ed ugualmente interessante è invece quello in cui l'intelligenza artificiale gioca *con* il giocatore e non *contro* di esso. In un contesto simile, vedi per esempio titoli come Bioshock Infinite [Irrational Games, 2013] o Black and White [Lionhead Studios, 2001], il giocatore avrà costantemente sotto agli occhi il comportamento dell'intelligenza artificiale e la sua finestra percettiva si estenderà quindi dall'inizio della partita fino alla sua fine. In questi casi sarà quindi necessario implementare una I.A. allo stato dell'arte [18] che tragga magari beneficio da tecniche più avanzate e vicine al mondo accademico.

2.2 Un modello generico per l'Intelligenza Artificiale

Un modello generico di implementazione di una I.A. per videogiochi è quello illustrato in Figura 2.1:

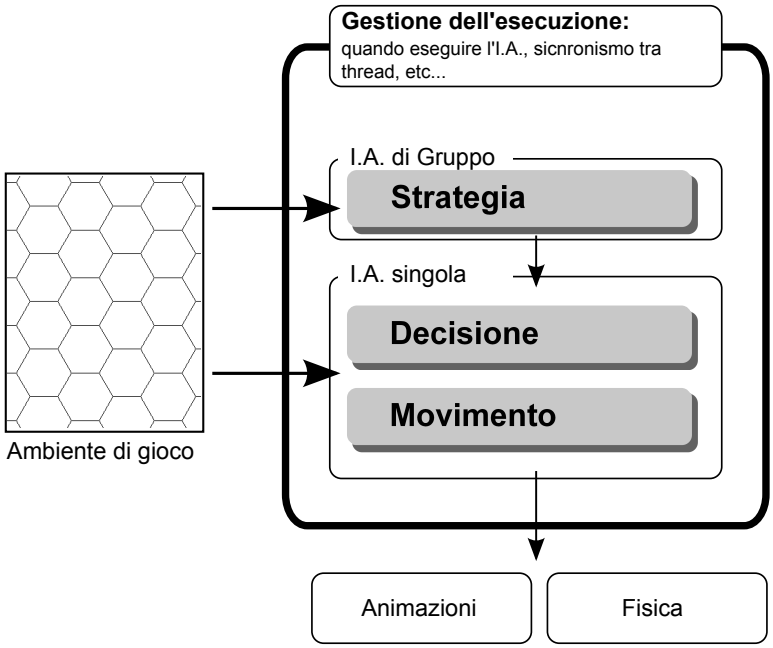


Figura 2.1: Modello generico di implementazione di una I.A.

Per quanto gli obiettivi finali ed i metodi utilizzati possano differire di gioco in gioco, il problema di sviluppare una solida intelligenza artificiale può essere generalmente sempre suddiviso in tre principali sotto-problemi:

- Il problema del **movimento** e del relativo *pathfinding* : i vari elementi di gioco dovranno sapere *come* muoversi.
  Da notare che domande come *quando* e *dove* muoversi non trovano qui ancora risposta, mentre sono invece problemi validi quesiti del tipo *i personaggi si muovono singolarmente o in gruppo?*, *il loro movimento sarà influenzato anche dagli altri personaggi?*, *quanto realistico dovrà essere il movimento e di quanta fisica avremo bisogno?*

- Il problema **decisionale** (decision-making): un agente controllato da I.A. può avere libero arbitrio e decidere autonomamente come comportarsi in base agli stimoli che riceve dall'ambiente di gioco. Questa sezione comprende anche le domande lasciate in sospeso riguardo al

movimento (quando muoversi? dove?), mentre altre domande tipiche sono: *quale dovrebbe essere l'intera gamma di azioni disponibili ad un agente? come queste azioni possono essere raggruppate per raggiungere l'obiettivo? in che situazioni un agente modificherà il proprio comportamento? il comportamento di un agente è influenzato dalle azioni del giocatore?*

- Il problema **strategico**: oltre a prendere decisioni autonome, le varie entità possono collaborare tra loro e stabilire una strategia che le porterà alla realizzazione di uno scopo (I.A. di gruppo). Domande tipiche possono essere: *gli agenti hanno necessità di lavorare insieme per raggiungere lo scopo? gli agenti hanno necessità di conoscere proprietà su larga scala del gioco, oppure devono limitarsi ai propri sensi e alla propria percezione? gli agenti possono pensare autonomamente anche quando devono mostrare comportamenti di gruppo?*

È importante sottolineare che in tale suddivisione, i primi due problemi (la parte di movimento e quella di decision making) contengono algoritmi che lavorano solo sul singolo agente, mentre l'ultima sezione, quella strategica, opera a livello più macroscopico e considera un'intero gruppo di agenti che collaborano per raggiungere uno scopo. Inoltre non è sempre detto che tutti i livelli siano necessari [16]: in alcuni casi, soprattutto nel caso dei board games , l'unico livello indispensabile è quello strategico (vedi per esempio gli scacchi o la dama, le singole pedine non hanno bisogno di algoritmi propri, e tutto è deciso a livello generale dal giocatore che le muove). In altri casi invece è proprio il livello strategico a mancare, in quanto i vari elementi mossi da un intelligenza artificiale non operano seguendo nessun tipo di strategia ma sono di tipo puramente reattivo. Queste tre tipologie di problemi vengono ora analizzate più nel dettaglio, e per ciascuna vengono presentate alcune pratiche comuni utilizzate nella loro risoluzione.

2.2.1 Il problema del movimento

Questo problema è probabilmente il più semplice da affrontare in quanto si riduce tipicamente al solo pathfinding e viene risolto utilizzando algoritmi oramai noti e ben consolidati (vedi Sezione 2.3.3). Tali algoritmi si dividono principalmente in due categorie [6]: quelli che si occupano del movimento indipendente di una singola entità, i quali comprendono algoritmi come A* e Dijkstra , e quelli invece di tipo cooperativo che cercano di trovare un percorso per unità multiple che si muovono in gruppo e che potrebbero ostacolarsi a vicenda, alcuni esempi possono essere evoluzioni del classico A*

[27, 31] (Cooperative A*, Hierarchical Cooperative A* e Windowed Hierarchical Cooperative A*) oppure tecniche più esotiche e provenienti dal settore della robotica come i campi potenziali [14] o ancora l'utilizzo di reservation tables spazio-temporali [6, 31] che memorizzano lo stato dell'ambiente di gioco anche per istanti di tempo successivi a quello corrente. Ovviamente più la complessità dell'ambiente di gioco aumenta, più l'algoritmo di pathfinding dovrà essere completo e rifinito. In alcuni casi si avranno ambienti dinamici in rapida evoluzione e sarà necessario utilizzare algoritmi di dynamic pathfinding (D*, D*-Lite) [30, 20]. Una delle caratteristiche (e delle limitazioni) principali dell'algoritmo A* è quella che per poter lavorare necessita di una certa discretizzazione dell'ambiente di gioco (divisione in caselle, definizione di waypoints), e ciò non sempre è possibile. Si prenda come esempio esempio quello di un veicolo che deve farsi strada a gran velocità tra il traffico. In questo caso sarà necessario utilizzare algoritmi di pathfinding a tempo continuo (Continuous Time Pathfinding) [16] nei quali i singoli nodi del grafo non rappresentano più posizioni all'interno del mondo di gioco, ma bensì stati di quest'ultimo. Un'altra direzione interessante nella quale è possibile muoversi e che potrebbe superare molte delle limitazioni imposte dai classici algoritmi di pathfinding è quella fornita dal campo del *machine learning*, in particolare tramite l'utilizzo di reti neurali, che porterebbe ad avere un modello generalizzato adatto a qualsiasi caso particolare si possa presentare poi in fase di gioco, anche quelli non previsti in fase di training [12, 11]. Un'alternativa più recente ai classici algoritmi di pathfinding è quella dell'utilizzo di mesh navigazionali (navmesh) [10] che sostituiscono la classica struttura a waypoints con un grafo connesso di poligoni *convessi* (Figura 2.2). Utilizzando una struttura simile ogni agente saprà sempre di potersi muovere liberamente all'interno del poligono in cui si trova proprio grazie alla sua convessità, mentre per spostarsi da un poligono all'altro è possibile utilizzare le tecniche classiche di pathfinding già citate.

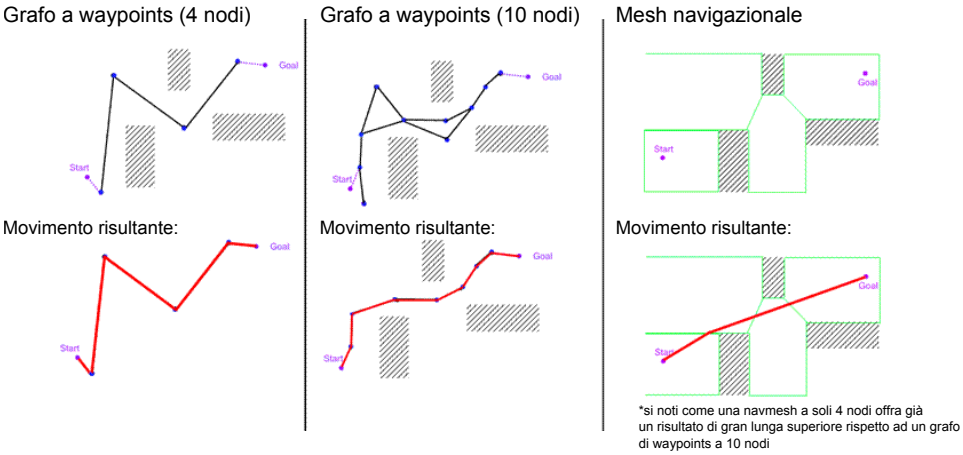


Figura 2.2: Waypoints e Mesh Navigazionali a confronto

2.2.2 Il problema decisionale

Quello decisionale è probabilmente il primo problema che viene alla mente quando si comincia ad implementare un motore di intelligenza artificiale e, a prima vista, si potrebbe pensare che esso ne costituisca la parte più corposa e complicata. In realtà, sebbene negli ultimi anni siano stiano vedendo la luce tecniche sempre più nuove e avanzate, la maggior parte dei prodotti commerciali utilizza ancora tecniche e modelli molto semplici come macchine a stati e alberi decisionali. Indipendentemente dalla tecnica utilizzata, il cuore di un sistema di decision making è sempre un semplice modello di input-output come quello mostrato in Figura 2.3:

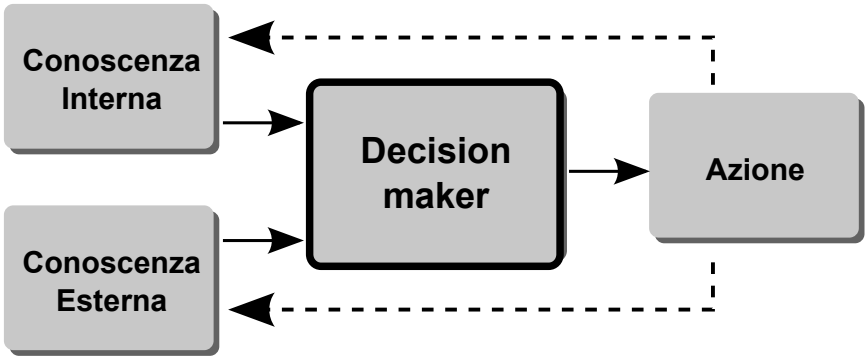


Figura 2.3: Modello generico di decision making

L'input è costituito da tutta la *conoscenza* posseduta dall'agente comandato dall'intelligenza artificiale che può essere suddivisa in **conoscenza interna** e **conoscenza esterna**. La conoscenza esterna racchiude tutto ciò che l'agente conosce a priori o può osservare riguardo l'ambiente che lo circonda,

mentre quella interna riguarda lo stato, le abilità e le conoscenze che essa possiede nonchè il suo obiettivo attuale e la storia passata di ciò che è stato fatto fino a quel momento. L'output, invece, è rappresentato da *azioni* che l'agente compirà in seguito alla decisione presa, che porteranno a dei cambiamenti dell'ambiente esterno, dello stato interno, o di entrambi.

Oltre alle già citate macchine a stati finiti e agli alberi decisionali, altre tecniche tipicamente utilizzate e che sono anche oggetto di interesse in campo accademico solo quelle dei sistemi esperti basati su regole [1] e quella delle reti neurali [24, 29] che vedono la loro prima applicazione al settore dei videogiochi oramai quasi 20 anni fa con il gioco Creatures [Millennium Interactive, 1996] un ambizioso simulatore di vita artificiale nel quale per ogni creatura era implementato un vero e proprio modello computazionale ispirato al cervello umano, basato su lobi e neuroni, e per nel quale lo scambio di informazioni era addirittura gestito tramite un modello biochimico semplificato [13]. Ovviamente tale modello voleva essere proprio il punto di forza dell'intero prodotto, e generalmente non è mai necessario entrare in un simile livello di dettaglio quando si vuole sviluppare un'intelligenza artificiale per un videogioco, anzi, in molti casi ad un modello complesso corrisponde un gameplay povero e noioso. Riuscire a capire quando è necessario aggiungere complessità e quando invece è preferibile mantenere un modello il più semplice possibile è una delle sfide più difficili da affrontare quando si vuole disegnare un motore di intelligenza artificiale, e spesso i migliori programmatori sono proprio quelli che riescono a usare tecniche molto semplici che danno invece l'illusione di grande complessità. Molto meno complessi nella struttura rispetto alle reti neurali, i sistemi esperti offrono il grande vantaggio di essere sempre consistenti e quindi l'agente risponderà sempre correttamente (ma probabilmente seguendo lo stesso pattern) agli stimoli che riceve dall'esterno, mentre il loro svantaggio principale consiste appunto in una certa predicibilità delle mosse da parte del giocatore, e in alcuni casi in comportamenti da parte dell'I.A. che mancano di un certo senso comune, e che portano quindi alla rottura di quell'illusione di intelligenza di cui si è parlato nella Sezione 2.1. Le reti neurali d'altro canto, offrono invece il vantaggio di non basarsi su regole e script prestabiliti e offrono quindi una maggiore variabilità delle mosse e dei comportamenti degli agenti. In questi casi il processo di training avviene in modo automatico ed è possibile principalmente optare per due soluzioni: nella prima il training avviene *offline*, ed una volta ottenuto il risultato desiderato si "congela" l'intelligenza artificiale, che da quel momento agirà quindi come desiderato senza ulteriori

modifiche. Ciò offre maggiore stabilità al gioco, ma impedisce alla I.A. di adattarsi in risposta alle tattiche di gioco dei giocatori. Dal lato opposto invece si può pensare di utilizzare un training di tipo *online*, nel quale l'intelligenza artificiale continua ad apprendere ed evolversi anche durante il gameplay [29]. Un'ultima tecnica di decision making che vale la pena menzionare poichè sempre più utilizzata negli ultimi anni è quella degli alberi comportamentali (Behavior Trees) che ha visto le sue prime applicazioni in campo videoludico in prodotti commerciali come Halo [17, 9] [Bungie Studios, 2001] e Spore [15] [Maxis, 2008] e di cui un esempio è mostrato in Figura 2.4:

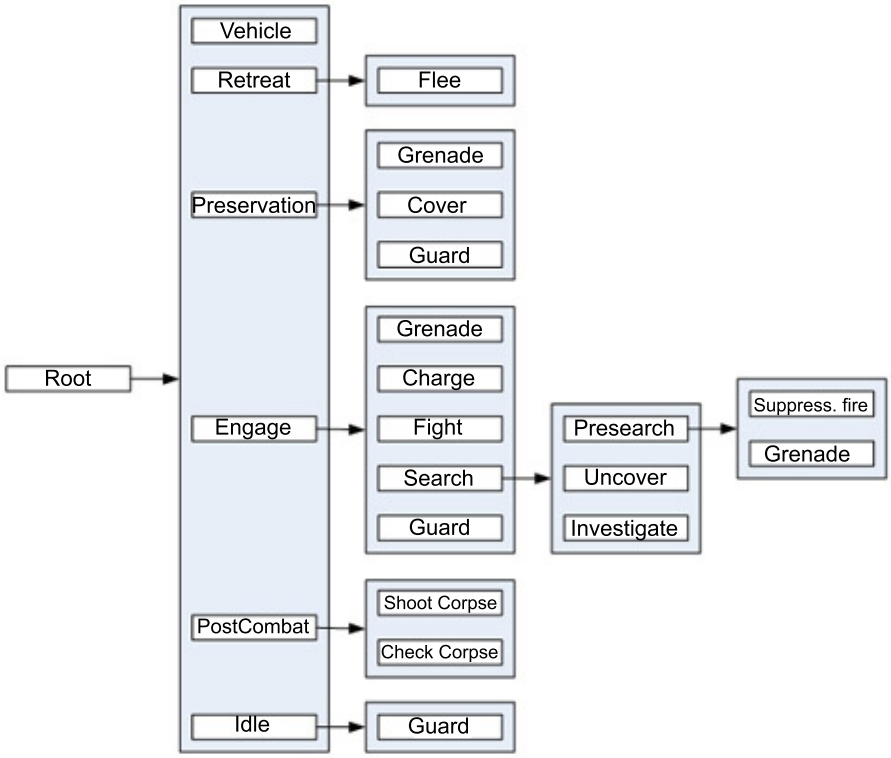


Figura 2.4: Esempio di Behavior Tree

Gli alberi comportamentali sono un utile strumento di design riconducibile sotto molti aspetti alle macchine a stati finiti gerarchiche, ma tuttavia molto più versatili, e verranno descritti più nel dettaglio nella Sezione 2.3.3. Per concludere, esistono poi tutta una serie di situazioni particolari per le quali il processo decisionale può essere risolto in modo non equivoco tramite tecniche e modelli tipici della teoria dei giochi[1], e che verranno discusse più

[1]La teoria dei giochi è una scienza matematica che studia le situazioni di conflitto

in dettaglio nella Sezione 2.4.

2.2.3 Il problema strategico

Sebbene l'implementazione del livello di movimento e di quello decisionale sia spesso sufficiente a costruire un buon motore di intelligenza artificiale, in alcuni casi è necessario spingere oltre la complessità ed aggiungere un ulteriore livello, quello strategico. Il livello strategico include tutti quegli algoritmi che operano su interi gruppi di agenti al fine di coordinarne le azioni, nonchè una serie di tecniche finalizzate all'analisi macroscopica della situazione di gioco. Molte delle tecniche presentate per risolvere i problemi decisionali possono essere estese anche al livello strategico (macchine a stati, alberi decisionali), mentre le tecniche proprie di questo livello si basano generalmente sull'*analisi tattica dall'ambiente di gioco*, che può essere principalmente di due tipi:

- un'analisi **statica**, o a priori, fatta in fase di level design. Un esempio tipico è quello dell'analisi tattica basata su waypoints (Figura 2.5) che consiste nel definire dei punti chiave nell'ambiente di gioco che possono essere usati per prendere decisioni in fase di calcolo delle strategie. Caso molto comune è quello di definire punti di copertura dove gli agenti possono trovare riparo o nascondiglio, punti di imboscata, punti di ritirata e punti di vittoria. Utilizzando correttamente queste informazioni, è possibile definire un algoritmo strategico in grado di coordinare contemporaneamente le mosse di più agenti al fine, per esempio, di tendere un'imboscata al giocatore o di raggiungere un obiettivo condiviso che porta alla vittoria della squadra.

- un'analisi **dinamica** basata sull'analisi in tempo reale di alcune proprietà della mappa di gioco. Analisi di questo tipo vengono generalmente fatte tramite le cosiddette *mappe di influenza* (Figura 2.6) che possono essere calcolate matematicamente tramite semplici funzioni di distanza del tipo

  $$I_d = \frac{I_0}{\sqrt{1+d}} \qquad I_d = \frac{I_0}{(1+d)^2}$$

  con I_d influenza calcolata alla distanza d e I_0 valore di influenza iniziale nel punto di che "genera" tale influenza, oppure utilizzando tecniche

ricercandone soluzioni competitive e cooperative tramite modelli ben precisi. Si occupa quindi dell'analisi delle decisioni individuali in situazioni di interazione con altri soggetti rivali, finalizzate in genere al massimo guadagno del soggetto.

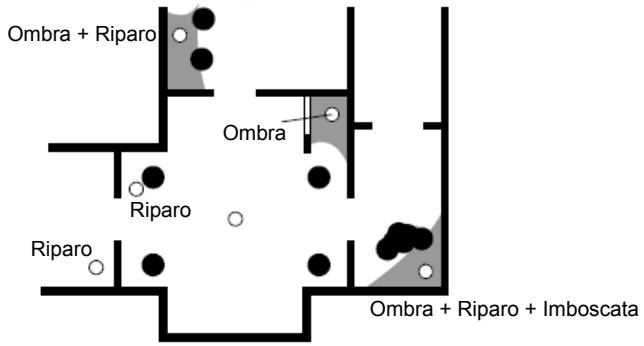


Figura 2.5: Analisi strategica basata su waypoints

come quella del map flooding che verrà descritta nella Sezione 2.3.3. Questo tipo di analisi si presta molto bene per giochi di tipo strategico, sia a turni che in tempo reale e, vista la loro struttura regolare del piano di gioco, ai board games.

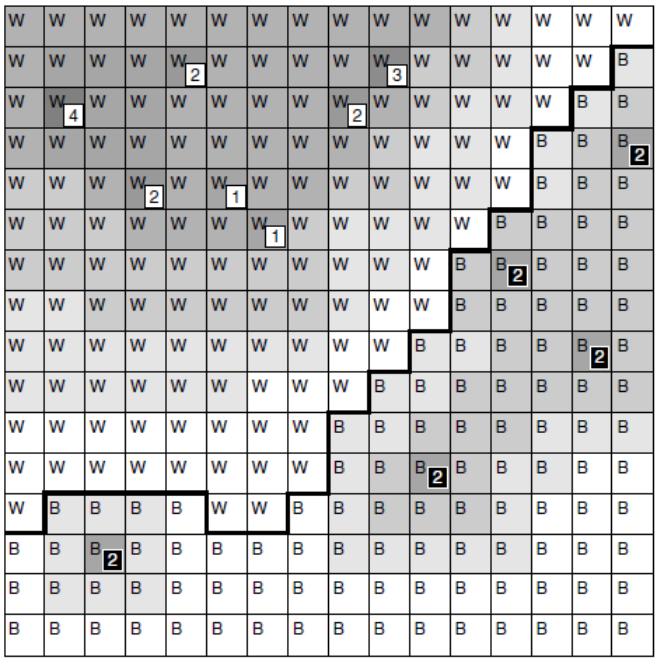


Figura 2.6: Esempio di mappa di influenza

2.3 Tecniche e strumenti

Nella Sezione 2.2 sono stati presentati i tre problemi principali da affrontare quando si desidera implementare un motore di intelligenza artificiale per giochi: il problema del movimento, il problema decisionale ed il problema strategico. Vediamo ora invece quali sono le tecniche e gli strumenti principali utilizzati nella risoluzione di tali problemi. Gli strumenti a disposizione possono essere principalmente suddivisi in tre categorie, e possono essere utilizzati a piacimento per affrontare tutte e tre le tipologie di problemi visti, essi sono:

- l'utilizzo di espedienti, workaround e soluzioni ad hoc non riconducibili ad alcuna tecnica "ufficiale" del campo dell'intelligenza artificiale. Spesso definite come dei "trucchi" utilizzati dal programmatore per aggirare il problema, queste tecniche sono invece perfettamente lecite da usare fin tanto che esse risultino trasparenti agli occhi del giocatore, e vengono infatti adottate in quasi la totalità dei prodotti commerciali.

- l'utilizzo di euristiche , ovvero di metodi empirici che in genere forniscono una soluzione accettabile al problema, ma che non danno alcuna garanzia di fornire la soluzione ottima. L'esempio più banale di euristica è quella per tentativi (trial and error): si prova a cercare per un determinato numero di volte la soluzione fin tanto che non non la si trova o che vengono esauriti i tentativi. In genere un'euristica viene utilizzata quando si hanno vincoli in termini di tempo di elaborazione o di memoria: mentre una ricerca esaustiva della soluzione (ottima) richiederebbe una grande quantità di risorse, l'utilizzo di un euristica è in grado di trovare una soluzione (sub-ottima, ma comunque accettabile) molto più rapidamente.

- l'utilizzo di algoritmi, ovvero l'esatto opposto delle euristiche: dei procedimenti *formali* che risolvono il problema in un numero definito di passi. Il grande vantaggio nell'uso di un algoritmo sta appunto nella sua formalità: sono sempre ben definiti quali sono gli input, quali sono gli output ed attraverso quali passi si arriva alla soluzione. Un'altro vantaggio risiede nel fatto che, mentre soluzioni ad hoc ed euristiche vengono implementate per risolvere un problema specifico, gli algoritmi non fanno di solito riferimento ad alcun caso particolare, e possono essere riutilizzati più volte in diversi prodotti, costituendo così il nocciolo fondamentale di un motore generico per l'intelligenza artificiale.

2.3.1 Espedienti e soluzioni ad-hoc

In molti casi le tecniche utilizzate nello sviluppo di una I.A. da gioco non sono riconducibili ad alcuna tecnica "ufficiale" usata nel campo delle intelligenze artificiali. In Pac-Man [Midway Games West, Inc., 1979], che è probabilmente il primo gioco che vanta di un'"intelligenza artificiale" vera e propria, i fantasmi nemici che si trovavano nella condizione di dover fuggire dal giocatore decidevano in quale direzione svoltare ad ogni incrocio tramite il risultato un semplice generatore di numeri casuali [16, 23], e generare un numero casuale non è di certo una tecnica di intelligenza artificiale. Nonostante questo il gioco risultava comunque divertente, e i giocatori percepivano una certa "intelligenza" da parte dei nemici. Un'altro "trucco" che porta ad ottenere ottimi risultati sia in termini di gameplay, sia di coinvolgimento del giocatore senza la necessità di sviluppare elaborati algoritmi di I.A. è quello di considerare con quali meccanismi il cervello umano percepisce l'intelligenza da parte di un altra entità, al fine di indurlo a credere di avere a che fare proprio con un essere dotato di intelletto.

Gran parte della nostra valutazione riguardo l'intelligenza di un altro individuo si basa sul semplice concetto di *comportamento*: siamo in grado di affermare se un altro individuo è intelligente semplicemente osservando come si comporta. Se un giocatore non è in grado di capire perchè l'I.A. ha preso una decisione o si sta comportando in un certo modo, essa diventa immediatamente meno furba ai suoi occhi, e farà molta più fatica a comprenderne gli obiettivi e a prevederla [5]. Una buona I.A. dovrebbe sempre rendere in modo più o meno evidente in quale stato si trovi e quali sono le sue intenzioni fornendo dei feedback al giocatore. Deve essere quindi *intelligibile* , ovvero riconoscibile e comprensibile tramite l'intelletto, poichè anche l'intelligenza artificiale più complessa e completa di questo mondo diventa inutile nel momento in cui il giocatore non è in grado di percepire a cosa stia pensando [5]. Uno dei metodi più semplici ma molto efficaci per rendere l'intelligenza artificiale intelligibile è quello dell'utilizzo di animazioni, che anche in questo caso non hanno nulla a che fare con algoritmi e tecniche del mondo dell'intelligenza artificiale. Nel gioco The Sims [Maxis, 2000], per esempio, gran parte delle decisioni e dei comportamenti vengono comunicati al giocatore tramite semplici animazioni dei personaggi. In altri casi un animazione può rendere immediatamente chiare le intenzioni e lo stato d'animo di un nemico: ha paura? vuole attaccare? Un'altra tecnica è quella di fornire dei feedback sonori al giocatore, nel gioco Halo [Bungie Software, 2004] per esempio i nemici urlavano una certa frase ogni volta che volevano lanciare

una granata contro il giocatore. Per quanto questo comportamento possa sembrare alquanto stupido (quale soldato avvertirebbe il nemico del pericolo?) il giocatore proverà di certo più piacere a trovarsi di fronte una I.A. di questo tipo, piuttosto che subire ogni volta la frustrazione di veder persa la partita per colpa di una granata lanciata in modo "intelligente" ma silenzioso. Infine, un'ulteriore tecnica che richiede forse un paio di righe di codice, ma di grande effetto nel dare al giocatore l'illusione di intelligenza è quella dell'utilizzo delle espressioni facciali. Anche in questo caso non abbiamo a che fare con alcuna tecnica del mondo delle intelligenze artificiali, tuttavia l'esecuzione di una giusta espressione facciale nel momento giusto darà immediatamente al giocatore l'impressione di avere a che fare con un entità intelligente. Molto spesso, quindi, non c'è bisogno di complicati modelli cognitivi, reti neurali o algoritmi genetici per dare l'impressione di intelligenza.

Altri espedienti si possono invece usare per quanto riguarda il movimento. Il gioco Rome Total War [Activision, 2004] (Figura 2.7) per esempio, vedeva la partecipazione simultanea di centinaia di unità che si muovevano in formazione. Anzichè usare complicati algoritmi di pathfinding cooperativo gli sviluppatori hanno deciso invece di trattare l'intera formazione come una singola entità, ed assegnare a ciascuna unità che la componesse un valore di offset dal centro di essa. Per ogni formazione veniva poi calcolata la larghezza e, nel caso in cui essa fosse troppo grande per passare in determinati punti, anche questa volta anzichè optare per complicate soluzioni che prevedessero la ricerca di un percorso alternativo per ogni singola unità si è scelto semplicemente di ridimensionare la formazione (facendola diventare eventualmente una linea) fino a che questa non fosse sufficientemente stretta da attraversare l'ostacolo [4].

2.3.2 Euristiche

Un'euristica è una regola generica o una soluzione approssimata che può adattarsi bene in molti casi, ma che non assicura di essere corretta per la totalità di essi. Spesso un'euristica viene incorporata all'interno di un algoritmo come supporto all'elaborazione per arrivare alla soluzione in tempi più brevi, mentre in altri casi vengono implementate delle euristiche ad-hoc per risolvere svariati problemi di I.A. Per esempio si potrebbe pensare di "guidare" un'algoritmo di pathfinding di un personaggio non giocante utilizzando un euristica del tipo *"muoviti verso la posizione del giocatore"*. La soluzione così trovata, benchè abbia indubbiamente un senso logico (muoversi verso il giocatore pare di certo una scelta sensata), potrebbe differire dalla soluzione

Figura 2.7: Muovere centinaia di unità non significa necessariamente utilizzare centinaia di algoritmi di pathfinding

ottimale che verrebbe trovata invece da un algoritmo di ricerca del percorso "puro" che valuta invece la totalità dei percorsi possibili. Il cervello umano fa un uso continuo di questo tipo di euristiche, che guidano il nostro processo decisionale facendoci inconsciamente distinguere tra "buone" e "cattive" mosse e facendoci arrivare rapidamente ad una decisione anche in situazioni a noi non note, ed è quindi evidente che l'utilizzo di euristiche diventa fondamentale anche nel campo dell'intelligenza artificiale, dove è richiesto che le decisioni vengano prese in una frazione di secondo e dove una ricerca esaustiva e non guidata della soluzione sarebbe inattuabile. Un calcolatore elettronico non è infatti in grado di capire in modo autonomo se una mossa è inutile o se un' informazione è irrilevante, e l'unico modo che ha di farlo è quello di testare e dimostrare algoritmicamente questo fatto esplorando tutte le strade possibili. Ciò condannerebbe inevitabilmente l'elaboratore a sprecare gran parte del proprio tempo a considerare strade e soluzioni che non porterebbero a nessuna conclusione [7], e dimostra quindi quanto importante sia l'utilizzo di euristiche in questo campo. Di seguito vengono presentate alcune tra le euristiche più comuni, che possono essere applicate ad una vasta gamma di problemi.

Variabile più vincolata (Most Constrained Variable)

Il problema che si pone è il seguente: dato lo stato corrente del mondo e un insieme di oggetti o di azioni solamente una di questa deve essere scelta, quale scelgo? Come regola generale (quindi un euristica) andrebbe sempre scelto l'oggetto o l'azione che diventa opzionale per il minor numero di casi. In altre parole bisognerebbe sempre dare priorità a ciò che è raro e che è vincolato da più fattori (se ho l'occasione di farlo, lo faccio) e lasciare in secondo piano ciò che è più probabile che possa avvenire ancora in un secondo momento.

Affrontare prima i problemi più difficili

Tale regola si applica soprattutto nei casi in cui l'esecuzione di un'azione a monte ha ripercussioni su tutta una serie di altri fattori a valle. In genere più una regola è complessa, più conseguenze essa porta al mondo di gioco ed è quindi consigliabile considerarla per prima. Dare priorità agli aspetti semplici, infatti, porterebbe ad avere dei risultati che probabilmente verrebbero resi obsoleti dall'applicazione di un'azione di più alto livello. Si prenda come esempio quello in Figura 2.8: si hanno 4 unità di livello 1 (fanteria) e un unità di livello 2 (carro armato) e le si vogliono dividere in due squadre con il vincolo che esse siano equilibrate e quindi il valore totale sia lo stesso. È evidente che il problema principale in questo caso è posto dal carro armato, che è l'unità più difficile da assegnare. Iniziare ad affrontare il problema partendo dalle unità di livello 1 porterebbe sempre ad una soluzione errata, mentre l'utilizzo di un euristica sensata porta ad una corretta distribuzione dei punteggi.

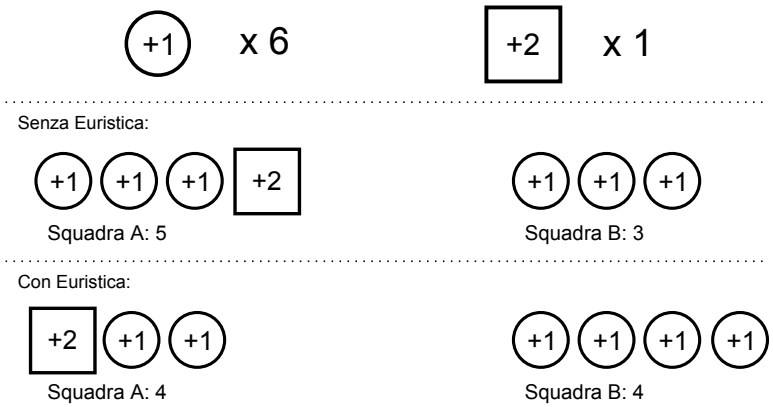


Figura 2.8: Esempio di applicazione dell'euristica

Questo esempio, se pur molto semplice, dimostra l'importanza e la potenza dell'utilizzo di euristiche nell'affrontare problemi decisionali.

Provare prima le strade più promettenti

Questo tipo di euristica si basa su un meccanismo molto semplice, ovvero quello di assegnare un *punteggio* ad ogni possibile strada da percorrere. Anche nel caso in cui tali punteggi siano altamente inaccurati, provare le soluzioni in un'ordine dettato da tale parametro porterà generalmente a soluzioni più rapide che tentare invece di affrontare il problema in maniera totalmente casuale.

2.3.3 Algoritmi e modelli

A differenza di un'euristica, un'algoritmo è un procedimento *formale* che deve rispettare i seguenti requisiti (Knuth 1968, 1973):

- deve essere **finito**, ovvero deve sempre terminare in un numero finito di passi.
- ogni passo deve essere ben **definito** e non ci devono essere ambiguità.
- deve essere ben chiaro quali sono gli **input** e quali gli **output**.
- deve essere **efficacie**: ovvero tutte le operazioni svolte dall'algoritmo dovrebbero essere sufficientemente semplici da poter essere in principio eseguite in un tempo finito anche da un essere umano munito di carta e penna.

Dalla definizione di Knuth si nota come l'utilizzo esclusivo di algoritmi per realizzare un motore di I.A. possa portare ad una soluzione troppo rigida, che poco si adatta all'applicazione finale in ambito videoludico, d'altro canto l'utilizzo delle sole euristiche e di soluzioni ad-hoc porterebbe inevitabilmente alla continua "reinvenzione della ruota" poichè per alcuni problemi esistono già soluzioni note e ben consolidate. Vengono quindi ora presentati alcuni dei principali algoritmi utilizzati per risolvere problemi tipici di una intelligenza artificiale per giochi.

Algoritmo A*

L'algoritmo A* (Hart, Nilsson, Raphael, 1968) è forse il più utilizzato dall'industria videoludica, tanto che si stima che più del 90% dei prodotti commerciali ne fa uso ancora oggi [8]. Per una definizione formale dell'algoritmo

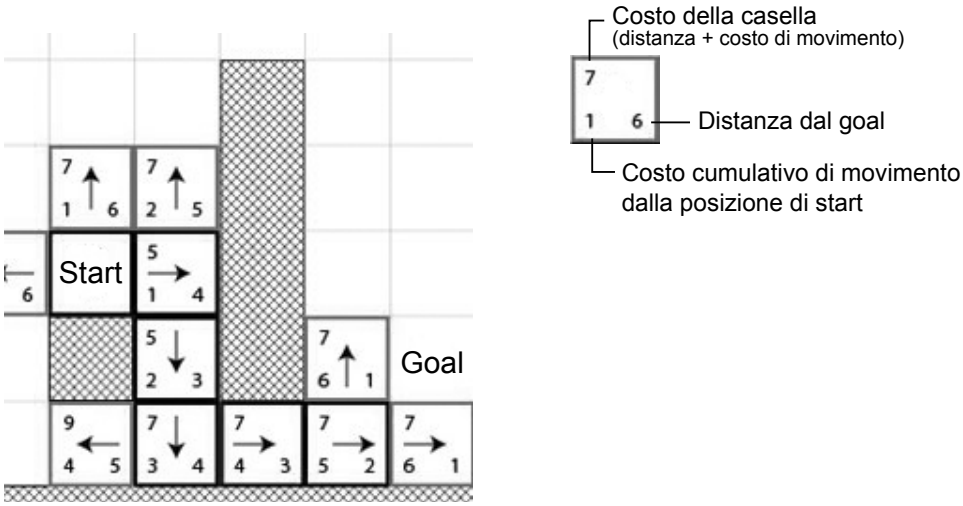


*Figura 2.9: Pathfinding con algoritmo A**

si rimanda il lettore all'Appendice A. Nel campo dei videogiochi questo algoritmo viene usato soprattutto per risolvere problemi di pathfinding in quanto è in grado di trovare rapidamente un percorso ottimale da un nodo di partenza ad un nodo goal finale. Come già accennato nella Sezione 2.2.1 per utilizzare questo algoritmo è quindi necessario discretizzare l'ambiente di gioco suddividendolo in nodi connessi tra loro, modellandolo quindi in pratica come un *grafo*. Spesso per semplificare le cose è consigliabile suddividere l'area di gioco in una vera e propria scacchiera (Figura 2.9), assegnando alle caselle che contengono un ostacolo un costo di attraversamento infinito. Come si può vedere dalla figura, l'algoritmo A* è un tipico esempio di algoritmo che trae beneficio dall'utilizzo di un euristica: anzichè cercare tutte le possibili strade che portano alla soluzione, si sceglie di dare priorità a tutte quelle che effettivamente ci stanno facendo avvicinare alla meta: l'euristica in questione è quindi semplicemente un semplice calcolo della distanza dalla posizione corrente a quella di goal.

Esistono molte varianti dell'algoritmo A*, le più importanti utilizzate in ambito videoludico sono:

- **Local Repair A* (LRA*)**: È stato per molto tempo lo standard utilizzato dall'industria [27]. Ogni agente calcola il proprio percorso in modo autonomo utilizzando una versione standard di A* ed ignorando tutti gli altri agenti presenti nell'area di gioco tranne quelli immediatamente adiacenti. Una volta trovato il percorso, l'agente comincia a seguirlo fino a che non viene rilevata una collisione imminente e, quando ciò avviene, ovvero quando l'agente sta per entrare in una posizione

già occupata, l'algoritmo viene eseguito nuovamente per calcolare il percorso rimanente. Uno dei problemi principali di questo algoritmo è la possibilità della nascita di cicli e la creazione di colli di bottiglia in situazioni affollate nelle quali troppi agenti cercano di evitarsi l'un l'altro, portando alla ricomputazione multipla di un nuovo percorso, che viene ripetuta quasi per ogni frame con un notevole impatto sulle prestazioni.

- **Cooperative A* (CA*)**: Questo algoritmo cerca di risolvere i problemi legati all'algoritmo LRA* semplicemente non più ignorando gli altri agenti, ma tenendo in considerazione anche le mosse da loro pianificate. Una nuova mossa di *wait* viene introdotta tra le possibili scelte selezionabili dall'agente, e lo stato spazio-temporale di ciascun nodo del grafo utilizzato per il pathfinding viene memorizzato in un'apposita tabella detta *reservation table* . Grazie alla reservation table, ciascun agente saprà in anticipo se una determinata posizione verrà attraversata da qualcun'altro in un *determinato istante di tempo*, e l'algoritmo potrà così trattare tale posizione come se fosse un ostacolo vero e proprio.

- **Hierarchical A* (HA*)**: È la versione gerarchica del normale algoritmo A*, non più un unico livello di pathfinding, ma più livelli a seconda della granularità richiesta dalla situazione (Figura 2.10). Si prenda per esempio un vasto ambiente di gioco costituito da più città, ciascuna con la propria planimetria e nelle quali esiste anche una geometria interna per alcuni degli edifici che ospita. Supponendo che l'intera struttura di gioco risieda completamente in memoria nello stesso tempo, non ha assolutamente alcun senso utilizzare l'intero grafo dei waypoint a livello più basso (quello che rappresenta l'interno degli edifici) per muovere, ad esempio, un agente che si deve spostare da una città ad un'altra. I diversi livelli di profondità dell'algoritmo si possono creare in maniera automatica tramite tecniche di clustering sui nodi, oppure manualmente definendoli a livello di codice in fase di level design.

Algoritmo di Dijkstra

A differenza dell'algoritmo A*, quello di Dijkstra non utilizza nessuna euristica ed esplora tutti i possibili cammini in un grafo senza distinguere una mossa "utile" da una "inutile". Dato un nodo di partenza, l'algoritmo non restituirà un singolo cammino verso un nodo obiettivo come accadeva con

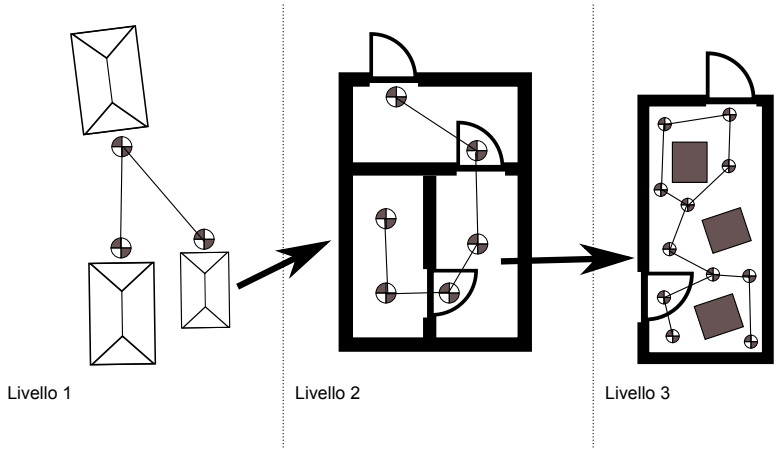


Figura 2.10: A Gerarchico*

A*, ma bensì una struttura contenente il costo per raggiungere *tutti gli altri nodi del grafo*. Ovviamente sarà possibile ricostruire il percorso desiderato anche in questo caso, tuttavia può essere uno spreco calcolare tutto un insieme di rotte che verranno poi scartate perchè di non interesse. Per questi motivi l'algoritmo di Dijkstra trova poche applicazioni in ambito di pathfinding a livello commerciale, tuttavia può però rivelarsi uno strumento molto utile per svolgere analisi tattiche e strategiche utilizzandolo all'interno di una tecnica che prende il nome di *map flooding* [16]. La tecnica del map flooding può essere facilmente compresa osservando la Figura 2.11: dati due nodi che esercitano una certa influenza sul resto della mappa (ad esempio due città nemiche che determinano i confini di ciascun territorio), quali proprietà devono avere tutti i nodi intermedi? Il map flooding risponde proprio a questa domanda: serve in pratica a "propagare" un determinato stato o una determinata proprietà di un nodo verso tutti gli altri sulla base di un valore di distanza, e tale distanza viene agevolmente calcolata tramite l'algoritmo di Dijkstra.
Per un listato formale dell'algoritmo di Dijkstra si faccia riferimento all'Appendice A.

Macchine a stati finiti

Le macchine a stati finiti sono un modello o un *design pattern* i cui concetti base vengono riutilizzati molto spesso nel campo della I.A. per giochi. Benchè siano utilizzate già oramai da più di 30 anni in questo campo (il gioco Pac-Man ne faceva già uso nel 1979), sono comunque degne di nota sia perchè in molti casi sono ancora largamente utilizzate tutt'oggi [16], sia

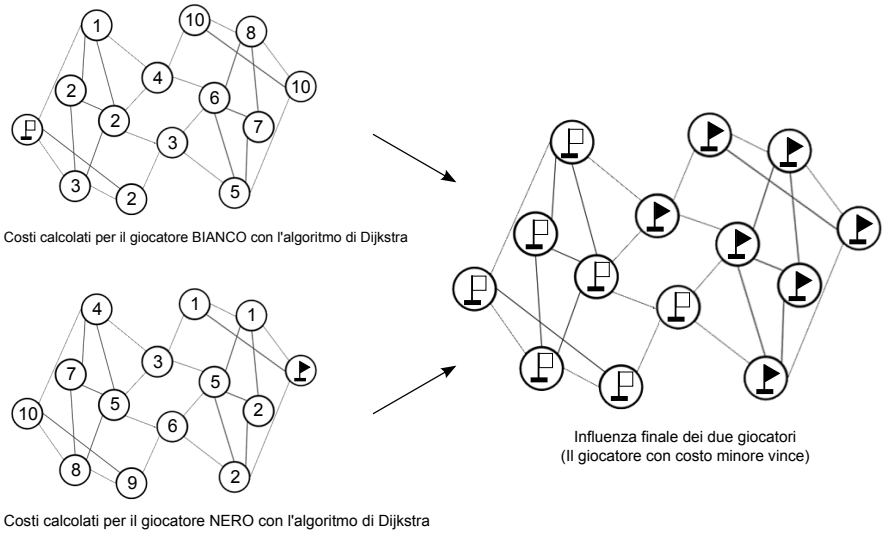


Figura 2.11: Algoritmo di Dijkstra nella tecnica di map flooding

perchè il concetto di *stato* è un elemento chiave nel campo dell'intelligenza artificiale. Una macchina a stati finiti (Figura 2.12) è una collezione di *stati* collegati tra loro tramite delle *transizioni*. All'avvenire di un determinato evento, o quando certe condizioni vengono soddisfatte, l'agente comandato dalla macchina passerà da uno stato ad un altro.

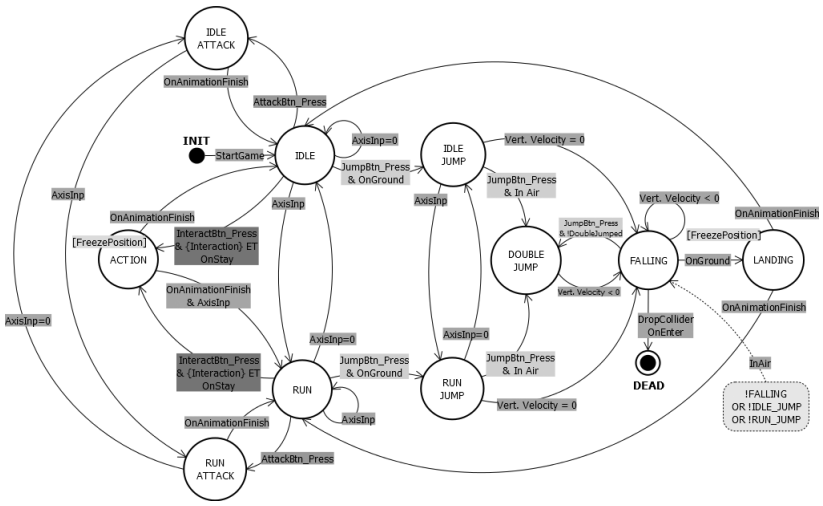


Figura 2.12: Esempio di Macchina a Stati

Pare immediato pensare che un modello simile ben si adatti a rappresentare il comportamento di un personaggio all'interno di un ambiente di gioco, e

infatti, per modelli semplici, così è sempre stato, tant'è che spesso i giocatori hanno l'illusione di assistere a complesse manovre coordinate da parte della I.A. [16] quando in realtà tutto ciò che sta alla base è una semplice macchina a due stati [23]. Si potrebbe addirittura pensare che un tale modello possa risolvere la quasi totalità dei problemi posti in questo ambito, tuttavia le macchine a stati finiti sono un modello ormai di gran lunga superato e che presentano molte limitazioni, e la loro fragilità è stata oramai stata appurata [3, 28]. È per questo motivo che negli ultimi anni l'industria si sta muovendo verso modelli più elaborati come alberi comportamentali (Behavior Trees) e sistemi di pianificazione (Planning Systems).

Behavior Trees

Gli alberi comportamentali sono un'altro modello che ha cominciato ad essere utilizzato in ambito videoludico nei primi anni 2000 grazie a titoli come Halo 2 [Bungie Software, 2004] e Spore [Maxis, 2008]. Si tratta di un modello che sintetizza diversi aspetti di I.A. tra i quali le macchine a stati finiti gerarchiche, lo scheduling, il planning e l'esecuzione in sequenza di azioni. Mentre le macchine a stati costituiscono un modello formale e ben definito, per gli alberi comportamentali non esistono ancora regole ben precise per la loro costruzione. Spesso un behaviour tree assomiglia di più ad una macchina a stati gerarchica tipo quella mostrata in Figura 2.4, in altri casi invece si possono utilizzare dei formalismi più elaborati come quelli che vengono descritti in seguito. Principalmente, in un behaviour tree, il concetto di stato viene sostituito da quello di *task composito* , ovvero un'azione che può essere suddivisa in diverse sotto-azioni e che può avere come esito un successo oppure un fallimento. Questi task compositi si possono dividere principalmente nei due tipi mostrati in Figura 2.13 e sono:

- **Task di selezione**: un task di selezione prova ad eseguire tutte le azioni presenti nei nodi figlio fino a che una di esse restituisce esito positivo. Se un azione fallisce viene provata quella successiva, se tutte falliscono il nodo di selezione restituirà a sua volta un fallimento.

- **Task di sequenza**: in un task di sequenza, invece, le azioni vengono eseguite una dopo l'altra in successione. Non appena una di esse restituisce un fallimento, la sequenza termina e il nodo padre restituisce a sua volta un fallimento.

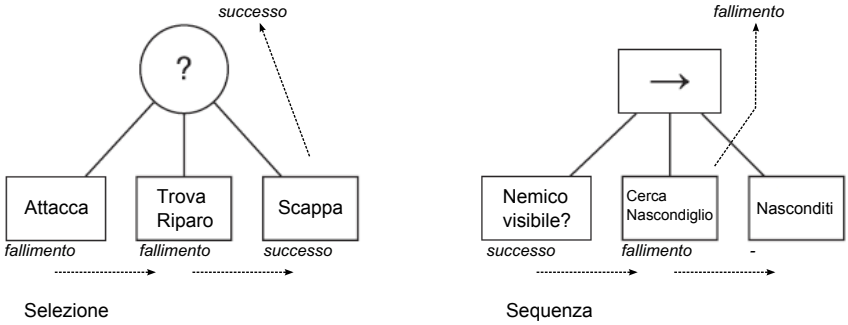


Figura 2.13: Task di selezione e sequenza in un Behavior Tree

Ovviamente questi elementi possono essere annidati tra di loro e composti per formare degli alberi complessi come quello mostrato in Figura 2.14:

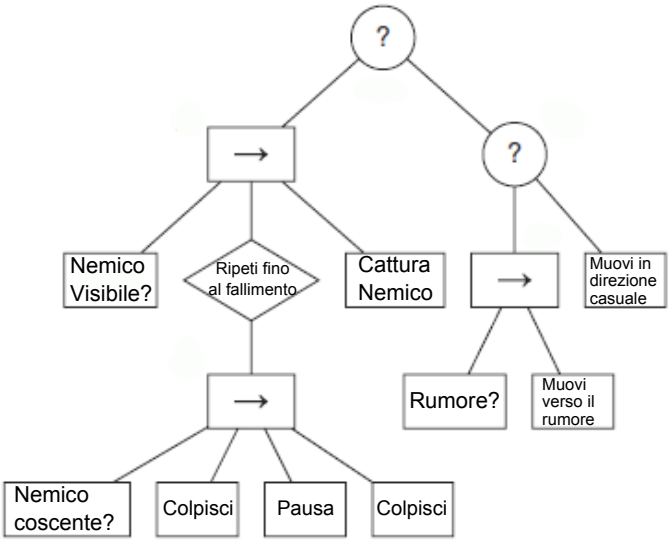


Figura 2.14: Esempio di albero comportamentale complesso che utilizza i task di selezione e sequenza

Ulteriori algoritmi e modelli

Oltre agli algoritmi e ai modelli presentati, esistono ovviamente tutta una serie di altri strumenti derivanti da diversi campi di ricerca e che vengono utilizzati regolarmente nel settore dell'intelligenza artificiale.
Tali strumenti comprendono:

- Logica Fuzzy
- Algoritmi Genetici
- Reinforcement Learning
- Steering Algotithm
- Algoritmo D*
- Algoritmo Windowed A*
- Constraint Satisfaction Problem
- Decision Trees
- Planning Systems
- Scheduling
- Processi Markoviani
- Sistemi Esperti
- Machine Learning
- Linguaggi di Scripting
- Ontologie

2.4 I.A. specifica per board games

Quando si parla di board games (giochi da tavolo) in ambito videoludico, andrebbero distinti principalmente tre casi:

1. Implementazione di board games "classici" (quali ad esempio scacchi, dama, othello, tris...) che rientrano quindi spesso in categorie di giochi ben definite come quelle dei *giochi finiti*, *giochi ad informazione perfetta* o *giochi a somma zero* [2] e per i quali la risoluzione può essere affrontata tramite modelli matematici e algoritmi della teoria dei giochi. Alcuni di questi giochi possono essere rappresentati graficamente tramite alberi (Figura 2.15) e possono essere risolti completamente con il solo utilizzo di tecniche matematiche ed euristiche (ad esempio il gioco della dama, nonostante le sue 5×10^{20} possibili mosse, è stato completamente risolto nel 2007, dopo quasi 18 anni di ricerca [26]) mentre per altri, ad esempio gli scacchi, sebbene teoricamente possibile, è impensabile poterli risolvere in maniera esaustiva. Tecniche comuni per lo studio di questi giochi comprendono quella dell'induzione a ritroso (Backward Induction), il metodo *minimax* e la tecnica della potatura alfa-beta (Alpha-Beta Pruning).

2. Giochi strategici a turni che presentano complessità molto maggiori rispetto ai giochi da tavolo classici ma per i quali l'assenza della necessità di un'elaborazione in tempo reale permette l'applicazione di

[2] Per giochi ad informazione perfetta si intendono quei giochi nei quali ogni giocatore conosce in ogni istante l'intero stato del gioco, mentre i giochi a somma zero sono quelli nei quali ciò che viene vinto (in termini di punteggio) da un giocatore è esattamente ciò che viene perso da un'altro giocatore [22].

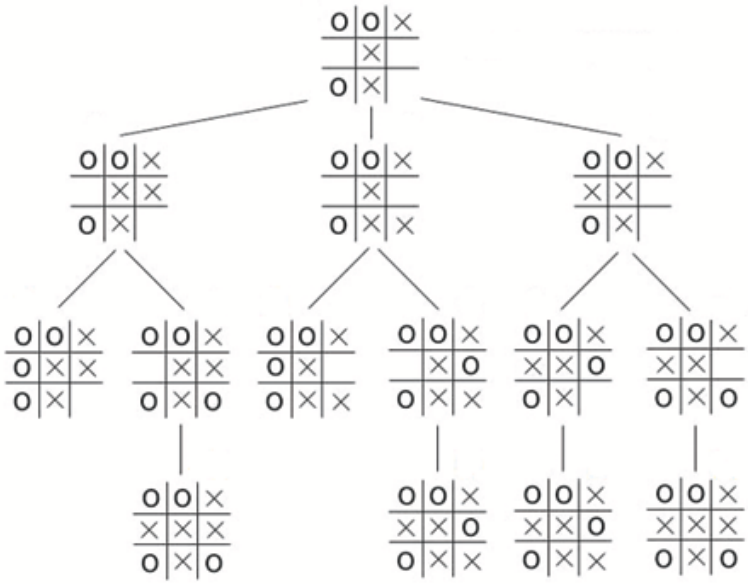


Figura 2.15: Porzione dell'albero del gioco del tris

complesse procedure di reasoning e di ricerca delle strategie spesso esaustiva, e in cui la struttura alla base del gioco è spesso riconducibile a quella dei board games (scacchiere, mappe esagonali). Esempi tipici di questi giochi sono la serie Civilization [Microprose, Fireaxis games, 1991-2013], Heroes of Might and Magic [New World Computing, 1995-2011] e i classici wargames basati su mappa esagonale. Giochi come questi introducono spesso elementi non deterministici (anche il semplice lancio di un dado per esempio introduce non determinismo) e sono ad informazione imperfetta, ovvero i giocatori non sono sempre a conoscenza dell'intero stato del gioco, rendendo quindi più difficile l'applicazione di tecniche standard di teoria dei giochi. Per risolvere i problemi di I.A. legati a questi giochi, vengono utilizzati metodi euristici, come Sistemi Esperti o Sistemi Rule-Based, talvolta con meccanismi di cheating per ridurre l'impatto delle componenti casuali del gioco [25].

3. Giochi ibridi in tempo reale, che mantengono una base riconducibile ai classici giochi da tavolo, ma che sono caratterizzati da meccaniche di gioco tendenzialmente proprie dei videogames. Iron Duke , il gioco sviluppato nell'ambito di questa tesi rientra in questa categoria, come anche molti altri giochi manageriali e, in alcuni casi, alcuni giochi strategici in tempo reale (RTS).

Analizziamo ora i problemi visti nella Sezione 2.2 (movimento, decision-making e strategia) sotto un'ottica però indirizzata al mondo dei board games.

2.4.1 Il problema del movimento nei board games

Risolvere il problema del movimento delle unità in un board game è piuttosto semplice, in quanto la struttura regolare dell'ambiente di gioco offre più possibilità di soluzione:

- Innanzitutto è sempre possibile utilizzare le tecniche classiche di pathfinding come l'algoritmo A* presentate nella Sezione 2.2.1. La struttura a caselle tipica della plancia di gioco di un board game si adatta bene ad un elaborazione di questo tipo in quanto è facilmente distinguibile quali aree rappresentano un ostacolo e quali invece sono libere per il movimento.

- Oltre agli algoritmi classici, è possibile trattare il problema in forma puramente matematica (Figura 2.16): in una scacchiera o in una mappa esagonale è infatti molto semplice calcolare le distanze tra un punto all'altro dell'area di gioco, verificare in che direzione si trova una casella rispetto ad un'altra, o convertire coordinate globali spaziali in coordinate discrete di casella e viceversa.

- Infine l'implementazione software del board game (soprattutto grazie alla programmazione ad oggetti) permette di modellizzare e di trattare ogni singola casella come un entità a sè del mondo di gioco, caratterizzandola con attributi e funzioni proprie, e con la possibilità di creare dei collegamenti diretti con le caselle adiacenti che rendono immediata la ricerca di un eventuale percorso (Figura 2.17).

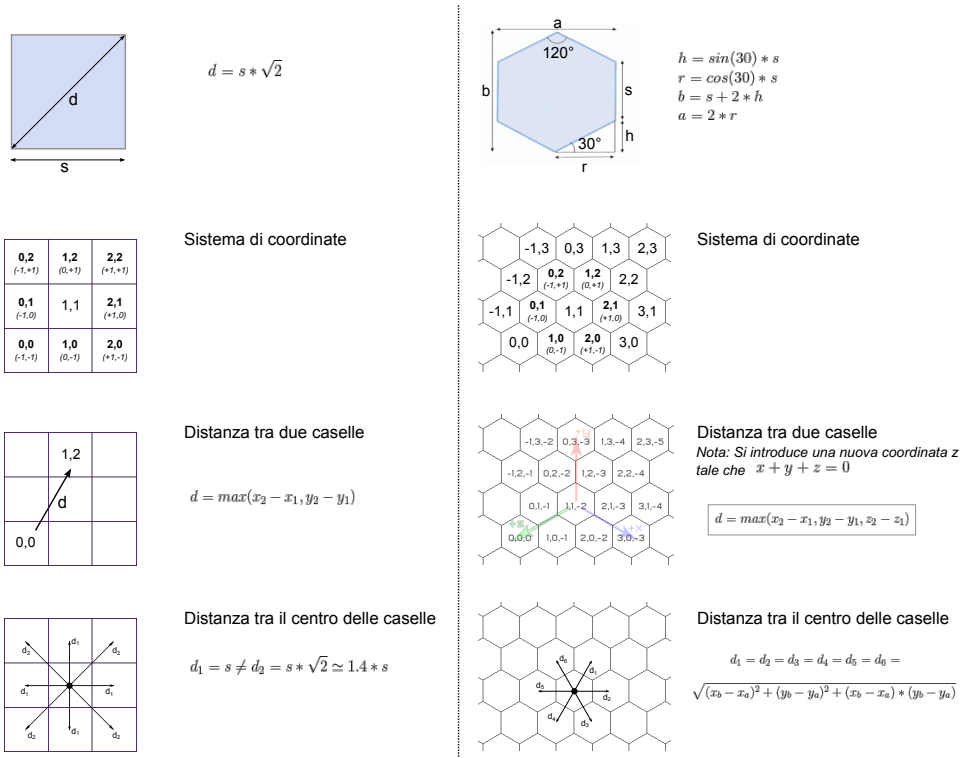


Figura 2.16: Matematica alla base di scacchiere e mappe esagonali

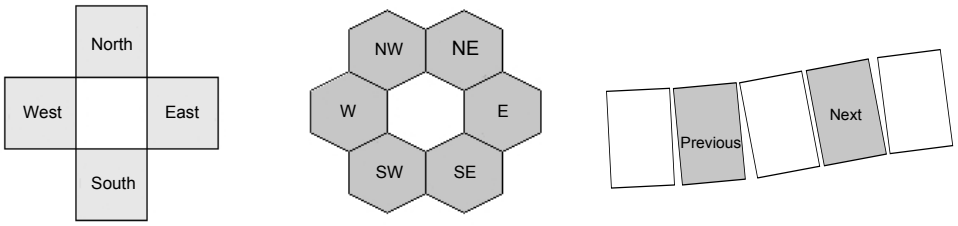


Figura 2.17: Esempio di riferimento alle celle adiacenti in un board game

Di seguito vengono riportati dei frammenti di codice che dimostrano come quest'ultima tecnica sia stata utilizzata per modellizzare la mappa utilizzata nel gioco Iron Duke.

```
Class HexMap
{
        ...
        public enum CellNeighbours { North, NorthWest, NorthEast, East, South, SouthEast, SouthWest, West, NotDefined };
        ...
        //la mappa è formata da una matrice bidimensionale di oggetti HexTile
        private HexTile[,] cells;
}

Class HexTile
{
        ...
        //ogni casella contiene un array con i riferimenti alle 6 celle adiacenti
        private HexTile[] neighbours = new HexTile[6];
        ...
        //funzione che restituisce direttamente una casella vicina data la direzione
        public HexTile getNeighbour(HexMap.CellNeighbours direction)
        //funzione che restituisce direttamente una casella vicina dato un angolo
        public HexTile getNeighbour(float angle_radians)
        //funzione che controlla se una data casella è adiacente a quella corrente
        public bool IsNeighbourOf(HexTile cell)
        //funzione che, data una cella, restituisce la direzione per arrivarci
        public HexMap.CellNeighbours GetAdjacentCellDirection_TO(HexTile cell)
        ...
}
```

2.4.2 Il problema decisionale-strategico nei board games

Nel caso dei board games, il problema decisionale e quello strategico vengono trattati insieme poichè spesso non esiste una vera e propria distinzione tra di essi, o ancora, in molti casi il livello decisionale non ha ragione di essere implementato in quanto in un board game spesso non esistono agenti

autonomi in grado di prendere decisioni, ma tutto viene comandato a livello macroscopico a livello di strategia. D'altro canto, anche se si utilizza un unico livello strategico ci si troverà molto spesso nella situazione di dover prendere delle decisioni, quindi anche le tecniche di decision making già viste si riveleranno molto utili. La tecnica delle mappe di influenza viste nella Sezione 2.2.3 si rivelano particolarmente potenti nell'utilizzo in un board game per via della struttura molto regolare dell'ambiente di gioco. Metodi più formali includono invece tecniche prese direttamente dalla teoria dei giochi e dalla teoria delle decisioni, come l'induzione a ritroso, il metodo minimax e la potatura alfa-beta, che verranno ora descritte più nel dettaglio. Sebbene l'intelligenza artificiale e la teoria dei giochi siano due discipline completamente separate, e la seconda sia incentrata più su aspetti e problemi legati al mondo dell'economia, è stato già dimostrato come le due abbiano connessioni molto profonde, e come la teoria dei giochi possa apportare molti benefici sia al mondo dell'informatica, sia a quello dell'intelligenza artificiale [32]. Anche se è praticamente impossibile implementare un motore di I.A. che si basi esclusivamente su tecniche della teoria dei giochi, gli esempi ora mostrati vogliono dimostrare come in certi casi tali tecniche possano rivelarsi estremamente utili per risolvere piccoli problemi decisionali e strategici anche nell'ambito di un'intelligenza artificiale per giochi, soprattutto board games visto la loro particolare struttura, ma non solo.

Induzione a ritroso

Questo tipo di ragionamento può essere applicato solo per giochi ad informazione perfetta e in forma estesa , ovvero che possono essere rappresentati sotto forma di albero. Il ragionamento è molto semplice: conoscendo già l'esito finale di ciascuna mossa selezionabile da ogni giocatore è possibile ripercorrere l'albero dalla fine verso l'inizio per decidere la strategia migliore da adottare in ciascun suo punto intermedio, e di conseguenza prevedere le mosse dell'avversario e calcolare una strategia vincente finale. Un'esempio tipico dell'applicazione dell'induzione a ritroso è quello del gioco dei tre politici mostrato in Figura 2.18. Ogni politico deve votare (in ordine) a favore o contrariamente ad una proposta a loro vantaggiosa ma poco popolare che passerà solo se ci sarà una maggioranza di voti favorevoli. Rappresentando il problema sotto forma di albero ed applicando l'induzione a ritroso si verifica facilmente che la posizione più vantaggiosa è quella occupata dal primo votante, e che la sua strategia vincente è quella di votare "no" anche se in realtà vuole che la proposta passi. Come è facilmente intuibile è impensabile implementare questa tecnica per risolvere problemi complessi posti dai mo-

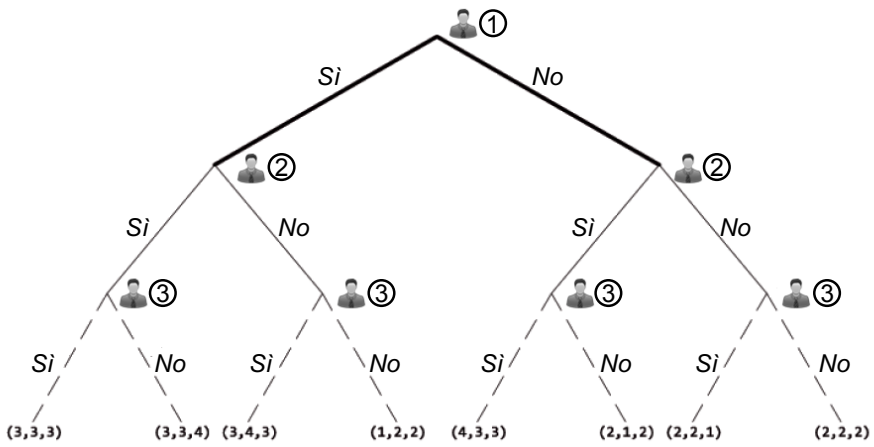


Figura 2.18: Albero del gioco dei tre politici

derni giochi (anche il semplice gioco del tris porta già alla costruzione di un albero con diversi livelli di profondità), tuttavia è possibile utilizzare questo approccio per prendere delle piccole decisioni nei casi in cui il problema sia modellizzabile come un sotto-gioco che presenta queste caratteristiche e sarà così quindi possibile tentare di prevedere le mosse dell'avversario a distanza di un paio di turni.

Metodo minimax

La tecnica di *minimaxing* (Von Neumann, 1928) si applica invece nei casi di giochi a somma zero , ovvero quei giochi in cui ciò che viene vinto da un giocatore è usualmente ciò che viene perso dagli altri giocatori. Anche in questo caso il concetto alla base è molto semplice: quando un giocatore effettua una mossa che lo porta a guadagnare un certo punteggio, cercherà di *massimizzare* tale guadagno. Poichè ciò che un giocatore guadagna corrisponde a ciò che un'altro giocatore perde, quest'ultimo cercherà inevitabilmente di fare una mossa che porterà il primo giocatore a guadagnare il meno possibile e tenterà quindi di *minimizzare le perdite*. Questo conflitto che si viene a creare tra gli obiettivi dei due giocatori (e soprattutto la consapevolezza di quale sia l'obiettivo dell'avversario), porterà alla definizione di una strategia ottimale che porta inevitabilmente ad un punto di equilibrio. Una cosa molto interessante che è possibile notare da questo tipo di giochi è il fatto che seguire una strategia prettamente *greedy* da parte di un giocatore potrebbe portarlo ad un esito finale decisamente a lui sfavorevole come si può vedere in Figura 2.19: se il primo giocatore gioca la prima colonna con la speranza

di ottenere il punteggio massimo (1000) finirà inevitabilmente per ottenere quello minimo (1).

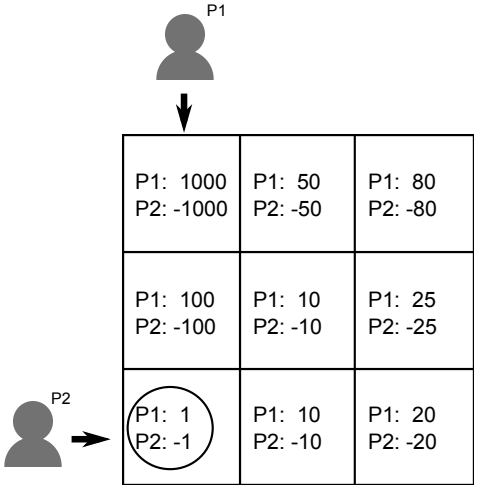


Figura 2.19: Esempio di gioco a somma zero

Anche in questo caso non è ovviamente possibile implementare degli agenti utilizzando esclusivamente tecniche di questo tipo, tuttavia sarebbe possibile includere in questi agenti dei micro-algoritmi di reasoning basati su questo modello che si potrebbero ben adattare a giochi basati su meccanismi economici e nei quali i giocatori si trovano a dover contrattare su un determinato prezzo, per esempio nei giochi manageriali o strategici.

Potatura Alfa-Beta

La potatura alfa-beta è una tecnica aggiuntiva che permette di ridurre drasticamente i nodi visitati dall'algoritmo di minimaxing. La potatura si basa per l'appunto sull'utilizzo di due valori (alfa e beta) che rappresentano in ciascun punto dell'albero la posizione migliore e la posizione peggiore che è possibile raggiungere.
In pratica:

- α è il punteggio *minimo* che il giocatore massimizzante può raggiungere partendo dalla posizione in esame
- β è il punteggio *massimo* che il giocatore massimizzante può raggiungere partendo dalla posizione in esame

Questi valori vengono aggiornati mano a mano che la ricerca progredisce in profondità all'interno dell'albero. Se per un certo nodo il valore di α

diventa maggiore di β si innesca il meccanismo di potatura del sotto albero e la ricerca riprende da un'altro punto. Un'implementazione in pseudocodice di questo algoritmo è riportata nell'Appendice A.

2.5 Conclusioni

L'intelligenza artificiale per giochi e l'intelligenza artificiale "accademica" potrebbero a prima vista sembrare due mondi a sé stanti. Da un lato si è visto che, salvo casi eccezionali, è inutile e controproducente utilizzare tecniche complesse quali reti neurali e algoritmi genetici quando si desidera implementare un motore di I.A. per giochi, e buona regola pare essere quella di sacrificare la complessità e "l'intelligenza" a favore di una maggiore semplicità e "artificialità". D'altra parte invece è stato dimostrato come l'I.A. per giochi non possa comunque fare completamente a meno di tecniche e modelli propri del mondo della ricerca in questo settore, e che in alcuni casi possa trarne indubbiamente beneficio. Seppur i problemi legati all'implementazione di una intelligenza artificiale per giochi siano a prima vista pochi e semplici (movimento, decisione e strategia) le soluzioni sono molteplici e possono trarre origine da svariati campi di ricerca o campi applicativi. Come è stato visto in chiusura di capitolo, un'altro settore che offre molti spunti per l'implementazione di I.A. per categorie particolari di giochi è per l'appunto quello della teoria dei giochi, una branca delle ricerca nata per lo più per risolvere problemi di tipo economico, ma che offre molti spunti per affrontare anche le sfide poste da questo tipo di I.A.

Capitolo 3

Descrizione del gioco Iron Duke

In questo capitolo viene presentato il gioco Iron Duke. Ne vengono prima descritte le regole e le componenti principali, per poi concludere con l'analisi del modello economico alla base del gioco. La parte inerente all'implementazione dell'intelligenza artificiale viene illustrata più nel dettaglio nel Capitolo 4.

3.1 Descrizione del gioco

Iron Duke è un gioco manageriale strategico sviluppato in collaborazione con Spartaco Albertarelli[1] ambientato nell'Inghilterra della seconda metà del XIX secolo, epoca d'oro del trasporto ferroviario. La meccanica di gioco trae spunto da titoli come Railroad Tycoon [Microprose, 1990], Transport Tycoon [Microprose, 1994] e Sid Meyer's Railroad! [Fireaxis Game, 2006],

[1] Game designer milanese, collabora da anni con Editrice Giochi ed è uno dei massimi editori italiani di giochi da tavolo.È stato per anni responsabile editoriale dell'edizione italiana di Dungeons & Dragons ed è responsabile dello sviluppo della linea RisiKo!, oltre ad aver pubblicato decine di giochi sia per adulti che per bambini.

tuttavia il gioco presenta diversi aspetti e regole originali ed è caratterizzato da alcuni elementi riconducibili al mondo dei board games tra cui una mappa a caselle esagonali , la possibilità di vedere più giocatori sfidarsi sulla stessa plancia di gioco, ed un sistema di avanzamento della partita basato su turni, seppur non direttamente visibile al giocatore (vedi Sezione 3.1.2).

3.1.1 Obiettivo del gioco

L'obiettivo del gioco è quello di costruire una rete ferroviaria il più efficiente possibile che permetta di trasportare merci e passeggeri da una città all'altra al fine di guadagnare denaro e punti vittoria. Il gioco prevede due modalità di gioco principali, ognuna con il proprio obiettivo finale:

- Una modalità a giocatore singolo, nella quale l'obiettivo consiste nell'accumulare il maggior numero di punti ed evitare la bancarotta fino al raggiungimento di un termine di tempo stabilito a priori.
- Una modalità multigiocatore, alla quale possono partecipare fino a 4 giocatori che possono essere sia umani che controllati da intelligenza artificiale. In questo caso vince il giocatore che al termine della partita possiede il punteggio più alto. Oltre che per il punteggio base, i giocatori competono per l'assegnazione di 10 premi finali (vedi Sezione 3.3.6) che possono influenzare molto l'esito finale della partita. Un aspetto interessante è che i giocatori umani si sfidano contemporaneamente sullo stesso dispositivo[2], rendendo la meccanica di gioco simile a quella di un vero e proprio gioco da tavolo.

3.1.2 Meccanica di gioco

Il gioco si basa su un meccanismo a turni trasparente al giocatore, ovvero, a quest'ultimo, viene data l'impressione che tutto si svolga in un tempo continuo basato sul naturale passaggio dei mesi e degli anni nei quali il gioco è ambientato. Tuttavia, sia lo scorrere del tempo, sia l'avvenire di certi eventi di gioco, sia il sincronismo tra gli oggetti e persino l'intelligenza artificiale vengono comandati da un sistema a turni basato su *time units* (*T.U.*). L'implementazione di questo sistema è molto semplice e si basa sull'utilizzo di due soli oggetti: il manager dei turni (*TurnManager*) e la classe astratta *TurnSyncedObject*. La Figura 3.1 mostra il diagramma UML del modello utilizzato.

[2]Il gioco è pensato per essere giocato su dispositivi mobili, in particolare tablet.

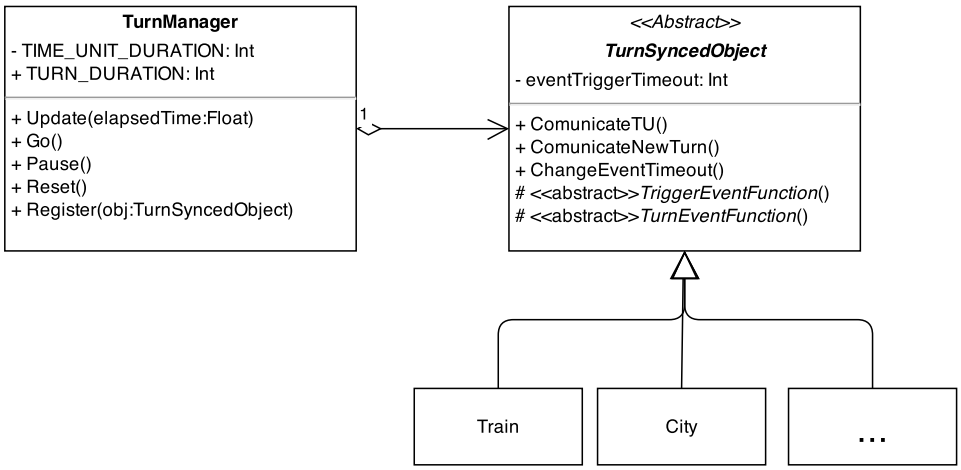


Figura 3.1: Diagramma UML del sistema di gestione dei turni

Il funzionamento del modello è il seguente:

- Il manager dei turni contiene una collezione di oggetti di tipo *TurnSyncedObject* che vengono registrati tramite la funzione *Register*.
  Ogni n frame di update del gioco (con n dato dal parametro *TIME_UNIT_DURATION*) viene fatta avanzare una time unit e l'evento viene comunicato a tutti gli oggetti registrati tramite la funzione *ComunicateTU*. Al passaggio di un numero di *T.U.* specificato dal parametro *TURN_DURATION* termina il turno corrente e anche in questo caso ciò viene comunicato a tutti gli oggetti registrati tramite la funzione *ComunicateNewTurn*. L'avanzamento del tempo di gioco può essere agevolmente interrotto e fatto ripartire in qualsiasi momento tramite le funzioni *Go* e *Pause*.

- La classe astratta *TurnSyncedObject* vincola ogni oggetto da essa derivato ad implementare i due metodi *TriggerEventFunction* e *TurnEventFunction*. Il primo serve ad implementare un evento proprio dell'oggetto che può avvenire dopo il passaggio di un numero prestabilito di time units all'interno del turno. Tale numero può variare di oggetto in oggetto e ci potranno quindi essere oggetti più "veloci" (più eventi per turno) e oggetti più lenti (meno eventi per turno). Per esempio un treno più rapido potrà invocare la propria routine di movimento dopo 10 *T.U.* mentre un treno più lento potrà farlo solo dopo 20 *T.U.* La funzione *TurnEventFunction*, invece, è sincronizzata con la fine di ciascun turno e viene chiamata per *tutti* gli oggetti ogni volta che ciò avviene. Per esempio tutte le città chiameranno le proprie routine di

aggiornamento dei prezzi delle merci allo scoccare di ogni turno, mentre i treni potranno eseguire i calcoli di invecchiamento e manutenzione in modo sincronizzato con il passaggio dei turni.

Quando un giocatore desidera effettuare un'azione di gioco (per esempio costruire un tratto di ferrovia o acquistare un nuovo treno) può interrompere l'avanzamento del tempo in qualsiasi momento. Una volta completata la mossa, il gioco riprenderà esattamente dall'istante temporale nel quale era stato fermato. Questo procedimento è illustrato schematicamente in Figura 3.2:

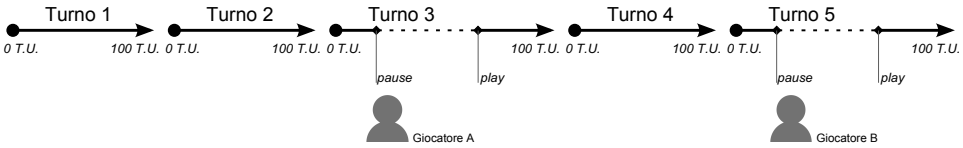


Figura 3.2: Schematizzazione della meccanica di gioco a turni in Iron Duke

3.2 Struttura ed elementi di gioco

In questa sezione vengono descritti nel dettaglio alcuni degli elementi principali che compongono il gioco.

3.2.1 Mappa

Il gioco si svolge su una mappa esagonale rappresentante la cartina del Regno Unito e sulla quale sono disposte 38 città scelte in modo da garantire una copertura più o meno uniforme ed interessante al fine del gioco (Figura 3.3).

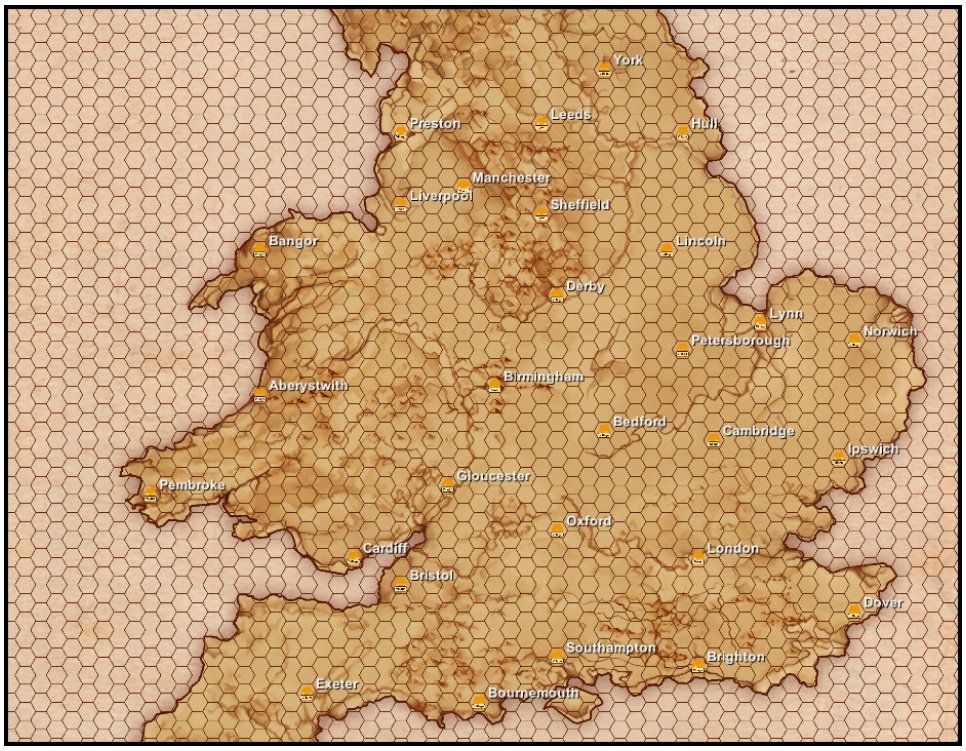


Figura 3.3: Mappa di gioco di Iron Duke

Proceduralmente un livello di gioco viene costruito utilizzando 4 elementi che rendono immediata sia la modifica del livello corrente, sia la creazione di nuove mappe di gioco. Essi sono:

1. Un file grafico rappresentante la mappa di sfondo priva della griglia esagonale. (Nel caso specifico un file .png di dimensione 4096×4096 pixel)

2. Un file grafico utilizzato come maschera per definire le aree di gioco non accessibili, per esempio mare e montagne. Il meccanismo si basa

su un controllo pixel per pixel del colore e le caselle corrispondenti ai pixel rossi vengono marcate come "non accessibili". Per il file maschera non è necessario utilizzare risoluzioni pari a quella dello sfondo, ed è possibile usare un file molto più piccolo in dimensioni.

3. Uno *spritesheet* contenente la grafica delle caselle. La griglia esagonale e le caselle vengono create in modo procedurale via software come verrà spiegato in seguito.

4. Un file XML che contiene la lista delle città più eventuali altre informazioni necessarie alla costruzione del livello, per esempio un'eventuale configurazione iniziale delle merci prodotte da una città oppure la costruzione di un segmento di ferrovia preesistente. La città di Londra per esempio viene istanziata nel gioco tramite il nodo XML:

```
<City name="London" worldPosition_x="0.649"
                    worldPosition_y="0.794">
        <startingLevel>6</startingLevel>
</City>
```

La creazione e la gestione della griglia esagonale viene gestita via software e segue i modelli descritti nella Sezione 2.4.1 a pagina 29: ogni singola cella è modellata come un oggetto a sé, e tutte le funzioni di gestione, ricerca e selezione delle celle sono gestite attraverso formule matematiche. Come si vedrà nel Capitolo 4 i problemi di pathfinding sono risolti tramite metodi che si basano sull'utilizzo dell'algoritmo A*. L'implementazione della mappa è realizzata tramite un modello completamente indipendente dal resto del motore di gioco e si basa sull'utilizzo di due sole classi: *HexTile* e *HexMap*. In Figura 3.4 ne viene presentato il diagramma UML.

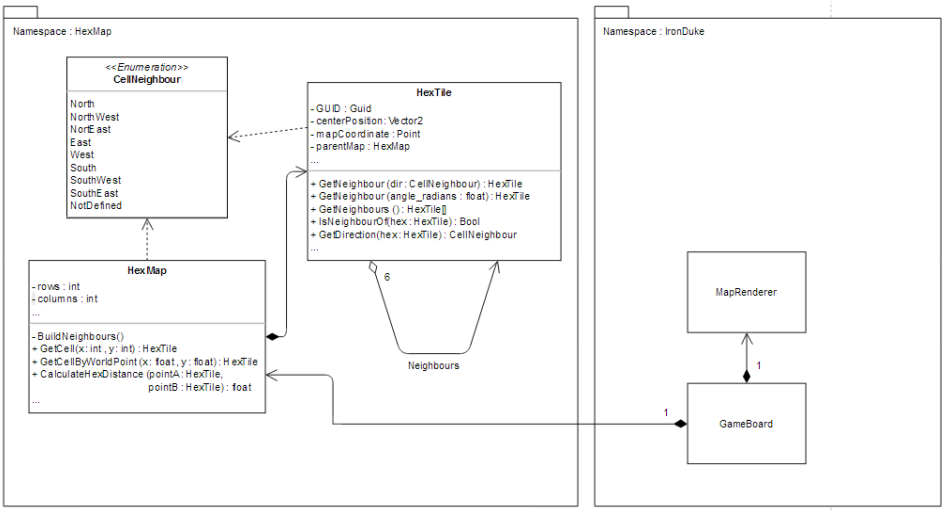


Figura 3.4: Diagramma UML del modello di gestione della mappa

3.2.2 Città

Sulla mappa di gioco sono collocate 38 città scelte sia per la loro importanza storica che per la loro posizione, in modo che venga sempre rispetto un vincolo di distanza minima (non ci possono essere due città a meno di 3 caselle di distanza) e che la loro disposizione offra un'interessante sfida ai giocatori. La funzione delle città è quella di costituire i nodi principali della rete ferroviaria. Ogni città è in grado di generare merci e passeggeri che dovranno essere trasportati al fine di ottenere denaro e punti vittoria. Ciascuna città è caratterizzata e descritta dai seguenti parametri principali:

Parametro	**Descrizione**
Livello	Livello che rappresenta la grandezza della città. Da 1 (villaggio) a 15 (capitale)
Offerta [3]	Array delle 3 merci prodotte o immagazzinate
Domanda [3]	Array delle 3 merci acquistate dalla città
ProductionStrength	Numero di merci prodotte ad ogni turno
Passeggeri [1..3]	Lista delle rotte di passeggeri in partenza (min 1 max 3)
Posta [1..3]	Lista dei sacchi postali in uscita (min 1 max 3)

Tabella 3.1: Parametri descrittivi di una città in Iron Duke

Il modello di domanda e offerta

La domanda e l'offerta di ogni città sono regolate come segue.
Ciascuna città contiene 3 slot di offerta e 3 slot di domanda[3]. La disposizione delle merci all'interno dei 6 slot viene generata in maniera casuale all'inizio di ciascuna partita, in alternativa è possibile specificare delle configurazioni fisse all'interno del file XML utilizzato per la generazione del livello (Vedi Sezione 3.2.1). I tre slot di domanda e di offerta non hanno tutti uguale importanza ma sono ordinati secondo tre livelli: alto, medio e basso, e ad ogni merce è assegnato un valore grazie al quale è possibile determinare su che livello si trova. Poichè la città è un oggetto di tipo *TurnSyncedObject* (Sezione 3.1.2) allo scoccare di ogni turno viene invocata la funzione *TurnEventFunction* che esegue un semplice algoritmo di rimescolamento delle merci sui tre livelli[4], mai in nessun caso tuttavia una merce potrà passare da uno slot di domanda ad uno slot di offerta e viceversa. L'algoritmo di scambio tra slot è il seguente:

$delta \leftarrow random_between(-maxDelta, +maxDelta)$;
se $delta = 0$ ripeti il punto precedente;
$categoria \leftarrow randomPick(domanda, offerta)$;
$slot \leftarrow random_between(0, 2)$;
$merce \leftarrow categoria[slot]$;
if $delta > 0$ **then**
  if $valore(merce) + delta \geq$ soglia minima del livello successivo **then**
    scambia le merci;
  end if
else
  if $valore(merce) - delta \leq$ soglia minima del livello corrente **then**
    scambia le merci;
  end if
end if

L'appartenenza di una merce ad un certo livello ne determina il prezzo di acquisto (vedi Sezione 3.3.2) ed il livello di produzione o meno dalla città. Tutte e tre le merci presenti negli slot offerta verranno rigenerate con il passare del tempo, tuttavia la merce che si trova sul livello alto verrà rigenerata molto più rapidamente delle altre. I rapporti di produzione sono 1, 1/3 ,

[3]Il gioco prevede in tutto 6 merci: cibo, carbone, legna, mattoni, sale e acciaio. Ogni città quindi tratterà sempre in ogni momento tutte le merci disponibili, nessuna esclusa.

[4]Per evitare cambi troppo repentini della situazione di gioco, lo scambio vero e proprio avviene solo dopo un numero di turni preimpostato, ad esempio 10.

1/5. In Figura 3.5 è proposto una schematizzazione del modello di domanda e offerta:

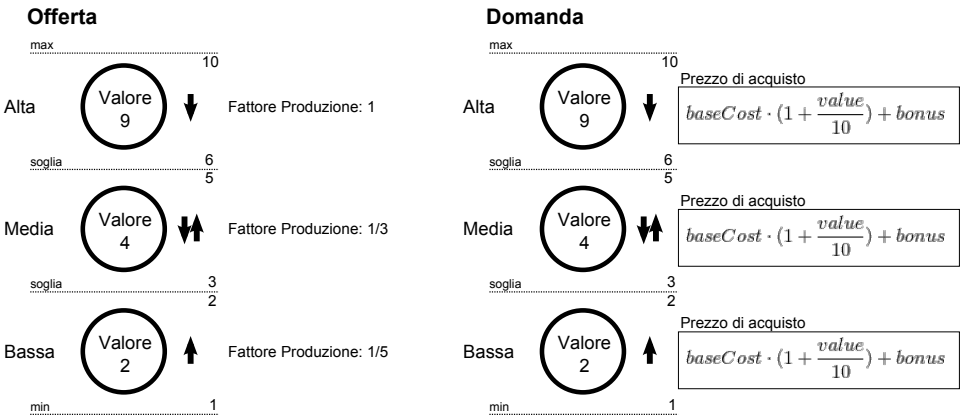


Figura 3.5: Schema del modello di domanda e offerta di una città

Passeggeri e posta

Oltre alle merci ciascuna città genera anche delle rotte passeggeri e postali che si basano su di un meccanismo leggermente diverso da quello visto precedentemente. A differenza delle merci , che possono essere consegnate e vendute a qualunque città ne faccia richiesta, i passeggeri e la posta devono essere consegnate ad una città specifica. All'inizio della partita ogni città genera a caso una rotta passeggeri e una rotta postale. Al completamento di un certo obiettivo (ad esempio trasportare con successo 50 passeggeri o 50 unità di posta) tale rotta viene definita soddisfatta e ne viene creata una nuova verso una città differente, fino ad un massimo di 3 rotte passeggeri e 3 rotte postali per città. I passeggeri e la posta aggiungono quindi un'ulteriore livello di sfida al gioco ed una meccanica differente ed originale rispetto a giochi simili che in genere permettono di scaricare queste due tipologie di carico in una qualunque città senza mai specificarne la destinazione.

Livello e premi

Ad ogni città è associato un livello che ne rappresenta la grandezza e l'importanza, che può andare dal valore 1 (villaggio) al valore 15 (capitale). Il livello della città determina quante merci possono venire prodotte in un turno, quanti passeggeri e quanta posta ci possono essere in attesa in un dato istante e in generale quanto è più difficile riuscire ad ottenere premi e

ricompense dalla città. All'inizio della partita le città partono da un livello determinato sulla base della popolazione storica dell'epoca riportata in Tabella 3.2, nel caso in cui tale dato non sia disponibile si considera un livello iniziale pari a 1.

n	Nome	Pop.	Liv.	n	Nome	Pop.	Liv.
1	London	2,804,000	6	20	Bristol	-	1
2	Liverpool	443,900	4	21	Cambridge	-	1
3	Glasgow	360,000	4	22	Cardiff	-	1
4	Manchester	338,300	3	23	Carlisle	-	1
5	Birmingham	296,000	3	24	Dover	-	1
6	Leeds	207,200	3	25	Dumfries	-	1
7	Edimburgh	191,000	3	26	Exeter	-	1
8	Sheffield	185,200	3	27	Gloucester	-	1
9	Newcastle	109,300	3	28	Ipswich	-	1
10	Hull	99,000	2	29	Lincoln	-	1
11	Preston	83,000	2	30	Lynn	-	1
12	Derby	43,100	1	31	Middlesboro	-	1
13	York	40,400	1	32	Norwich	-	1
14	Aberdeen	-	1	33	Oxford	-	1
15	Aberystwith	-	1	34	Pembroke	-	1
16	Bedford	-	1	35	Petersborough	-	1
17	Bangor	-	1	36	Plymouth	-	1
18	Bournemouth	-	1	37	Southampton	-	1
19	Brighton	-	1	38	Whitehaven	-	1

Tabella 3.2: Popolazione storica (1860) e livello iniziale delle città.

Ciascuna città mette a disposizione 9 premi legati al raggiungimento di 9 obiettivi inerenti alla consegna delle merci e dei passeggeri: 3 premi per la consegna delle merci richieste, 3 premi per le tre rotte dei passeggeri e 3 premi per le tre rotte postali. Ogni volta che un premio viene assegnato ad un giocatore la città cresce ed aumenta di livello, è quindi evidente che non tutte le città sono in grado di raggiungere il livello massimo e che la maggior parte di esse potrà crescere al massimo fino al raggiungimento del livello 10.

3.2.3 Rete ferroviaria

Binari

Lo scopo del gioco è quello di costruire una rete ferroviaria efficiente che permetta di guadagnare il numero maggiore di punti a fine partita. Ai giocatori viene quindi data la possibilità di costruire dei binari sulla plancia di gioco con l'unico vincolo che ogni segmento debba iniziare o terminare su almeno un punto della propria rete oppure su una città, e deve sempre terminare su un altro segmento o su una città. Non è quindi possibile costruire segmenti isolati, incompleti o che collegano solo pezzi di tracciato posseduti da altri giocatori. Ci sono quattro diversi tipi di binario tra cui scegliere: binario singolo, binario singolo con segnaletica, binario doppio e binario doppio con segnaletica, che offrono diverse possibilità di movimento ai treni che devono attraversarli come mostrato in Figura 3.6:

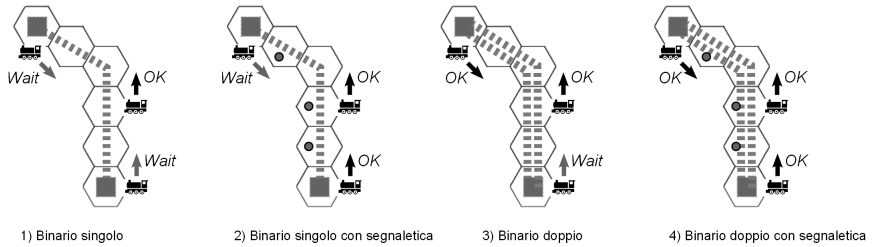


Figura 3.6: I diversi livelli di binario

I binari singoli possono ospitare solo un treno per segmento[5], mentre l'aggiunta di segnaletica offre la possibilità di avere più di un treno che si muove nella stessa direzione. Il binario doppio funziona in modo analogo ma permette di avere contemporaneamente più treni che si muovono in direzione opposta. Inoltre il binario doppio con segnaletica è l'unico che offre la possibilità di costruire scambi ferroviari.

Per evitare situazioni di conflitto o comportamenti sleali da parte dei giocatori, ad esempio la costruzione di segmenti di ferrovia con il solo scopo di ostacolare le mosse dell'avversario, ad un giocatore viene data la possibilità di collegarsi anche alla rete di un'altro giocatore pagando eventualmente l'aggiornamento necessario alla costruzione di uno scambio. Ogni volta che un giocatore si collega ad un tratto posseduto da un'altro giocatore viene

[5]Per segmento si intende un tratto di binari che va da stazione a stazione, da una stazione a uno scambio, o da scambio a scambio

pagata a quest'ultimo una tassa fissa di collegamento, rendendo così ancor più interessanti le scelte strategiche e la meccanica di gioco.

Stazioni e scali ferroviari

Le stazioni vengono create in modo automatico ogni volta che si costruisce un binario sopra una casella contenente una città. Ogni stazione può contenere un numero indefinito di treni e collega i binari di tutte le caselle adiacenti. Gli hub ferroviari devono essere invece costruiti esplicitamente dal giocatore e hanno un costo molto maggiore rispetto ad una normale stazione, eccetto il primo che è gratuito. Gli hub rappresentano dei punti nevralgici di tutta la rete ferroviaria in quanto sono gli unici punti che permettono l'inserimento di nuovi treni e la manutenzione straordinaria di quest'ultimi.

3.2.4 Treni

I giocatori possono costruire dei treni che verranno poi immessi sulla rete ferroviaria nei punti contenenti degli hub di loro proprietà, fino ad un massimo di 25 a testa. Ogni treno è costituito da una locomotiva e da un massimo di 5 vagoni che si dividono in vagoni merci, vagoni passeggeri e vagoni posta. A livello implementativo anche i treni sono oggetti di tipo *TurnSyncedObject*. La funzione *TriggerEventFunction* viene chiamata più o meno velocemente a seconda del parametro velocità della locomotiva e gestisce le operazioni di movimento e carico/scarico merci, mentre la funzione *TurnEventFunction*, che è sincronizzata con il passaggio del turno, gestisce l'invecchiamento del treno e la sua manutenzione. Un manager dei treni viene infine utilizzato per la gestione, la creazione e la ricerca dei treni. Il diagramma UML in Figura 3.7 mostra come tale modello viene implementato nel gioco.

Locomotive

Nel gioco sono presenti 9 diverse classi di locomotive suddivise in 3 fasce di qualità e avanzamento tecnologico. All'inizio della partita saranno disponibili solo le locomotive di fascia bassa e le successive verranno sbloccate in base al passare del tempo e ad un investimento nella ricerca deciso dal giocatore. Ciascuna fascia è caratterizzata dalla presenza di 3 tipi di locomotive che si distinguono per il loro rapporto tra velocità ed efficienza di trasporto e per la loro affidabilità per quanto riguarda la manutenzione. Ogni fascia conterrà sempre una locomotiva veloce ma con poca capacità di carico, una molto efficiente ma lenta, e una equilibrata. Questi valori sono visibili in

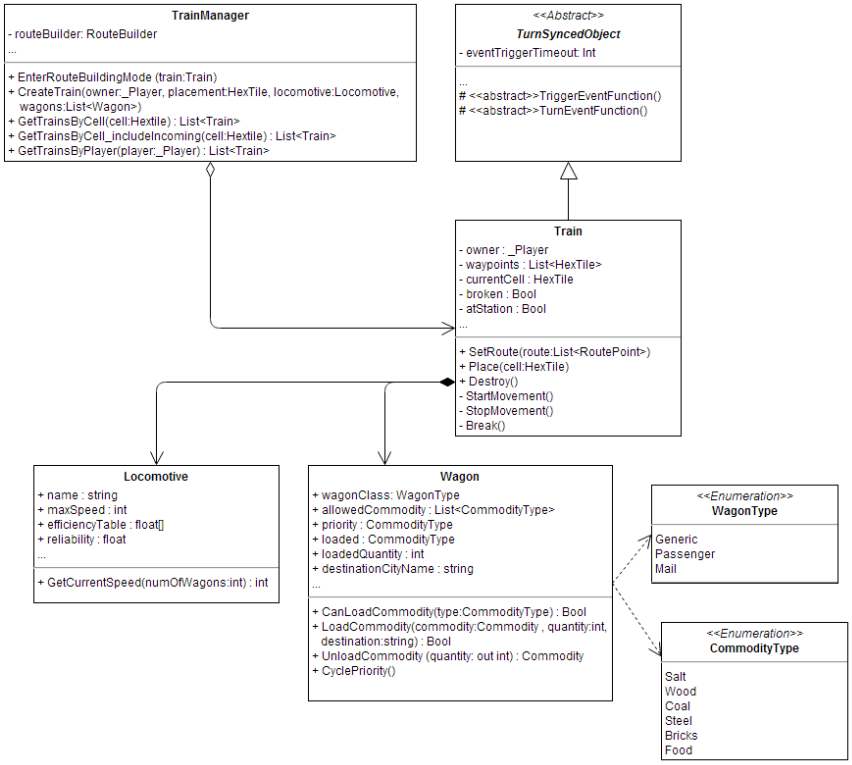


Figura 3.7: Diagramma UML del modello di gestione dei treni in Iron Duke

Tabella 3.3.

Velocità in base al numero di vagoni[6]						**Costo**[7]	**Affidabilità**
	1	2	3	4	5		
A	30%	30%	30%	20%	10%	100%	35%
B	50%	30%	20%	10%	0	120%	30%
C	60%	40%	10%	0	0	150%	30%
D	50%	50%	50%	30%	20%	190%	65%
E	60%	50%	40%	20%	0	240%	70%
F	70%	70%	30%	0	0	300%	75%
G	60%	60%	60%	40%	30%	370%	95%
H	80%	70%	60%	40%	0	450%	100%
I	100%	100%	80%	0	0	550%	100%

Tabella 3.3: Tabella di efficienza delle locomotive in Iron Duke

[6]Rispetto al parametro FASTEST_ENGINE_SPEED
[7]Rispetto al parametro BASE_COST

Le locomotive sono descritte principalmente da 7 parametri, che sono:

Parametro	**Descrizione**
FASTEST_ENGINE_SPEED	Velocità della locomotiva più rapida
BASE_COST	Costo della locomotiva più lenta
costRatio	Costo della locomotiva in rapporto al valore BASE_COST
efficiencyTable[ ]	Valori di efficienza in funzione del numero di vagoni trasportati (Vedi Tabella 3.3)
reliability	Affidabilità della locomotiva, usata negli algoritmi per la manutenzione
reliability_decay	Quanta affidabilità viene "consumata" per ogni casella di movimento. Quando l'affidabilità raggiunge lo 0% il treno deve andare in manutenzione
mainteinance_baseCost	Prezzo base di manutenzione della locomotiva, calcolato in rapporto al suo valore
mainteinance_runningCosts	Costo di esercizio della locomotiva, addebitato ad ogni casella di movimento

Tabella 3.4: Parametri principali delle locomotive in Iron Duke

Con il passare del tempo le locomotive si usurano e necessitano di manutenzione. L'algoritmo utilizzato è il seguente e viene applicato ogni volta che il treno si muove di una casella:

```
if broken = false then
  mainteinance_currentLevel ← mainteinance_currentLevel−reliabilityDecay;
  if mainteinance_currentLevel ≤ 0 then
    broken ← true;
    breakdown_count ← breakdown_count + 1;
    if breakdown_count mod 10 = 0 then
      Manda il treno in manutenzione straordinaria;
    else
      Manda il treno in manutenzione ordinaria;
    end if
  end if
end if
```

Quando il treno raggiunge una stazione o un hub per la manutenzione, viene riparato e l'affidabilità massima decresce, in modo da renderlo sempre più inaffidabile con il passare del tempo:

$broken \leftarrow$ **false**;
$reliability \leftarrow max(5\%, reliability * 95\%)$;
$mainteinance_currentLevel \leftarrow reliability$;

Analizzando la serie seguente è possibile calcolare la vita massima (in caselle) di ciascuna locomotiva

$$ls = \frac{R}{d} + \frac{R*0.95}{d} + \frac{R*0.95^2}{d} + \cdots + \frac{0.05}{d} = \sum_{i=0}^{\log_{0.95}\frac{0.05}{R}} \frac{R*0.95^i}{d}$$

con ls = lifespan, R = reliability, d = reliability_decay

che con pochi passaggi ed applicando la serie notevole diventa:

$$ls = \sum_{i=0}^{\log_{0.95}\frac{0.05}{R}} \frac{R*0.95^i}{d} = \frac{R}{d}\sum_{i=0}^{\log_{0.95}\frac{0.05}{R}} 0.95^i = \frac{R(1-0.95^{\log_{0.95}\frac{0.05}{R}})}{0.05d}$$

Grazie a questa formula è possibile stabilire dei parametri per d in modo da ottenere dei tempi di vita equilibrati per ogni locomotiva.

Vagoni

Una volta costruita la locomotiva è necessario acquistare fino ad un massimo di 5 vagoni per completare il treno. Non tutte le locomotive (Tabella 3.3) sono in grado di trasportare tale numero di vagoni ed alcune non potranno muoversi se superano la quantità massima da loro consentita. I vagoni si dividono in tre classi principali:

- **Vagoni merci:** possono trasportare una qualsiasi delle sei merci disponibili (cibo, carbone, legna, mattoni, acciaio e sale). In qualsiasi momento è possibile assegnare una priorità di trasporto in modo che venga trasportata, se disponibile, la tipologia di merce desiderata. Ciò risulta molto utile quando si desidera effettuare delle consegne al fine di ottenere un premio specifico tra quelli messi a disposizione dalle città (vedi Sezione 3.3.6).

- **Vagoni passeggeri:** destinati esclusivamente al trasporto di passeggeri. Ogni vagone è in grado di caricare una singola unità di passeggeri in attesa in una città.

- **Vagoni postali:** funzionano in modo analogo ai vagoni passeggeri.

Il gioco utilizza un'algoritmo di carico delle merci intelligente: se non viene specificata alcuna priorità di carico viene selezionata automaticamente la merce che garantisce il maggior introito possibile in termini di denaro, considerando solo le stazioni in cui il treno effettuerà delle soste. I passeggeri e la posta vengono caricati automaticamente solamente se il percorso del treno prevede una sosta nelle città di destinazione, in caso contrario i passeggeri e i sacchi di posta eventualmente disponibili verranno ignorati anche se il treno dispone dei vagoni adeguati.

3.3 Il modello economico

Iron Duke si basa su di un modello economico semplificato che viene ora descritto ed analizzato. Tutte le possibili voci di spesa e di guadagno presenti nel gioco sono elencate in Tabella 3.5:

	Ricavi		**Spese**
A	Consegna delle merci	I	Costi di costruzione (Rete e treni)
B	Vendita delle locomotive	J	Costi di manutenzione
C	Incasso tasse di collegamento	K	Costi di esercizio
D	Vincita di premi	L	Finanziamento della ricerca
E		M	Costi di collegamento
F		N	Costi di carico e scarico delle merci

Tabella 3.5: Elenco di tutte le voci economiche di Iron Duke

Sarà quindi necessario costruire un modello che cerchi il più possibile di equilibrare gli ingressi e le uscite al fine di fornire al giocatore un'esperienza di gioco competitiva ma allo stesso tempo divertente. In particolar modo si dovrà porre attenzione ai costi vivi (Voci A, J, K, N) in quanto una cattiva gestione porterebbe da un lato ad un rapido esaurimento delle risorse economiche, dall'altro ad un guadagno troppo facile che renderebbe il gioco poco interessante. Si valuta che un buon compromesso possa essere quello di un guadagno intorno al 30-40% rispetto ai costi ed è sulla base di tali stime che verranno eseguiti i calcoli per calibrare i parametri del modello. Un ulteriore buon punto di partenza è quello di fissare a priori una ripartizione dei costi, in modo da avere delle linee guida su cui orientarsi. Data un'ipotetica spesa di 10000 unità di valuta per un treno che viaggia tra due città poste ad una distanza media di 5 caselle, i costi potrebbero essere ragionevolmente ripartiti come mostrato in Tabella 3.6.

	Voce	min %	max %	Costo(min)	Costo (max)
I	**Costruzione iniziale**	**15%**	**30%**	**1500**	**3000**
I1	*Binari*	*80%*	*90%*	*1200*	*2700*
I2	*Stazioni*	*10%*	*20%*	*150*	*600*
I	**Acquisto treno**	**35%**	**55%**	**3500**	**5500**
I3	*Locomotiva*	*90%*	*95%*	*3150*	*4950*
I4	*Vagoni*	*5%*	*10%*	*175*	*550*
N	**Costi di carico/scarico**	**2,5%**	**5%**	**250**	**500**
K	**Costi di esercizio**	**15%**	**20%**	**1500**	**2000**
J	**Costi di manutenzione**	**20%**	**25%**	**2000**	**2500**
J1	*Ordinaria*	*55%*	*70%*	*1100*	*1750*
J2	*Straordinaria*	*30%*	*45%*	*600*	*945*

Tabella 3.6: Ripartizione teorica dei costi

3.3.1 Modello implementativo

I guadagni e le spese di ciascun giocatore possono essere facilmente gestiti in qualsiasi punto del codice tramite la semplice chiamata delle funzioni *Bill* e *Accreditate* grazie all'implementazione di un modello statico mostrato nel diagramma UML in Figura 3.8. Per ciascun giocatore viene inoltre memorizzato un report contenenti tutte le voci di budget che può essere molto utile sia in fase di gioco sia per l'analisi statistica del modello.

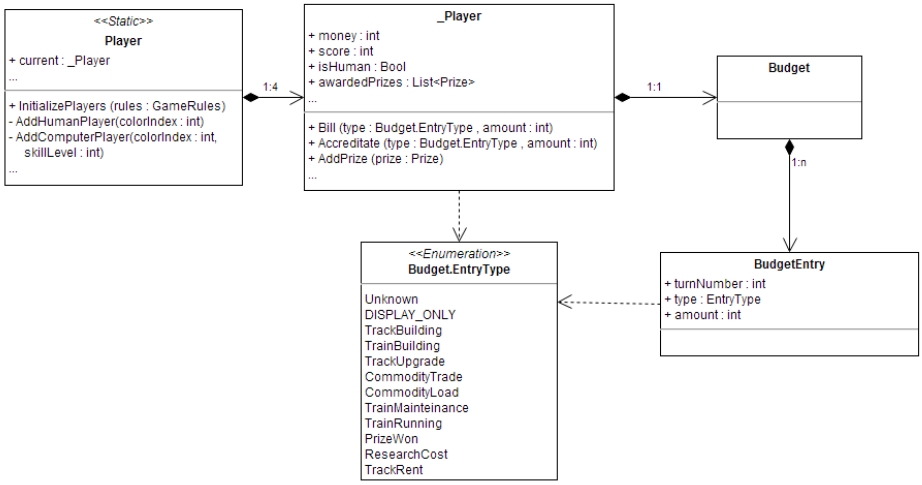


Figura 3.8: Diagramma UML del modello di gestione del budget

Per mantenere coerenza tra tutti i costi presenti nel gioco ciascun valore viene calcolato in modo relativo rispetto ad un'altro. Fissato il costo base della locomotiva più lenta, tutti gli altri prezzi vengono calcolati seguendo lo schema mostrato in Figura 3.9:

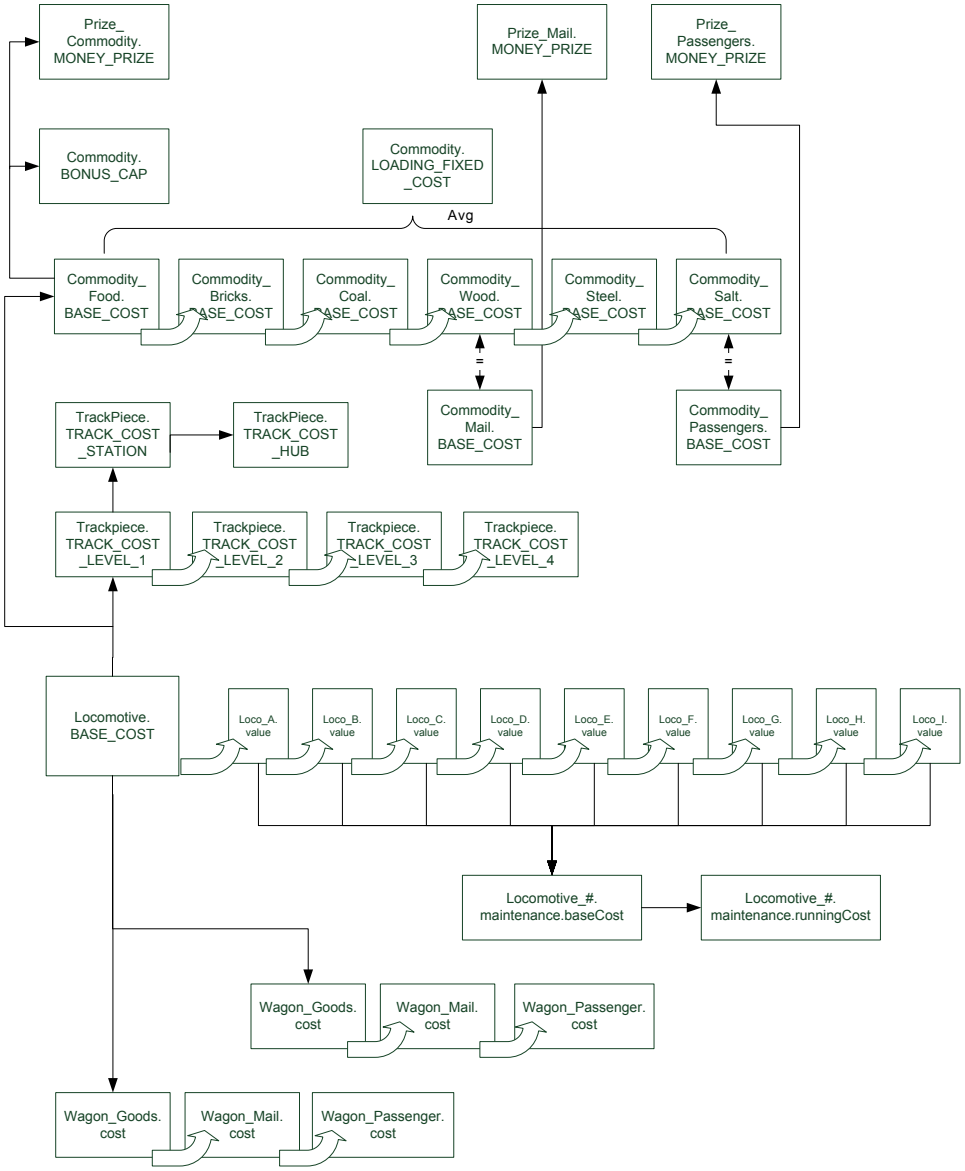


Figura 3.9: Diagramma dei costi in Iron Duke

3.3.2 Il mercato delle merci

Nel gioco sono presenti 6 diversi tipi di merci che seguono un modello base di domanda e offerta. Ogni città all'interno del gioco richiederà sempre contemporaneamente 3 merci e offrirà le rimanenti 3 (vedi Sezione 3.2.2 per maggiori dettagli). Analizzando giochi simili [33, 21] si nota come la scelta dei prezzi base delle merci non segua delle regole precise, ma pare tuttavia buona norma rimanere entro un intervallo di costo che rimanga entro il 250-300% rispetto al prezzo della merce più economica. Anche per Iron Duke si è scelto quindi di adottare un intervallo simile, come mostrato in Figura 3.10.

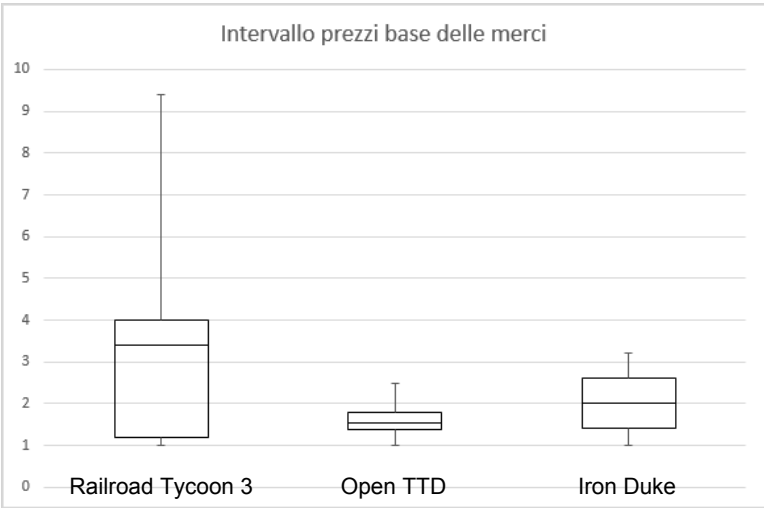


Figura 3.10: Analisi dei prezzi delle merci in giochi simili ad Iron Duke

I rapporti di prezzo utilizzati in Iron Duke sono riportati in Tabella 3.7:

Cibo	1
Mattoni	1,2
Carbone	1,5
Legna	2
Acciaio	2,4
Sale	3,2

Tabella 3.7: Rapporto tra i prezzi delle merci in Iron Duke

A differenza di passeggeri e posta, il cui funzionamento verrà descritto nella prossima sezione, il ricavo derivante dalla consegna delle merci è indipendente sia dalla distanza che dal tempo di trasporto.

Oltre al prezzo base di una merce, il guadagno totale varia in modo dinamico a seconda di quanta merce viene consegnata in una determinata città nel modo seguente: se la merce non viene consegnata per un numero prestabilito di turni ($delayMax$), al suo valore viene aggiunto un bonus che aumenta al passaggio di ogni turno fino a raggiungere un tetto massimo. Non appena la merce viene consegnata, il bonus comincia a decrescere fino a tornare a zero. Nel caso in cui un giocatore riesca a raggiungere un particolare obiettivo di consegna e vincere il relativo premio, si ritiene che la domanda della città sia soddisfatta ed il valore base della merce crolla al 33%. Il valore di consegna all'istante t di una merce può essere quindi calcolato come:

$$value_t = \begin{cases} v + bonus_{t-1} * 0,8 & \text{se } \ d < delay_{max} \\ v + \text{MIN}(bonus_{max}, (d - delay_{max})^{1.5}) & \text{se } \ d \geq delay_{max} \end{cases}$$

con

$$v = \begin{cases} c_0 * (1 + \frac{level}{10}) & \text{se premio ancora disponibile} \\ c_0 * (1 + \frac{level}{10}) * 0.33 & \text{se premio già consegnato} \end{cases}$$

e

$bonus_{max} = c_0 * 2$
$d =$ delay , ovvero il ritardo in turni dall'ultima consegna effettuata.
$delay_{max} = 10$ turni
$c_0 =$ Commodity.BASE_COST, ovvero il prezzo base della merce.

La variabile *level* è un valore che va da 1 a 10 che rappresenta il livello di richiesta della merce all'interno della città e che viene utilizzato per determinare se la merce appartiene alla fascia bassa, media o alta come descritto nella Sezione 3.2.2 a pagina 43. In pratica una città è disposta a pagare come bonus fino al doppio del valore base della merce nel caso in cui essa sia altamente richiesta. Anche se a prima vista questo valore può sembrare alto, come si può vedere dal grafico in Figura 3.11 il bonus decresce praticamente subito una volta effettuata la prima consegna.

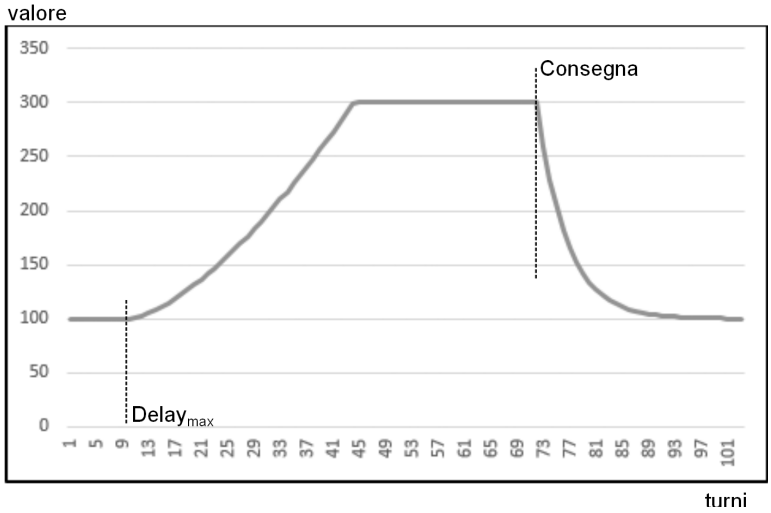


Figura 3.11: Grafico dell'andamento del prezzo di una merce in funzione dei ritardi di consegna

3.3.3 Passeggeri e posta

Il calcolo dei guadagni derivanti dalla consegna di passeggeri e posta funziona in modo differente rispetto a quello delle merci e tiene in considerazione sia la distanza tra le città sia la velocità di consegna.
Il procedimento è il seguente:

1. Viene calcolata la distanza media tra tutte le città.

2. Viene calcolato a priori un bonus a seconda dello scostamento tra la distanza di trasporto e la media calcolata al punto 1 secondo la Tabella 3.8:

Scostamento dalla media	**Casi possibili**	**Tipo di bonus**	**Valore bonus**
+100% , $+\infty$	31 (4,4%)	Bonus massimo	+150% del valore base
+61% , +99%	62 (8,8%)	Bonus alto	+125% del valore base
+31% , +60%	93 (13,2%)	Bonus medio-alto	+100% del valore base
-29% , +30%	276 (39,2%)	Bonus medio	+75% del valore base
-49% , -30%	117 (16,6%)	Bonus basso	+50% del valore base
-74% , -51%	97 (13,7%)	Bonus minimo	+25% del valore base
$-\infty$, -75%	27 (3,8%)	Nessun bonus	Nullo

Tabella 3.8: Fasce di bonus per il trasporto di passeggeri e posta

Le soglie sono state scelte in modo da garantire una distribuzione sensata tra i vari casi possibili offerti dalla disposizione delle città e da favorire leggermente i casi favorevoli (l'insieme dei bonus medio, medio-alto, alto e massimo formano coprono insieme il 65% dei casi possibili) come si può vedere dal grafico in Figura 3.12:

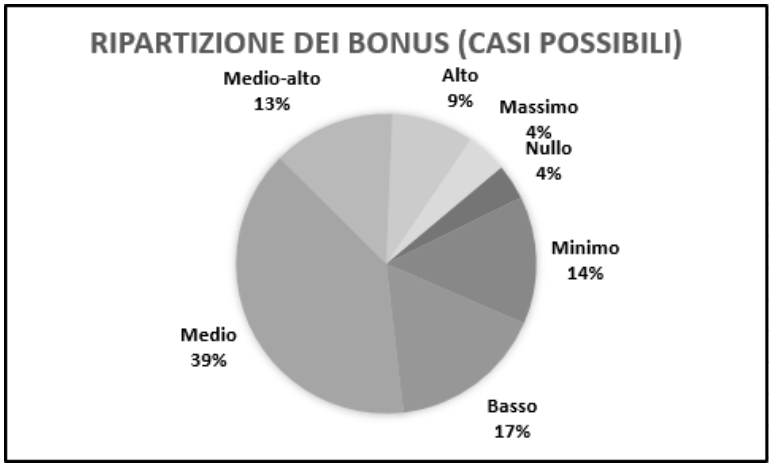


Figura 3.12: Ripartizione dei bonus per passeggeri e posta

3. Per ciascuna fascia di bonus viene assegnato un valore di decremento percentuale ($timePenalty$) che viene applicato al passare di ogni turno. È stato scelto di utilizzare un valore diverso per ogni fascia in modo da rendere più facile la ricerca di un equilibrio tra distanza di trasporto e rapidità di consegna.

4. Alla consegna il valore totale può essere quindi calcolato come:

   $$v = \text{MAX}(\frac{v_0}{2}, v_f - (v_f \cdot timePenalty \cdot deliveryTime))$$

   con
   v_0 = valore base
   $v_f = v_0 + initialBonus$

Fissato quindi un numero di turni entro il quale andrebbe fatta la consegna ($targetTime$) per ciascuna fascia di distanza è possibile calcolare un valore ideale di penalità:

$$timePenalty = \frac{2 \cdot v_f - v_0}{2 \cdot v_f \cdot targetTime}$$

3.3.4 I costi di costruzione

Il calcolo dei costi di costruzione della rete andrebbe fatto analizzando dapprima l'utilità dei diversi livelli di binario disponibile. Sebbene a prima vista possa sembrare che esista un semplice rapporto lineare tra l'utilità di questi ultimi (ad un binario doppio corrisponde una consegna doppia e quindi un guadagno doppio) in realtà il fatto che la città possa produrre solo un numero ridotto di merci per turno potrebbe far sì che una consegna troppo rapida faccia terminare rapidamente le scorte, portando così a zero ogni guadagno. Il modello teorico che è stato scelto per l'analisi dell'utilità dei binari è il seguente:

$$score_i = \begin{cases} throughput_i & \text{se } \begin{matrix} productionRatio \geq \\ missThreshold_i \end{matrix} \\ throughput_i - \dfrac{0.7 * 2}{d} \cdot misses_i & \text{altrimenti} \end{cases}$$

Le ipotesi utilizzate per la formazione di questo modello sono le seguenti:

1. Ci sono n treni che si spostano tra due città poste a d caselle di distanza. Poichè l'assenza di segnaletica permette il passaggio di un solo treno per segmento avremo contemporaneamente al massimo $n = 1$ treni attivi su binario singolo e $n = 2$ treni attivi su binario doppio. La presenza di segnaletica permette invece di avere al massimo $n = d$ e $n = 2d$ treni attivi contemporaneamente rispettivamente su binario singolo con segnaletica e doppio con segnaletica. In pratica si suppone una "coda" di treni che occupa l'intera distanza tra le due città.

2. I treni si muovono alla velocità puramente ipotetica di una casella per turno.

3. Per ogni livello di binario è quindi possibile calcolare il *throughput* di consegne per turno che possono essere portate a termine:

$$throughput_1 = \frac{1}{d} \quad throughput_2 = \frac{2}{d}$$

$$throughput_3 = 0.5 \quad throughput_4 = 1$$

4. Il fatto che un treno inizi a mancare delle consegne o meno è regolato dai due parametri p ($productionRatio$) e $M_i(missThreshold)$ che sono rispettivamente un'indice indicativo sulla velocità di produzione ed una soglia calcolata per simulazione che determina il punto critico sotto il quale il treno inizia a fare delle consegne a vuoto.

5. Infine il valore di penalità è calcolato sulla base delle ipotesi che le spese di trasporto equivalgano al 70% dei guadagni come ipotizzato in apertura della Sezione 3.3 e che ciascun viaggio a vuoto comporti sia l'andata che il ritorno del treno.

Nello specifico le formule per i 4 livelli[8] di binario sono:

$$score_1 = \begin{cases} \dfrac{1}{d} & \text{se } \; p \geq \dfrac{1}{2 \cdot d} \\ \dfrac{1}{d} - \dfrac{0,7 * 2}{d} \cdot \dfrac{1}{2 \cdot d \cdot p} & \text{altrimenti} \end{cases}$$

$$score_2 = \begin{cases} \dfrac{2}{d} & \text{se } \; p \geq \dfrac{1}{d} \\ \dfrac{2}{d} - \dfrac{0,7 * 2}{d} \cdot \dfrac{1}{d \cdot p} & \text{altrimenti} \end{cases}$$

$$score_3 = \begin{cases} 0,5 & \text{se } \; p \geq 0,5 \\ 0,5 - \dfrac{0,7 * 2}{d} \cdot \dfrac{0,5}{p} & \text{altrimenti} \end{cases}$$

$$score_4 = \begin{cases} 1 & \text{se } \; p \geq 1 \\ 1 - \dfrac{0,7 * 2}{d} \cdot \dfrac{1}{p} & \text{altrimenti} \end{cases}$$

I risultati delle simulazioni sono mostrati nei grafici di Figura 3.13 dai quali risulta evidente come l'utilità dei binari con segnaletica sia particolarmente sensibile sia alla distanza, sia alla velocità di produzione delle città, mentre rimane pressochè invariata nel caso di binari senza segnaletica.

[8] Livello 1: binario singolo, Livello 2: binario doppio, Livello 3: binario singolo con segnaletica, Livello 4: binario doppio con segnaletica

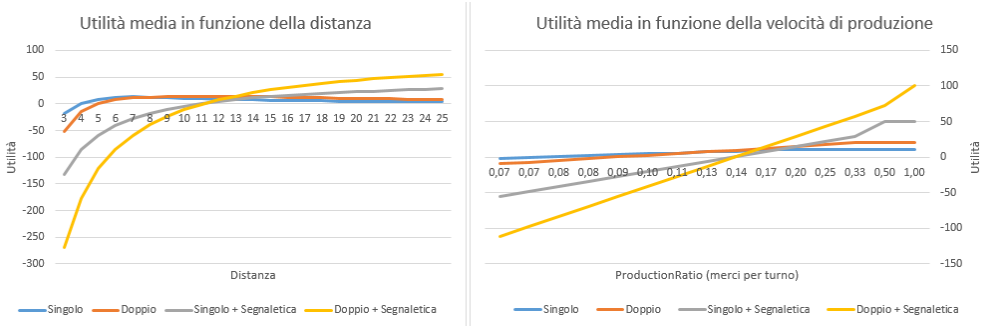


Figura 3.13: Grafici di utilità dei diversi livelli di binario

La Tabella 3.9 riporta i rapporti tra i valori trovati nella simulazione. Da essi è possibile stimare i costi dei vari livelli di tracciato. Il costo delle stazioni e degli scali ferroviari può essere fissato a posteriori senza particolari calcoli in modo da bilanciare eventuali squilibri tra spese e guadagni, mantenendo comunque una ragionevole differenza minima rispetto al costo base del singolo binario.

Elemento	**Max [Utilità]**	**Mediana [Utilità]**	**Costo finale**
Binario singolo	1	1	1
Binario singolo con segnaletica	1,5	1,6	1,6
Binario doppio	2	1,68	2,5
Binario doppio con segnaletica	3	3,2	3,3
Stazione	-	-	$\geq$4
Scalo	-	-	$\geq$12

Tabella 3.9: Rapporto tra i prezzi di costruzione della rete ferroviaria in Iron Duke

3.3.5 I costi di manutenzione

La manutenzione rappresenta un aspetto fondamentale del modello economico in quanto è la spesa principale in baso di controbilanciare i guadagni e rendere il gioco più o meno difficile. I costi di manutenzione si dividono in:

- **Costi di manutenzione ordinaria delle locomotive.**
  Ogni n turni il treno viene marcato come "da riparare" e non appena raggiunge una qualsiasi stazione viene sottoposto a manutenzione.
  Nella Sezione 3.2.4 sono state descritte nel dettaglio le modalità per cui un treno entra in fase di manutenzione.

- **Costi di manutenzione straordinaria delle locomotive.**
  Simile alla manutenzione ordinaria ma meno frequente, in quanto avviene solamente ogni x manutenzioni ordinarie. In questo caso il treno necessita di recarsi ad uno scalo per le riparazioni in quanto le normali stazioni non sono sufficienti.

- **Costi di esercizio dei treni.**
  È un costo che viene calcolato per ogni casella in cui il treno si è mosso e che viene addebitato al giocatore ogni volta che il treno raggiunge una stazione per effettuare le operazioni di carico e scarico. Rappresenta il costo più importante di tutto il modello di manutenzione, in quanto la sua granularità molto fine permette un riequilibro del rapporto spesa/guadagno molto preciso.

I costi di manutenzione ordinaria e straordinaria vengono calcolati in funzione del valore della locomotiva e della sua età secondo la formula seguente:

$$c = \mathrm{MIN}(v_0,\ v_0 * M_{value} + age\ * M_{age})$$

con

v_0 = engine.BaseCost, ovvero il valore base della locomotiva
M_{value} = rapporto tra valore locomotiva e costi di manutenzione
M_{age} = rapporto tra età della locomotiva e costi di manutenzione
age = età (in turni) della locomotiva

Un primo calcolo che è possibile fare è quello sul parametro M_{age} che indica il rapporto che intercorre tra età del treno (in turni) e incremento del costo di manutenzione. Riprendendo i calcoli fatti nella Sezione 3.2.4 dove si era visto che un treno ha in media ciclo di vita (*lifespan*) massimo pari a

$$ls_{caselle} = \frac{R(1 - 0.95^{\log_{0.95} \frac{0.05}{R}})}{0.05d}$$

caselle, e supponendo di voler fissare il momento in cui i costi di manutenzione raggiungono l'apice a quando il treno raggiunge il 75% della sua età massima, si ha allora che:

$$M_{age} = \frac{v_0 - v_0 * M_{value}}{\frac{ls_{caselle}}{engineSpeed} * 0.75}$$

con $engineSpeed =$ velocità del treno espressa in caselle per turno

Tutto il resto del modello di spesa di manutenzione di basa su due soli parametri: M_{value} ($costToMainteinanceRatio$) e M_{run} ($mainteinanceToRunningRatio$). Il primo serve a calcolare i costi base di manutenzione in rapporto al valore della locomotiva e può essere fissato arbitrariamente intorno al 5-10% di quest'ultimo. Il secondo invece rappresenta l'aspetto forse più importante di tutto il modello in quanto serve a calcorare i costi di esercizio (costi di movimento) di ogni locomotiva e va calibrato in modo da controbilanciare i guadagni, eventualmente variandolo anche in modo dinamico durante la partita in modo da garantire sempre un certo equilibrio.

3.3.6 Premi e punteggio

La vittoria di una partita si basa su un sistema a punti assegnati tramite la vincita di premi. I premi si dividono in due categorie:

- **Premi delle città:** ogni città mette a disposizione 9 premi legati al soddisfacimento delle proprie richieste, ovvero la consegna in entrata delle merci negli slot domanda e il trasporto in uscita dei passeggeri e della posta verso le città destinazione. I premi si suddividono quindi in 3 premi per le merci, 3 per i passeggeri e 3 per la posta. Poichè ogni ad ogni consegna di un premio la città sale di un livello quest'ultima potrà crescere fino ad un massimo di 9 livelli rispetto quello iniziale.
- **Premi finali:** per le partite multigiocatore vengono messi a disposizione 10 premi finali che verranno distribuiti a fine partita in base alle prestazioni dei diversi giocatori. Questi premi sono:
  - Un premio per chi ha costruito la rete ferroviaria più grossa.
  - Un premio per chi, al termine della partita, possiede più denaro.
  - Un premio per chi ha consegnato più passeggeri.
  - Un premio per chi ha consegnato più posta.
  - Sei premi per chi ha consegnato più merce per ciascuna categoria (cibo, sale, carbone, legna, mattoni e acciaio)

Il livello di avanzamento di ciascun premio è sempre assoluto e riferito alla domanda della città che lo mette a disposizione. Se per esempio una città richiede 100 unità di carbone, queste vengono conteggiate in modo cumulativo indipendentemente dal giocatore che le porta. Non appena l'ultima

unità viene consegnata il premio viene assegnato ed il suo valore, sia in punti che in denaro, viene distribuito come mostrato in Figura 3.14, ovvero:

- Il punteggio ed il valore in denaro base vengono distribuiti tra i giocatori che hanno contribuito al raggiungimento dell'obiettivo in modo proporzionale al loro contributo.
- Al giocatore che ha contribuito in maggior misura viene attribuito un bonus in punti.

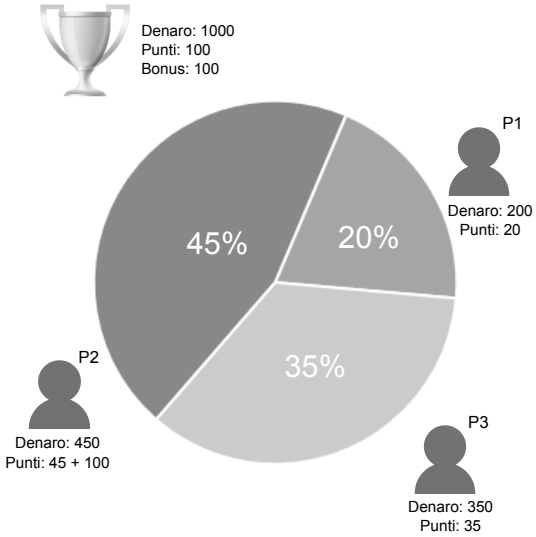


Figura 3.14: Ripartizione del valore dei premi in Iron Duke

Sia il valore del bonus che la soglia necessaria al raggiungimento di un premio vengono calcolati sulla base del livello della città , rendendone sempre più remunerativo ma allo stesso tempo più difficile il processo di crescita.

Capitolo 4

L'intelligenza artificiale in Iron Duke

In questo capitolo viene illustrato come sia stato affrontato il problema di creare un motore di intelligenza artificiale durante lo sviluppo del gioco Iron Duke. Verrà dapprima presentato un modello generico che ricalca le linee guida dei modelli già visti nel Capitolo 2, per poi entrare più nel dettaglio e spiegare come sono stati risolti i problemi di movimento, di decisione e di strategia all'interno del gioco.

4.1 Il modello di base

Il modello base utilizzato prende spunto da quello già visto nel Capitolo 2 (Figura 2.1, pagina 8) e riproposto in Figura 4.1 evidenziando però questa volta come ciascun problema sia stato affrontato e risolto nell'implementazione di Iron Duke.

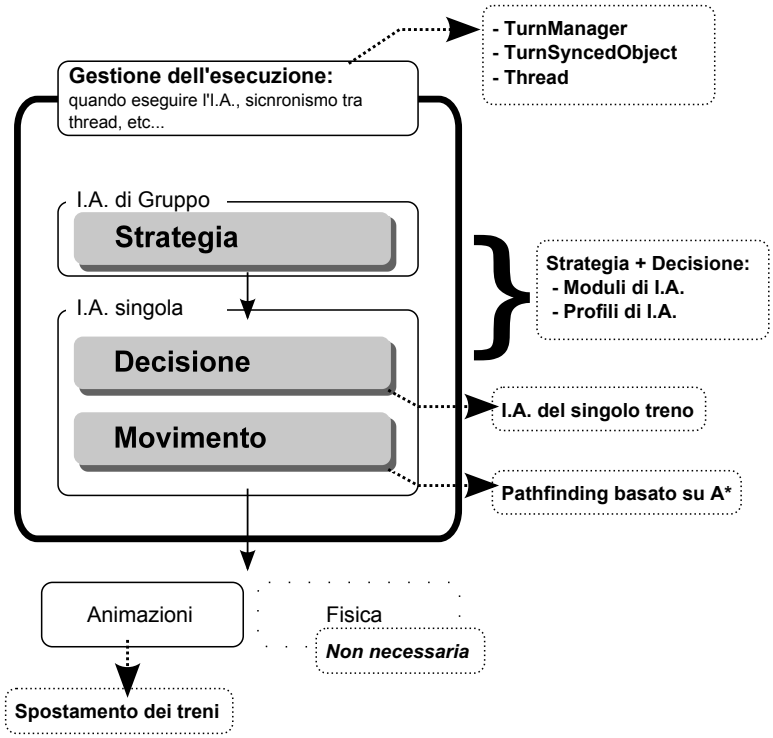


Figura 4.1: Modello base dell'intelligenza artificiale in Iron Duke

Come gran parte del resto del gioco, anche la gestione dell'esecuzione della I.A. è affidata al manager dei turni e quindi al meccanismo già descritto nella Sezione 3.1.2. Anch'essa fa quindi utilizzo degli oggetti di tipo *TurnSyncedObject* visti in precedenza, i quali garantiscono sempre coerenza e sincronismo tra le azioni dell'intelligenza artificiale ed il proseguire del gioco e limitano automaticamente le azioni dell'intelligenza artificiale ai soli momenti in cui il gioco sta effettivamente girando. (In altre parole, l'intelligenza artificiale non potrà mai eseguire azioni di gioco "scorrette" durante gli istanti in cui un giocatore umano mette il gioco in pausa per eseguire la propria mossa). Oltre a garantire un sincronismo immediato, l'utilizzo degli oggetti *TurnSyncedObject* e delle le loro funzioni *TriggerEventFunction* rende facile l'implementazione di diversi livelli di abilità dei giocatori

artificiali: risulta infatti triviale poter creare sia delle I.A. veloci in grado di eseguire molte azioni per turno, sia delle I.A. più lente che possono eseguire solo un numero limitato di calcoli.

Poichè potrebbero esserci casi in cui l'intelligenza artificiale si trovi a dover eseguire calcoli piuttosto onerosi dal punto di vista computazionale e che potrebbero interferire con il normale avanzamento del gioco, oltre al meccanismo a turni gestito dal turn manager, il motore di I.A. di Iron Duke fornisce un'interfaccia a *thread* attivabile a piacimento dal modulo di I.A. che ne faccia richiesta. Ciò permette l'esecuzione parallela dei calcoli che possono essere eseguiti contemporaneamente alle normali routine di gioco. Il funzionamento globale del modello è mostrato nel diagramma di flusso in Figura 4.2:

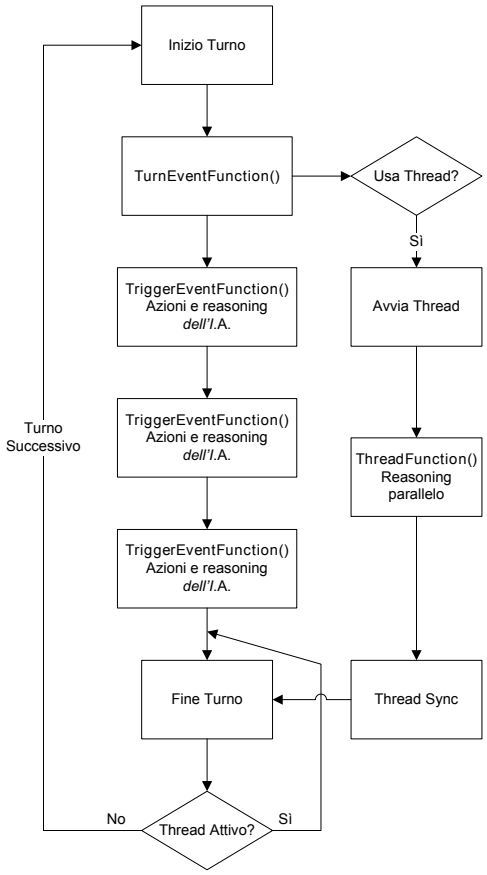


Figura 4.2: Diagramma di flusso della gestione dell'esecuzione dell'intelligenza artificiale in Iron Duke

Proseguendo l'analisi del modello di Figura 4.1, si nota come i livelli strategico-decisionali siano stati unificati in quanto la struttura del gioco non necessita di un livello decisionale dedicato. Dato che in Iron Duke non sono presenti personaggi non giocanti, non è infatti necessario implementare complessi algoritmi decisionali che operino a livello di singola unità e l'unico caso in cui ciò potrebbe essere possibile è quello della micro-gestione locale di un singolo treno, che comunque non necessita di algoritmi troppo complessi. L'implementazione completa del modello è mostrata nel diagramma UML in Figura 4.3:

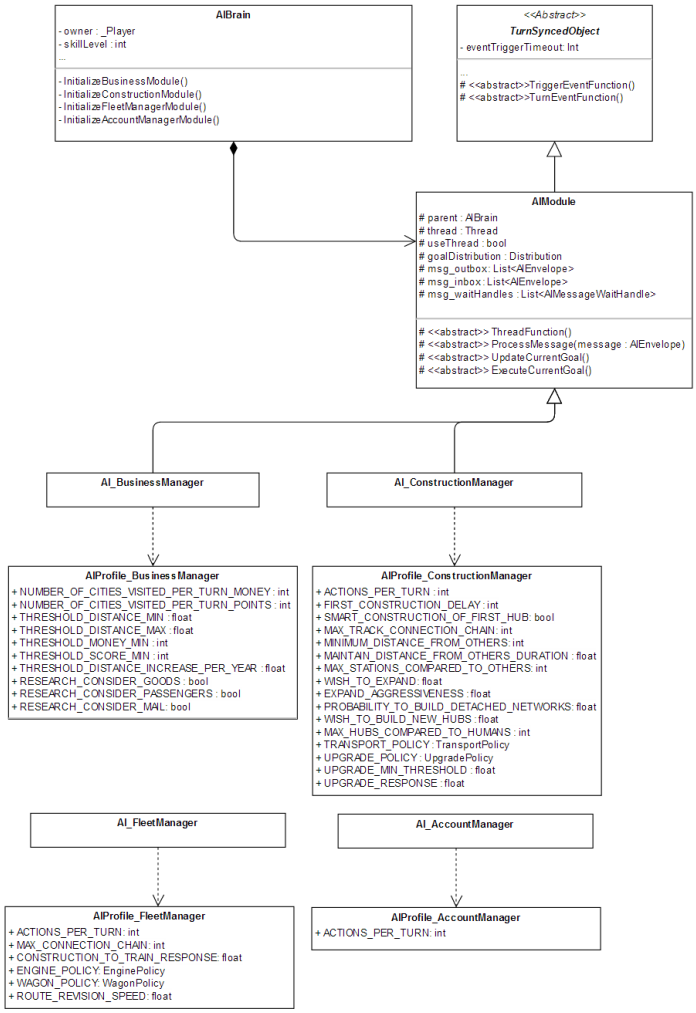


Figura 4.3: Diagramma UML del modello di IA utilizzato in Iron Duke

Il cuore del modello ruota intorno al concetto di concetto di *modulo* (*AI-Module*) . La classe *AIBrain* funge semplicemente da contenitore di questi moduli ed il suo unico scopo è quello di raggrupparli ed inizializzarli. Oltre a far ciò, la classe *AIBrain* contiene anche una serie di variabili ed oggetti condivisi tra le diverse intelligenze artificiali, nel caso ce ne sia più di una attiva. Tali variabili comprendono flag e semafori per la protezione di risorse condivise ed una *reservation table* utilizzata per gli algoritmi di pathfinding cooperativo. Come si può vedere dal diagramma, ogni modulo è un oggetto *TurnSyncedObject* che ha come scopo quello di gestire e portare avanti un'aspetto particolare del gioco. Per esempio di modulo *AI_BusinessManager* si occupa dell'analisi della situazione di gioco al fine di rintracciare le opportunità di guadagno migliori, il modulo *AI_ConstructionManager* si occupa della costruzione della rete ferroviaria, l'*AI_AccountManager* si occupa della gestione del budget e dei fondi disponibili al giocatore, e così via. Tutti i parametri che comandano gli algoritmi di reasoning di ogni modulo sono contenuti in appositi *profili* esterni . In questo modo utilizzando diversi profili è possibile ottenere comportamenti completamente differenti anche dallo stesso modulo, inoltre l'uso di profili rende molto semplice implementare diversi livelli di abilità dei giocatori comandati da un'intelligenza artificiale. Ora che è stata fornita una descrizione generale del modello utilizzato, è possibile analizzare più nel dettaglio come sono stati affrontati i problemi di movimento e di decisione all'interno del gioco.

4.2 Il problema del movimento: l'algoritmo di pathfinding in Iron Duke

Il problema del movimento in Iron Duke è stato risolto tramite l'utilizzo di tecniche di pathfinding basate sull'utilizzo dell'algoritmo A* già introdotto nel Capitolo 2. La modellizzazione della mappa tramite la struttura ad oggetti descritta nelle Sezioni 2.4.1 e 3.2.1 rende inoltre estremamente semplice la navigazione da una casella all'altra della plancia di gioco e semplifica notevolmente la ricerca di un percorso. Il codice dell'algoritmo di ricerca è riportato nella pagina seguente. Le prove pratiche in fase di sviluppo hanno dimostrato come l'algoritmo A* sia estremamente rapido ed efficiente nella risoluzione dei problemi di movimento.

```
1  public static List<HexTile> FindPath(HexTile start, HexTile goal,
       BoardScript board, PathFindingOptions options)
2  {
3    List<HexTile> closedSet = new List<HexTile>();
4    List<HexTile> openSet = new List<HexTile>();
5    openSet.Add(start);
6    Dictionary<Guid,HexTile> came_from = new Dictionary<Guid,HexTile>();
7    Dictionary<Guid,float> g_score = new Dictionary<Guid,float>();
8    Dictionary<Guid,float> f_score = new Dictionary<Guid,float>();
9          //calcolo la distanza in linea d'aria tra start e goal
10   float totalAirDistance = (float)Math.Sqrt(Math.Pow(goal.
         CenterPosition.x - start.CenterPosition.x, 2) + Math.Pow(goal.
         CenterPosition.y - start.CenterPosition.y, 2));
11   g_score.Add(start.GUID, 0);
12   f_score.Add(start.GUID, g_score[start.GUID] + totalAirDistance);
13
14   while (openSet.Count > 0)
15   {
16     //scelgo la casella nell'openset con con valore f_score più
           piccolo
17     HexTile currentHex = null;
18     float minScore = float.MaxValue;
19     foreach (HexTile entry in openSet)
20     {
21       if (f_score[entry.GUID] < minScore)
22       {
23         currentHex = entry;
24         minScore = f_score[entry.GUID];
25       }
26     }
27     //se la cella corrente è il goal ho finito e ricostruisco il
           percorso
28     if (currentHex.GUID == goal.GUID)
29       return ReconstructPath(came_from, currentHex);
30
31     //tolgo la cella corrente dall'openset
32     bool found = false;
33     int k = 0;
34     while (!found && k < openSet.Count)
35     {
36       if (openSet[k].GUID == currentHex.GUID)
37       {
38         openSet.RemoveAt(k);
39         k--;
40         found = true;
41       }
42       k++;
43     }
44     //aggiungo la cella corrente al closedSet
45     closedSet.Add(currentHex);
46     //ora analizzo ciascun vicino della cella corrente
47     HexTile[] neighbours = currentHex.getNeighbours();
48     for (int j = 0; j < neighbours.Length; j++)
49     {
50       HexTile n = neighbours[j];
51       if (n != null)
```

```
52        {
53          //devo verificare che non si trovi già nel closedSet
54          bool alreadyVisited = false;
55          k = 0;
56          while (!alreadyVisited && k < closedSet.Count)
57          {
58            if (n.GUID == closedSet[k].GUID)
59              alreadyVisited = true;
60            k++;
61          }
62          if (!alreadyVisited)
63          {
64            float tentative_g_score = g_score[currentHex.GUID] +
                CalculateMovementCost(n,currentHex, came_from,
                totalAirDistance, board, options);
65            //controllo se il vicino corrente non si trova già nell'
                openSet
66            bool alreadyInOpenset = false;
67            k = 0;
68            while (!alreadyInOpenset && k < openSet.Count)
69            {
70              if (n.GUID == openSet[k].GUID)
71              alreadyInOpenset = true;
72              k++;
73            }
74            //se non appartiene all'openSet, oppure se vi appartiene e
                il g_score corrente è minore di quello precedentemente
                calcolato...
75            if (alreadyInOpenset == false || tentative_g_score < g_score
                [n.GUID])
76            {
77              //...aggiorno i punteggi g_score e f_score per la cella
78              HexTile trash;
79              float airDistance = DistanceBetween(goal, n);
80              //se il dizionario contiene già la cella aggiorno
                  semplicemente i punteggi, altrimenti creo una nuova
                  entry
81              if (came_from.TryGetValue(n.GUID, out trash))
82              {
83                came_from[n.GUID] = currentHex;
84                g_score[n.GUID] = tentative_g_score;
85                f_score[n.GUID] = g_score[n.GUID] + airDistance;
86              }
87              else
88              {
89                came_from.Add(n.GUID, currentHex);
90                g_score.Add(n.GUID, tentative_g_score);
91                f_score.Add(n.GUID, g_score[n.GUID] + airDistance);
92              }
93              //se la cella non era già presente nell'openSet, la
                  aggiungo
94              if (!alreadyInOpenset)
95                openSet.Add(n); }}}}
96          }
97    return new List<HexTile>(); //fallimento->restituisco lista vuota
98  }
```

Il percorso finale viene restituito dalla funzione *ReconstructPath* alla riga 29 che serve a ricostruire il percorso a ritroso partendo da una casella qualsiasi e utilizzando una lista contenente tutti i riferimenti alle caselle di provenienza, in questo caso la lista *came_from* che viene appunto calcolata dall'algoritmo. L'elemento fondamentale dell'algoritmo è invece rappresentato dalla funzione *CalculateMovementCost* (linea 64), che serve a determinare il costo di ciascuna mossa sulla base di una serie di parametri contenuti nella struttura *PathFindingOptions*.
Durante le prime fasi di implementazione, è risultato subito evidente come l'algoritmo di ricerca nella sua forma "pura" restituisse risultati troppo "artificiali" e non particolarmente credibili. Inoltre nascevano sempre più situazioni nelle quali si rivelava necessario cercare il percorso secondo determinati parametri. Per questi motivi è stata introdotta una funzione aggiuntiva per il calcolo del costo che basa le proprie decisioni proprio su una serie di opzioni modificabili a piacimento a seconda della situazione. I parametri impostabili all'interno della struttura *PathFindingOptions* sono:

- **CornerMode**: questa opzione è stata introdotta con lo scopo di rendere l'intelligenza artificiale più credibile agli occhi del giocatore.

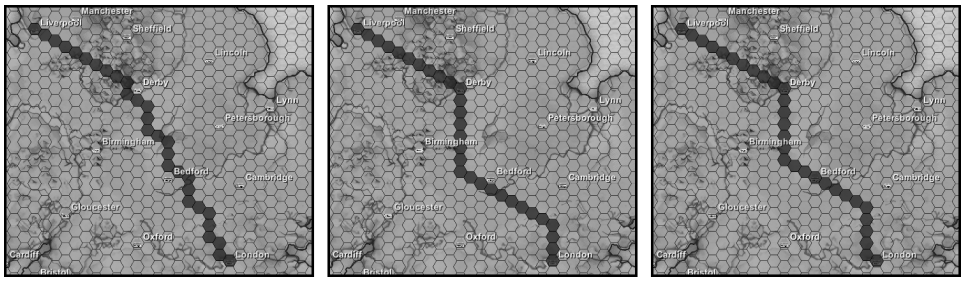


Figura 4.4: Esempi di applicazione dell'algoritmo di pathfinding in Iron Duke con diverse opzioni: algoritmo puro, ottimizzato per non creare percorsi artificiosi, ottimizzato per attraversare le città vicine.

Come già discusso nel Capitolo 2, uno degli obiettivi principali quando si sviluppa una I.A. per giochi è quello di dare al giocatore umano l'illusione di avere sempre a che fare con un'entità intelligente, in quanto la rottura di tale illusione potrebbe portare ad un'immediata perdita di immersione da parte del giocatore. Come si può vedere dalla Figura 4.4, seppur l'utilizzo dell'algoritmo nella sua forma "pura" (figura a sinistra) porti ad un risultato migliori in termini di efficienza (l'algoritmo troverà sempre il percorso più breve), un giocatore umano difficilmente traccerà una rotta che segue un pattern così "artificiale"

e macchinoso come quello mostrato. È quindi utile cercare di implementare un meccanismo che renda la ricerca magari meno ottimale ed "intelligente" ma più "stupida" e tuttavia "umana". Le opzioni che sono state implementate sono quindi:

  - **Don't care:** l'algoritmo lavora nella sua forma "pura" e cerca sempre di percorrere la distanza più breve.
  - **Avoid too much corners:** Viene applicata una penalità nel caso in cui la casella da scegliere comporti un cambio di direzione. I risultati delle prove in fase di sviluppo mostrano dei pattern di ricerca piuttosto convincenti, ed i percorsi trovati appaiono molto più naturali e meno artificiosi.

- **CitiesOptions**: sempre con lo scopo di dare all'intelligenza artificiale un comportamento più credibile, questa opzione è stata implementata per far sì che il tracciato restituito includa le città alle quali passa vicino. Poichè le città rappresentano dei punti nevralgici del gioco, ci si aspetta che un giocatore furbo crei dei collegamenti che, dove possibile e sensato, passino proprio attraverso quest'ultime piuttosto che ignorarne completamente la presenza.

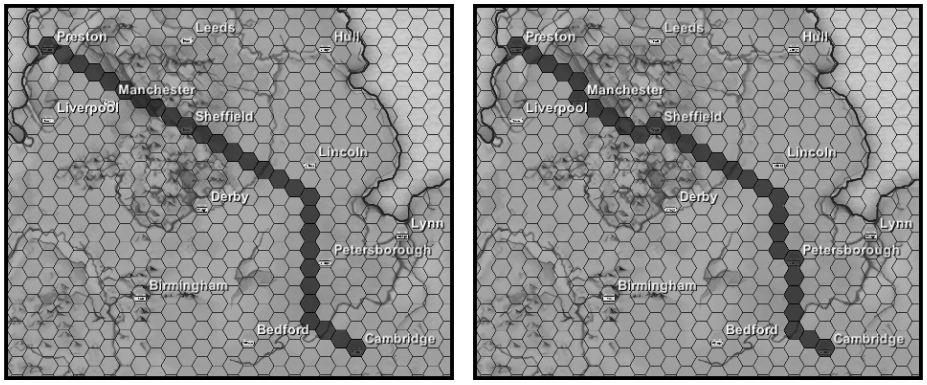


Figura 4.5: Esempi di applicazione dell'opzione passThroughNearbyCities nell'algoritmo di pathfinding in Iron Duke

La Figura 4.5 mostra un esempio dell'applicazione di questa opzione: nell'immagine a sinistra le città di Manchester e Petersborough vengono completamente ignorate nonostante sia più sensato usarle come scali intermedi del percorso, cosa che invece accade nell'immagine a destra. Le opzioni disponibili per questa opzione sono:

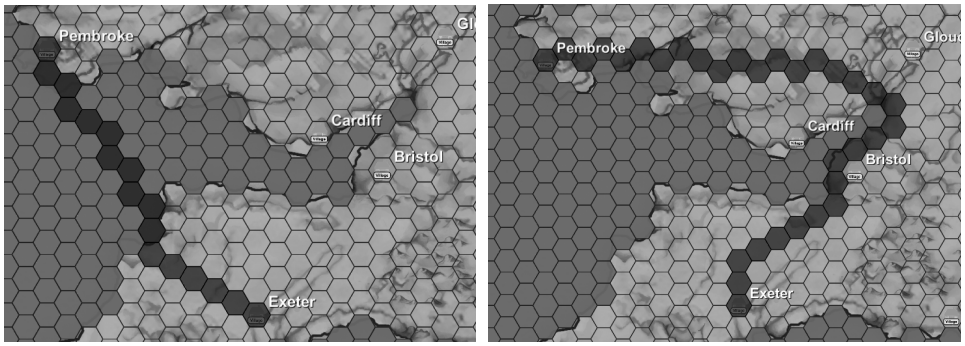
  - **Don't care:** l'algoritmo lavora nella sua forma "pura" e cerca sempre di percorrere la distanza più breve ignorando completamente la presenza delle città.
  - **Pass through nearby cities:** Viene applicato un costo *negativo* (quindi un bonus) al movimento verso una casella che contiene una città. Più il bonus è consistente, più l'algoritmo cercherà di includere le città anche più lontane dal percorso originale. È quindi necessario tarare i parametri in modo da non rendere l'algoritmo troppo aggressivo e far apparire così la ricerca ancora più artificiale agli occhi del giocatore.

- **TrackOptions**: questa opzione considera i binari costruiti sulla mappa ed è stata ideata sia con lo scopo di dare credibilità all'intelligenza artificiale, sia per una necessità strettamente pratica allo svolgimento del gioco. È possibile selezionare tra:

  - **Ignore tracks:** l'algoritmo ignora i binari e cerca semplicemente il percorso più breve.
  - **Prefer tracks:** con questa opzione attiva il percorso viene ricercato dando priorità alle caselle che contengono già dei binari. Come le opzioni avoidTooMuchCorners e passThroughNearbyCities, anche questa opzione è stata implementata con lo scopo di rendere le scelte dell'intelligenza artificiale più comprensibili da parte del giocatore. È infatti ovvio che, nel caso la AI debba costruire un nuovo tracciato, essa preferisca riutilizzare dei pezzi già costruiti fin tanto che ciò sia possibile piuttosto che tracciarne un percorso completamente nuovo che ignora le configurazioni già esistenti. Anche in questo caso le diverse prove hanno dimostrato come la costruzione automatica da parte della I.A. risulti molto più efficiente,credibile ed ordinata quando questa opzione è attiva.
  - **Only tracks:** ovviamente la logica di gioco ha necessità di poter seguire un percorso di binari già costruiti e questa opzione fà sì che il percorso venga ricercato esclusivamente tra le caselle contenenti un binario. È importante notare tuttavia che affinchè l'algoritmo sia funzionale non è sufficiente controllare semplicemente se una casella contenga o meno un binario, ma è necessario verificare che la direzione segua effettivamente il percorso, che non vengano prese direzioni vietate, che non vengano creati incroci non concessi e così via. È interessante anche notare come, a differenza di

tutti i casi precedenti, questo sia l'unico per il quale è possibile determinare dei punti di uscita forzata dall'algoritmo, senza che quest'ultimo debba necessariamente valutare tutta una serie di "cattive mosse" che lo porterebbero comunque ad un fallimento. Ciò rende l'algoritmo particolarmente efficiente e rapido.

- **CellMode**: l'ultima opzione considera lo stato logico di una casella. Come è stato descritto quando è stata introdotta la struttura della mappa utilizzata nel gioco (Sezione 3.2.1, pagina 41) non tutte le caselle sono effettivamente utilizzabili ed uno speciale file grafico viene utilizzato per marcare quelle inattive. Inoltre, poichè in una partita possono essere presenti più intelligenze artificiali attive contemporaneamente, è necessario implementare un qualche sistema di pathfinding cooperativo che memorizzi in che stato si trova ciascuna cella. Per far fronte a questi due problemi, le opzioni quindi implementate sono:
  - **Don't care**: tutte le celle vengono considerate allo stesso modo e l'algoritmo cerca semplicemente il percorso più breve.
  - **Only active**: vengono considerate valide solo le caselle al momento attive. Quelle inattive vengono considerate ostacoli (Figura 4.6).
  - **Prefer reserved for AI**: poichè ad una stessa partita possono partecipare più I.A. contemporaneamente, si è visto necessario implementare un meccanismo di pathfinding cooperativo che permettesse di evitare situazioni di conflitto. La soluzione adottata si basa sul meccanismo delle *reservation table* viste a pagina 23: ogni I.A. ha la possibilità di marcare una casella come propria per il tempo nel quale dovrà operarci e quando questa opzione è attiva il percorso viene calcolato prediligendo le caselle riservate per l'intelligenza artificiale che ha fatto richiesta del pathfinding.
  - **Avoid reserved for AI**: come nel caso precedente, ma in questo caso vengono evitate le caselle già riservate da un'altra intelligenza artificiale.



Pathfinding puro

Pathfinding che considera solo le celle attive

Figura 4.6: Ulteriori esempi di applicazione dell'algoritmo di pathfinding in Iron Duke

Gli esempi mostrati nelle Figure 4.4, 4.5 e 4.6 dimostrano come l'algoritmo A* sia estremamente utile ed efficiente nel risolvere problemi di pathfinding, e di come piccoli accorgimenti possano aumentare notevolmente l'illusione di avere a che fare con un'entità realmente "intelligente" e non con un semplice algoritmo che esegue artificiosamente le proprie istruzioni, soddisfacendo così tutti i requisiti che un buon motore di intelligenza artificiale dovrebbe avere e che sono stati descritti nel Capitolo 2.

4.3 Il problema decisionale-strategico: la struttura modulare

Come già spiegato in apertura di capitolo i livelli decisionali e quello strategico sono stati unificati in quanto non si è ritenuto necessario implementare un livello decisionale a sé. Il problema è stato affrontato seguendo una approccio di tipo *divide et impera*: il problema principale è stato quindi diviso in diversi sotto problemi che sono stati analizzati ed affrontati in modo autonomo. Per ogni sotto problema è stato quindi sviluppato un *modulo di intelligenza artificiale* in grado di affrontare al meglio la problematica in esame. Il nocciolo del motore di I.A. è costruito quindi sulla base di una struttura modulare. Anche se preso singolarmente ogni modulo contiene una serie di funzioni ed algoritmi di reasoning propri, tutti i moduli condividono la stessa struttura di base, costituita principalmente da tre elementi fondamentali:

1. Un sistema basato su *goal*
2. Una *macchina a stati*

3. Un sistema a *scambio di messaggi* con gli altri moduli.

Ognuno di questi componenti viene ora descritto più nel dettaglio.

4.3.1 Il modulo

Ogni modulo è un'oggetto di tipo *TurnSyncedObject* e come tutti gli oggetti analoghi basa il suo funzionamento sulle due funzioni *TriggerEventFunction* e *TurnFunction*. Il sincronismo con l'avanzamento di gioco è ereditato in maniera automatica dato che gli oggetti di questo tipo sono regolati dal clock offerto dal manager dei turni (Sezione 3.1.2), inoltre viene garantito che l'intelligenza artificiale non possa eseguire azioni di gioco scorrette quando questo si trova in stato di pausa. Per evitare che il susseguirsi delle azioni tra diverse I.A. segua sempre lo stesso ordine e che una I.A. possa prevalere sulle altre soltanto perchè le sue azioni vengono sempre eseguite per prime è stato implementato un meccanismo apposito nel gestore dei turni che permette un'esecuzione di tipo *out of order* delle chiamate ai moduli di intelligenza artificiale. Il codice alla base di un modulo è estremamente semplice ed è il seguente:

```
1  public abstract class AIModule : TurnSyncedObject
2  {
3    protected AIBrain parentBrain;
4
5    //Goals
6    protected Distribution goalDistribution;
7
8    //Messaggistica
9    protected List<AIEnvelope> msg_inbox;
10   protected List<AIEnvelope> msg_outbox;
11   protected List<AIMessageWaitHandle> msg_waitHandles;
12   public List<AIMessageWaitHandle> WaitHandles { get { return
         msg_waitHandles; } }
13   protected AIEnvelope msg_current;
14
15   //Threading
16   protected Thread thread;
17   protected bool useThread;
18
19   protected override void TriggerEventFunction()
20   {
21     //leggo un messaggio
22     bool read = false;
23     for (int k = 0; k < this.msg_inbox.Count && !read; k++)
24     {
25       if (!this.msg_inbox[k].timedOut)
26       {
27         if (this.msg_inbox[k].postPoneCounter <= 0)
28           {
```

```
29                this.msg_current = this.msg_inbox[k];
30                this.msg_inbox.RemoveAt(k);
31                this.ProcessMessage(msg_current);
32                this.RemoveWaitHandle(msg_current);
33                read = true;
34              }
35              else
36                this.msg_inbox[k].postPoneCounter--;
37          }
38          else
39          {
40            this.RemoveWaitHandle(this.msg_inbox[k]);
41            this.msg_inbox.RemoveAt(k);
42            k--;
43          }
44        }
45        //controllo tutti gli handle di attesa ed elimino eventuali
              messaggi andati in timeout
46        for (int k = 0; k < this.msg_waitHandles.Count; k++)
47        {
48          float elapsedTurns = ((float)TurnManager.TurnNumber +
                TurnManager.TurnProgress) - this.msg_waitHandles[k].message.
                timestamp;
49          if (elapsedTurns >= this.msg_waitHandles[k].timeout)
50          {
51            this.msg_waitHandles[k].message.timedOut = true;
52            this.msg_waitHandles.RemoveAt(k);
53            k--;
54          }
55        }
56        //gestisco i goal
57        this.UpdateCurrentGoal();
58        this.ExecuteCurrentGoal();
59                    //spedisco tutti i messaggi
60        this.DispatchAllMessages();
61      }
62
63      protected override void TurnEventFunction()
64      {
65        //threading
66        if (useThread)
67        {
68          if (this.thread != null)
69            this.thread.Join();
70          this.thread = new Thread(this.ThreadFunction);
71          this.thread.Start();
72        }
73        //spedisco tutti i messaggi
74        this.DispatchAllMessages();
75      }
76      //Funzioni di reasoning proprie da implementare nei moduli figlio
77      protected abstract void ThreadFunction();
78      protected abstract void ProcessMessage(AIEnvelope message);
79      protected abstract void UpdateCurrentGoal();
80      protected abstract void ExecuteCurrentGoal();
81    }
```

Alle righe 5 e 8 si possono cominciare a vedere gli elementi usati dai due componenti principali: il sistema dei goal e il sistema per lo scambio di messaggi, mentre alla riga 15 si possono invece vedere quelli utilizzati per l'esecuzione parallela a thread descritta all'inizio di questo capitolo nella Sezione 4.1 (Figura 4.2). Scendendo nel codice si ritrovano le ormai ben note funzioni *TriggerEventFunction* (riga 19) e *TurnEventFunction* (riga 63). Nella prima vengono eseguiti i calcoli principali del modulo e vengono ricevuti e spediti i messaggi, mentre la seconda serve principalmente al lancio e alla sincronizzazione del thread, e all'invio dei messaggi in uscita. In chiusura si trovano le quattro funzioni astratte *ThreadFunction*, *ProcessMessage*, *UpdateCurrentGoal* e *ExecuteCurrentGoal*. Va notato che anche la classe *AIModule* è in realtà astratta e non può essere quindi istanziata direttamente. I moduli veri e propri saranno quindi delle estensioni di questa classe e dovranno implementare le proprie versioni di queste funzioni, che costituiscono il vero nocciolo degli algoritmi di reasoning.
Nello specifico queste funzioni servono a:

- **ThreadFunction**: il codice all'interno di questa funzione viene eseguito *parallelamente* su di un thread separato che viene creato automaticamente e sincronizzato con ogni turno nel caso in cui il flag *useThread* sia settato a *true*.

- **ProcessMessage**: ogni modulo deve specificare quali messaggi accettare e quali azioni eseguire nel caso venga ricevuto un messaggio valido. Ogni volta che viene ricevuto un messaggio viene quindi invocata questa funzione. Il sistema dello scambio dei messaggi viene spiegato nel dettaglio nella Sezione 4.3.4 di questo capitolo.

- **UpdateCurrentGoal**: questa funzione viene chiamata ad ogni azione del modulo e serve per determinare il prossimo obiettivo da raggiungere.

- **ExecuteCurrentGoal**: all'interno di questa funzione vengono eseguite le azioni vere e proprie, che dipendono dallo stato corrente in cui si trova il modulo.

Il flusso di esecuzione delle azioni è quindi riassunto e schematizzato in Figura 4.7:

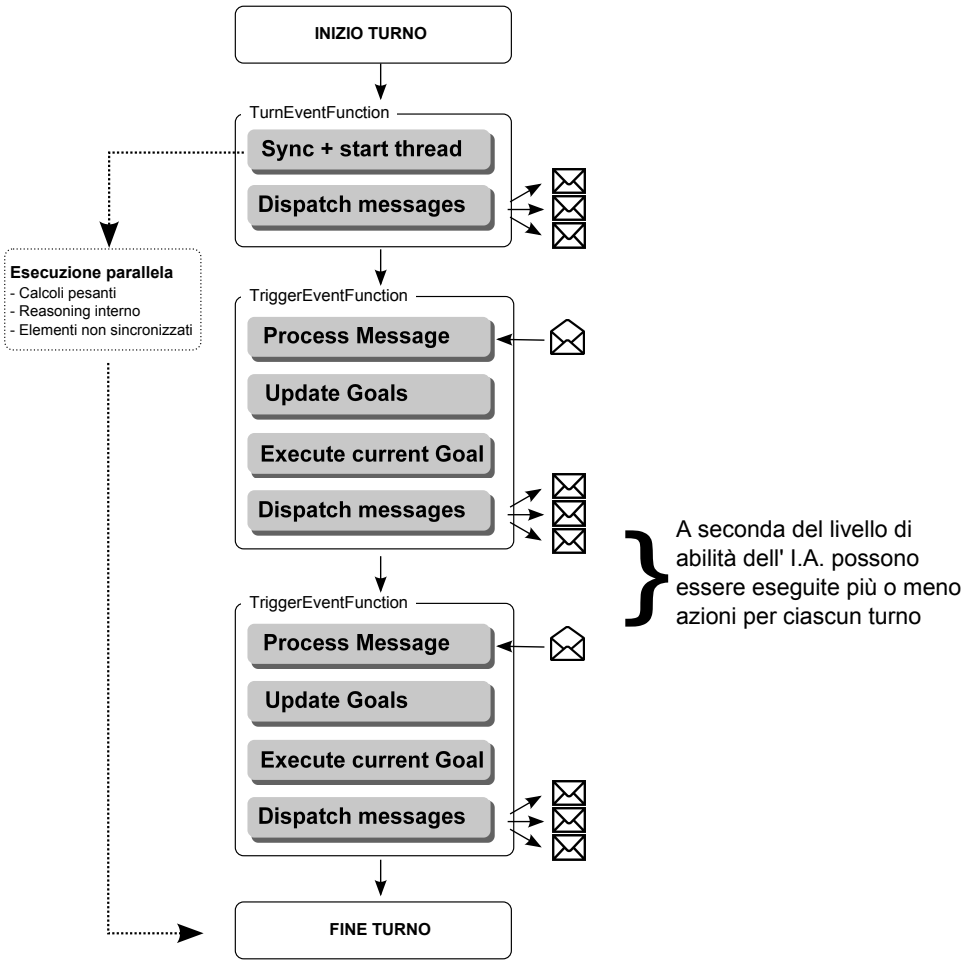


Figura 4.7: Flusso di esecuzione di un modulo di I.A.

4.3.2 Il sistema dei goal

Il sistema dei goal basa il proprio funzionamento su di una struttura matematica simile ad una *funzione di densità di probabilità discreta* descritta come segue:

dato un insieme G di goal $G = \{g_1, g_2, \ldots, g_n\}$ a cui viene associato un vettore $(p_1, \ldots, p_n)$

la probabilità che il goal corrente γ sia il goal g_i è pari a:

$$P(\gamma = g_i) = \begin{cases} p_i & \text{se } \nexists\, x \neq i \text{ t.c. } p_x > p_i \text{ , } x = 1 \dots n \\ 1 & \text{se } p_y = 0 \text{ e } \nexists\, x \neq i \text{ t.c. } p_x > p_i \text{ , } x = 1 \dots n \\ 0 & \text{altrimenti} \end{cases}$$

con $0 \leq p_i \leq 1$ e $\sum_{i=0}^{n} p_i = 1$ $y =$ Indice del goal corrente

In pratica viene selezionato il goal che al momento ha probabilità maggiore di essere estratto e si esegue un test di probabilità su di esso, in caso di successo il goal cambia, altrimenti il goal rimane quello precedente. Nel caso in cui più goal abbiano lo stesso valore p_i se ne sceglie uno a caso tra di essi e si esegue successivamente il test. Solo nel caso in cui il goal corrente abbia un valore di probabilità pari a zero si passa automaticamente a quello con probabilità maggiore. Si è scelto di considerare solo il goal con valore più alto per evitare cambi troppo repentini: fin tanto che un goal predomina sugli altri non potrà esserci alcun cambio, ma non appena un'altro goal supera di valore il goal corrente il cambio *potrebbe* avvenire. Il modello può essere visualizzato graficamente come mostrato in Figura 4.8:

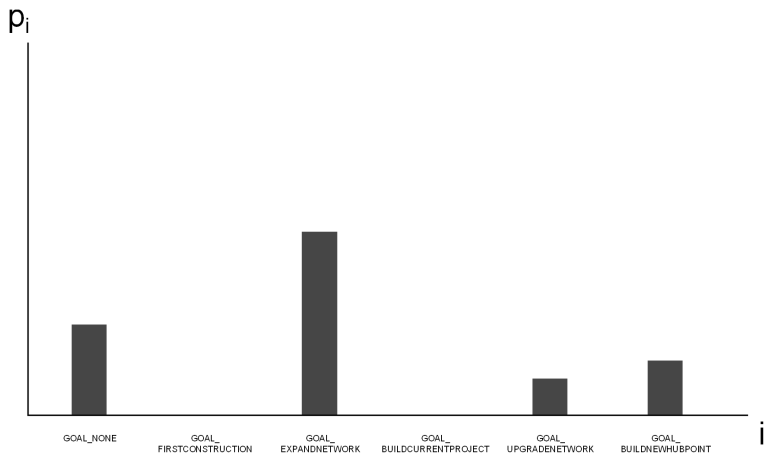


Figura 4.8: Visualizzazione grafica di una delle distribuzioni dei goal usata in Iron Duke

Poichè la somma di tutte le p_i deve essere sempre uguale a 1, quando la probabilità di un goal cresce tutte le altre diminuiscono e viceversa. Ciò garantisce sempre coerenza all'interno del modello e rende automatici alcuni

passaggi logici tra goal. Per esempio non appena un goal viene "svuotato" perchè soddisfatto, automaticamente si passerà al goal con valore immediatamente più alto. L'algoritmo che effettua il cambio tra i goal è contenuto all'interno della funzione *UpdateCurrentGoal* ed è il seguente:

```
1  //Scelgo il goal più ''desiderato'': ottengo dalla distribuzione
2  //gli indici con probabilità più alta
3  int[] maxValues = goalDistribution.GetMaxValuesIndexes();
4  int nextGoalIndex = 0;
5  bool goalSwitch = false;
6  //Se l'indice è unico uno scelgo quello, altrimenti ci sono indici
7  //a pari merito e quindi ne scelgo uno in modo random
8  if (maxValues.Length == 1)
9    nextGoalIndex = maxValues[0];
10 else
11   nextGoalIndex = maxValues[Random.Next(maxValues.Length)];
12
13 //Eseguo un test di probabilità per decidere se cambiare il goal
       corrente.
14 //Nel caso in cui il goal corrente abbia probabilità pari a zero
       effettuo il cambiamento in ogni caso.
15 if (Random.NextFloat() <= goalDistribution.GetValue(nextGoalIndex) ||
16     goalDistribution.GetValue((int)this.currentGoal) == 0)
17   goalSwitch = true;
18
19 //Effettuo l'eventuale cambio di goal
20 if (goalSwitch)
21 {
22   switch (nextGoalIndex)
23   {
24     case GOAL_NONE:
25       this.currentGoal = Goal.None;
26       break;
27     case GOAL_FIRSTCONSTRUCTION:
28       this.currentGoal = Goal.FirstConstruction;
29       break;
30     case GOAL_EXPANDNETWORK:
31       this.currentGoal = Goal.ExpandNetwork;
32       break;
33     case GOAL_BUILDCURRENTPROJECT:
34       this.currentGoal = Goal.BuildCurrentProject;
35       break;
36     case GOAL_UPGRADENETWORK:
37       this.currentGoal = Goal.UpgradeNetwork;
38       break;
39     ...
40   }
41 }
```

I valori $p_1, \ldots, p_n$ della distribuzione vengono anch'essi calcolati all'interno della funzione *UpdateCurrentGoal*: il modulo osserva attivamente i cambiamenti dell'ambiente di gioco e reagisce modificando i pesi associati a ciascun goal. Il frammento di codice seguente mostra un esempio del processo di reasoning:

```
1  protected override void UpdateCurrentGoal(){
2    switch (currentGoal)
3    {
4      case Goal.None:
5        //Se non è stato costruito ancora niente l'obiettivo sarà
6        //certamente quello di effettuare la prima costruzione
7        if (owner.Stats.building_tracks == 0 &&
8            TurnNumber >= profile.FIRST_CONSTRUCTION_DELAY &&
9            constructionProject==null)
10             goalDistribution.FillValue(GOAL_FIRSTCONSTRUCTION);
11       //Altrimenti considero l'espansione della rete corrente
12       else if (owner.Stats.building_tracks > 0 &&
13                constructionProject == null)
14       {
15         //Conto in media quante stazioni hanno i giocatori umani
16         int total = 0, count = 0, average = 0, maxBuildScore = 0;
17         for (int p = 0; p < Player.playersList.Count; p++)
18         {
19           if (Player.playersList[p].Stats.building_tracks >
                 maxBuildScore)
20             maxBuildScore = Player.playersList[p].Stats.
                   building_tracks;
21           if (Player.playersList[p].isHuman){
22             count++;
23             total += Player.playersList[p].Stats.building_stations;
24           }
25         }
26         average = total / count;
27         //La AI ha un limite massimo di costruzione riferito a quante
28         //stazioni in media hanno i giocatori umani
29         if (owner.Stats.building_stations < average + profile.
               MAX_STATIONS_COMPARED_TO_HUMANS)
30         {
31           goalDistribution.AddDelta(profile.WISH_TO_EXPAND / profile.
                 ACTIONS_PER_TURN, GOAL_EXPANDNETWORK);
32           //se la AI nota di avere meno binari degli altri, il suo
                 desiderio di costruzione aumenta ulteriormente
33           int buildDifference = owner.Stats.building_tracks -
                 maxBuildScore;
34           if (buildDifference < 0)
35             goalDistribution.AddDelta((Math.Abs(buildDifference / 10)
                   * profile.EXPAND_AGGRESSIVENESS) / profile.
                   ACTIONS_PER_TURN, GOAL_EXPANDNETWORK);
36         }
37       }
38       ... //ulteriore reasoning... (upgrade dei pezzi già esistenti,
             costruzione di nuovi hub, etc.)
39 }
```

Alle righe 10, 31 e 35 del codice si può vedere come la distribuzione venga alterata per favorire un goal rispetto agli altri. Nel primo caso l'intera distribuzione viene deviata su di un goal specifico che verrà quindi eseguito immediatamente. Negli altri due casi, invece, il peso del goal *GOAL_EXPANDNETWORK* viene aumentato gradualmente e secondo dei parametri che dipendono sia dal profilo della I.A. sia dalla situazione di gioco attuale.

4.3.3 La macchina a stati

L'esecuzione delle azioni all'interno di un modulo può essere paragonata a ciò che avviene all'interno di una macchina a stati: le transizioni tra stati vengono comandate dal meccanismo di modifica dei pesi dei goal appena descritto, mentre lo stato corrente corrisponde al goal corrente. Ogni volta che il modulo deve eseguire un'azione viene chiamata la routine *ExecuteCurrentGoal* e, a seconda dello stato corrente vengono eseguite istruzioni diverse. È importante notare che a questo punto dell'esecuzione il modulo non dovrà più in alcun modo modificare la distribuzione dei pesi dei goal, ma dovrà semplicemente limitarsi ad eseguire le azioni che lo porteranno al raggiungimento dell'obiettivo. Il completamento o meno di quest'ultimo verrà poi osservato nella successiva fase di update, all'interno della quale il modulo potrà decidere cosa fare come mossa successiva.

4.3.4 Il meccanismo a scambio di messaggi

Anche se ciascun modulo è stato ideato per svolgere le principali operazioni in modo autonomo, possono capitare situazioni in cui il compimento di un obiettivo deve vedere la partecipazione di più moduli che collaborano tra loro. Un esempio è mostrato in Figura 4.9.

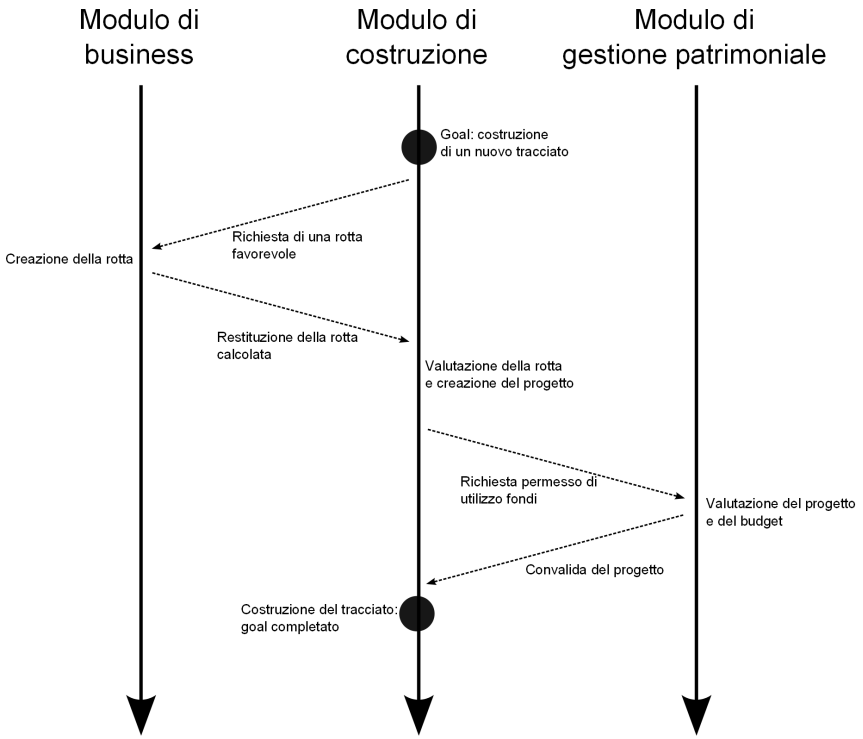


Figura 4.9: Esempio di collaborazione tra i moduli dell'intelligenza artificiale

Come prima cosa appare evidente come l'utilizzo di una struttura modulare renda molto più semplice anche la sola definizione a livello logico della sequenza azioni che l'intelligenza artificiale deve svolgere, sarebbe infatti impensabile realizzare un motore di I.A. di tipo monolitico e nel quale non esista una minima divisione dei compiti. D'altro canto, un tipo di elaborazione come quello mostrato in Figura 4.9 non giustifica di per sé l'utilizzo di una struttura a scambio di messaggi e tutto potrebbe essere risolto tramite la normale esecuzione sequenziale delle chiamate alle routine degli altri moduli (Figura 4.10).

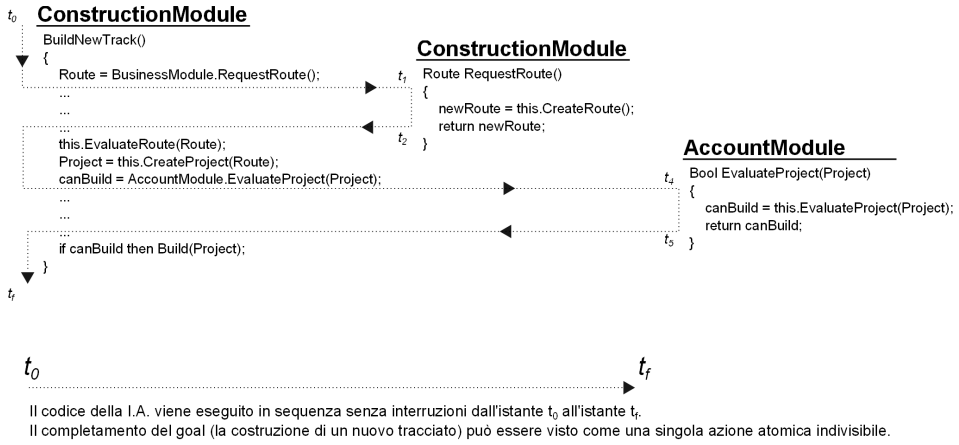


Figura 4.10: Una possibile soluzione (esecuzione sequenziale) al problema della comunicazione tra moduli

Il problema principale di un'implementazione di questo tipo risiede nella sua atomicità e quindi nell' indivisibilità delle azioni svolte. Alcuni algoritmi di ricerca utilizzati dall'intelligenza artificiale potrebbero essere piuttosto complessi ed onerosi in termini computazionali e, poichè l'esecuzione del codice della I.A. viene alternata a quella del codice del gioco di base, ciò potrebbe portare a una perdita di prestazioni e a latenze inaccettabili. Se si aggiunge anche il fatto che alla stessa partita possono partecipare più intelligenze artificiali, ognuna che esegue il proprio codice sequenzialmente e anch'esse in sequenza una dopo l'altra, è evidente di come una struttura simile potrebbe portare a delle situazioni di blocco temporaneo dell'intero avanzamento del gioco (Figura 4.11).

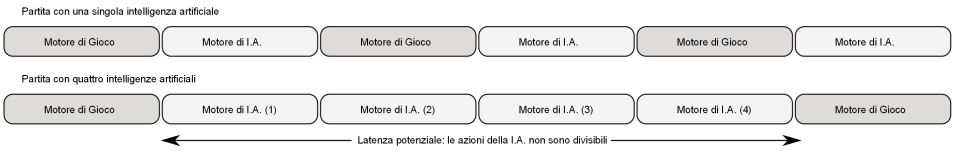


Figura 4.11: L'esecuzione atomica e sequenziale delle azioni di I.A. potrebbe causare latenza

Per ovviare a questo problema si potrebbe pensare di implementare un semplice sistema di esecuzione parallela tramite thread dei diversi motori di intelligenza artificiale che partecipano alla partita. Anche se a prima vista questo tipo di implementazione risolve i problemi di performance sopra citati poichè le azioni delle varie intelligenze artificiali vengono eseguite contemporaneamente, risulta tuttavia necessario implementare dei complessi meccanismi ad-hoc per garantire il sincronismo, sia tra le diverse I.A., sia

tra la I.A. ed il gioco, ma soprattutto tra la I.A. ed il meccanismo a turni già implementato. I problemi più evidenti sono mostrati in Figura 4.12:

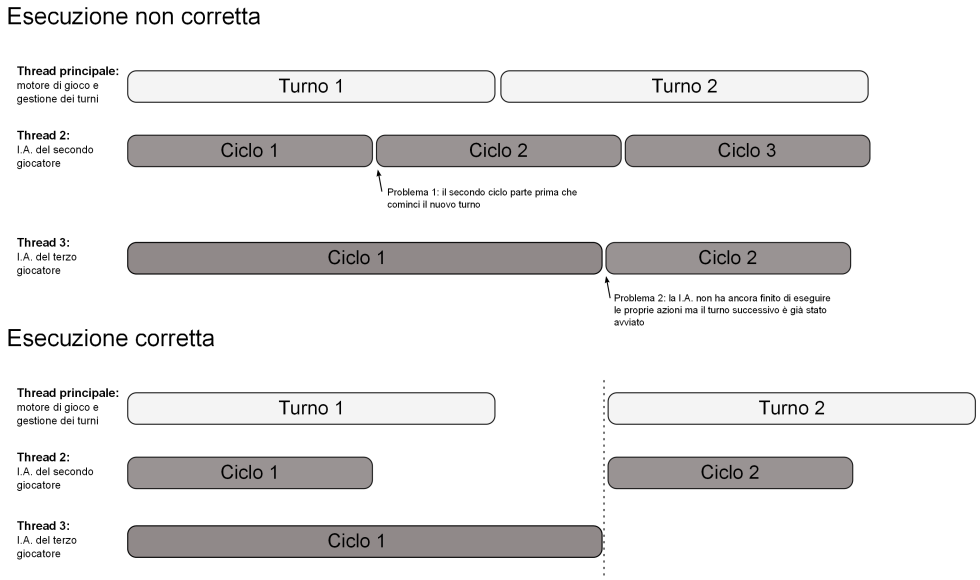


Figura 4.12: Una possibile implementazione (esecuzione parallela) del motore di intelligenza artificiale

Come si può vedere anche implementando i meccanismi di sincronismo richiesti si potrebbe facilmente incorrere in situazioni di che determinano latenza nel gioco (alla fine del primo turno il gioco deve fermarsi per aspettare che l'intelligenza artificiale abbia finito). Inoltre utilizzando una struttura come quella raffigurata l'aumento delle prestazioni è legato solamente alla capacità di calcolo parallelo offerta dal dispositivo hardware, e non dalla logica implementativa con cui é stata implementata l'intelligenza artificiale. In Iron Duke, vista la particolare struttura a turni, è stato quindi implementato un sistema a scambio di messaggi che probabilmente è la parte più interessante dell'intero motore di intelligenza artificiale. I vantaggi derivanti dall'utilizzo di tale modello sono molteplici:

- **Sincronismo automatico**: poichè anche il meccanismo di invio e ricezione dei messaggi segue il clock fornito dal manager dei turni (Vedi Sezione 3.1.2) l'intelligenza artificiale sarà sempre sincronizzata con il resto del gioco. Questo avviene "gratuitamente" senza dover implementare alcun meccanismo di controllo aggiuntivo.

- **Maggior scalabilità e modularità delle azioni**: l'intelligenza artificiale non esegue più azioni monolitiche, lineari ed indivisibili ma è in

grado di frammentare i propri calcoli e rendere il flusso di esecuzione più snello.

- **Maggiore semplicità nel progettare le azioni**: il modello prende ispirazione da come le comunicazioni avvengono nella vita reale, in particolare all'interno di un'ambiente di business. I vari moduli sono stati modellati come se fossero uffici o dipartimenti di una società, e le comunicazioni avvengono proprio sotto quest'ottica. Per esempio è naturale pensare che un progetto di costruzione debba essere valutato ed approvato prima che possa prendere il via. Il dipartimento di costruzione dovrà quindi rivolgersi agli uffici di bilancio per far valutare e convalidare il progetto, ed è proprio secondo questo il filo logico che lega le comunicazioni tra i moduli. Il lavoro del progettista dell'intelligenza artificiale risulta quindi molto più logico, ordinato e lineare.

- **Un framework comune**: tutti i moduli hanno la possibilità di utilizzare un sistema comune e condiviso, che rende il lavoro di design ancora più semplice.

- **Debug semplice** : il corretto funzionamento di un modulo può essere verificato analizzando semplicemente il modulo in questione e ignorando tutti gli altri. Poichè alcune azioni vengono lanciate in risposta ai messaggi ricevuti, è sufficiente inviare manualmente un messaggio di prova per valutare il comportamento del modulo, senza dover necessariamente ripercorrere l'intera sequenza delle azioni che porterebbe all'invio di quel messaggio. Se invece si sospetta che l'anomalia sia generata a monte dal modulo che genera la richiesta, tutto ciò che bisogna fare è analizzare il contenuto del messaggio per vedere se contiene dei parametri errati.

- **Un livello in più rispetto la semplice macchina a strati**: la ricezione e l'elaborazione dei messaggi avviene in modo del tutto separato rispetto all'esecuzione della macchina a stati. Un messaggio verrà sempre ricevuto ed elaborato indipendentemente dallo stato corrente nel quale il modulo si trova.

Tutto il sistema dello scambio dei messaggi si basa su poche semplici classi mostrate nel diagramma UML in Figura 4.13:

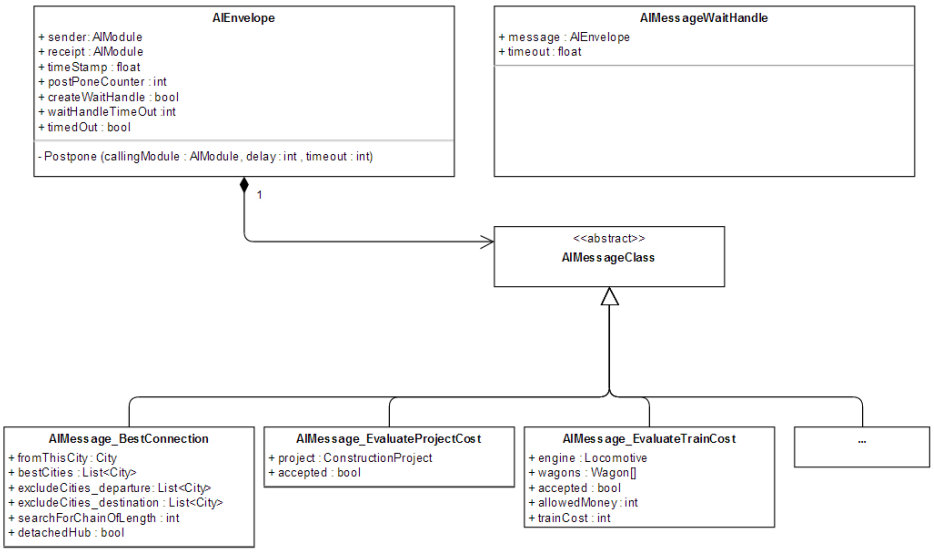


Figura 4.13: Diagramma UML delle classi dei messaggi utilizzati dal motore di intelligenza artificiale in Iron Duke

Come si può vedere l'implementazione segue volutamente un modello molto semplice ispirato allo scambio di corrispondenza nel mondo reale: una classe *Envelope* (busta) viene marcata col mittente e il destinatario del messaggio e contiene il messaggio vero e proprio, che può assumere forma diversa a seconda della situazione richiesta. Per esempio il modulo di costruzione può inviare al modulo di gestione del patrimonio un messaggio di tipo *AIMessage_EvaluateProject* inserendo al suo interno il progetto da valutare. Il destinatario si occuperà quindi di analizzare la richiesta, riempire le informazioni mancanti e rispedire al mittente il messaggio che potrà così continuare con la propria elaborazione. Ovviamente è necessario tener traccia dei messaggi inviati e gestire eventuali timeout. Viene data quindi la possibilità di creare (automaticamente) degli *handle* di attesa e posporre richieste che al momento non possono essere esaudite.

Di seguito viene mostrato come viene gestita l'operazione di costruzione di una nuova rotta già vista in Figura 4.9 dal motore di intelligenza artificiale di Iron Duke:

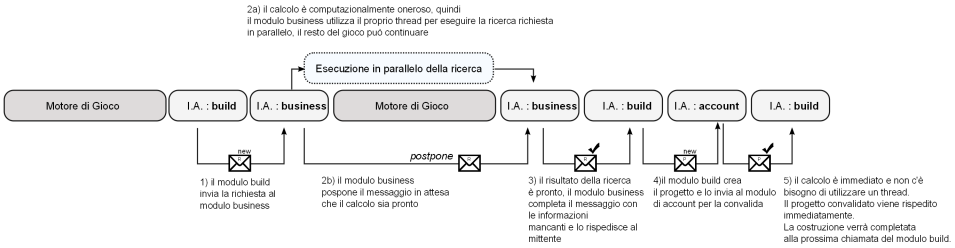


Figura 4.14: Esempio di esecuzione di una azione nel motore di intelligenza artificiale in Iron Duke

Come si può vedere viene sfruttata sia un'esecuzione di tipo sequenziale, sia un'esecuzione parallela per svolgere i calcoli più complessi. Le singole istruzioni svolte dalla I.A. sono inoltre molto più semplici e brevi rispetto a quelle viste nella soluzione proposta a pagina 86. Le prove in fase di implementazione hanno riscontrato come alcune latenze durante l'esecuzione del programma siano state completamente eliminate utilizzando questo modello[1].

[1]La maggior parte delle latenze riscontrate non erano da attribuirsi ad una reale incapacità dell'hardware di gestire il carico di lavoro di una singola I.A., ma più che altro a situazioni a "collo di bottiglia" nei casi in cui più intelligenze artificiali si trovavano ad eseguire calcoli onerosi nello stesso istante. Un'altro tipo di latenza evidente e propria di un gioco con una struttura a turni come quella di Iron Duke era determinato dal fatto che un turno dovesse attendere il termine delle azioni di I.A. iniziate nel turno precedente prima di poter iniziare.

Capitolo 5

Playtesting e simulazioni

L'obiettivo dell'intelligenza artificiale in Iron Duke è quello di riuscire a partecipare in modo autonomo ad una partita di gioco, accumulando più punti possibili ed offrendo così una sfida interessante al giocatore umano che si trova ad affrontarla. I test effettuati in questo capitolo analizzano quindi numericamente le prestazioni dell'intelligenza artificiale, e se e come quest'ultime variano in funzione di alcuni parametri principali contenuti nei diversi profili introdotti nel capitolo precedente. Lo scopo di questo primo playtest non sarà quindi quello di validare il motore di intelligenza artificiale nella sua totalità, ma piuttosto quello di verificare se le scelte fatte fin'ora si sono rivelate efficaci e quello di raccogliere informazioni e spunti utili per un suo futuro miglioramento. L'elenco completo dei parametri che verranno utilizzati nei test ora descritti è disponibile in Appendice B.

5.1 Test 1: equilibrio di gioco e risposta della I.A.

Il primo test mira a verificare che le I.A. siano equilibrate tra loro, e quindi che i risultati finali dipendono effettivamente dal loro comportamento e non dalla pura casualità degli eventi. Sono state quindi simulate 25 partite che vedevano la partecipazione simultanea di 3 giocatori comandati dalla CPU.

Ovviamente i test sono stati effettuati a parità di parametri tra i diversi giocatori. Al termine dei test si sono analizzati i punteggi ottenuti per controllare che non ci fossero grosse disparità tra i risultati. I dati ottenuti sono riassunti in Tabella 5.1:

Punteggio medio del primo classificato	5001
Punteggio medio del secondo classificato	3946
Punteggio medio del terzo classificato	3271
Differenza media tra 1° e 2°	21,04%
Differenza media tra 1° e 3°	34,60%

Tabella 5.1: Differenza media di punteggio finale tra giocatori comandati da I.A.

I risultati appaiono buoni: a parte un paio di casi isolati la differenza di punteggio tra il giocatore vincente e quello perdente tende a non superare mai il 50% e in media si attesta intorno al 35%.

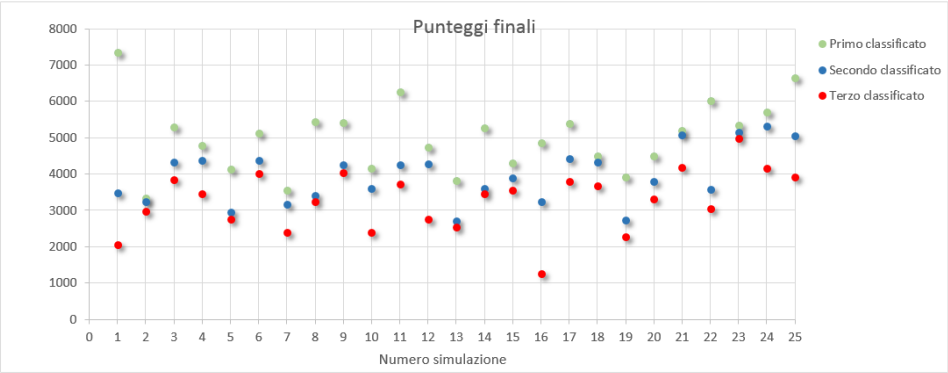


Figura 5.1: Rappresentazione grafica del distacco tra i giocatori rilevato dal primo playtest

5.1.1 Test 1.1: risposta e aggressività

Ovviamente l'intelligenza artificiale che si trova in ultima posizione dovrebbe essere in grado di reagire e cercare di diminuire le distanze. Uno dei parametri su cui si può agire per ottenere una tale reazione dovrebbe essere quello dell' *EXPAND_AGGRESSIVENESS* legato al modulo di costruzione. Questo parametro regola la voglia del giocatore di espandersi costruendo nuove tratte in relazione a due fattori:

1. La differenza nel numero di binari totali costruiti rispetto agli altri giocatori.
2. La differenza di punteggio rispetto ai giocatori in posizione più avanzata.

Il secondo criterio dovrebbe quindi garantire che all'aumentare del parametro di aggressività corrisponda una maggior risposta dei giocatori che stanno perdendo e quindi una diminuzione del distacco tra i punteggi. Sono quindi state risimulate altre 25 partite utilizzando 3 diversi parametri di aggressività: 0, 0.5 e 1, e i risultati sono riassunti nei grafici in Figura 5.2:

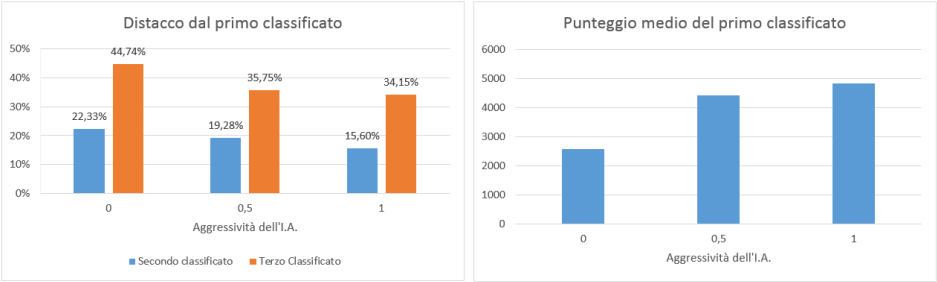


Figura 5.2: Distacco tra i giocatori e punteggio massimo al variare del parametro di aggressività

Come si può vedere il distacco decresce effettivamente all'aumentare del parametro ed anche il punteggio massimo ottenuto varia sensibilmente. Tuttavia è possibile anche notare che la variazione sembra essere piuttosto accentuata nell'intervallo 0 - 0.5, mentre appare contenuta nell'intervallo 0.5 - 1. Questo fatto può essere probabilmente spiegato analizzando i dati sulla costruzione mostrati nel grafico in Figura 5.3: si vede infatti come il punteggio ottenibile sia certamente legato alla grandezza della rete costruita, tuttavia si nota come anche la costruzione raggiunga già un punto di saturazione al parametro di aggressività 0.5, in pratica a quel punto praticamente tutte le città nella mappa sono state collegate e l'I.A. non può espandersi ulteriormente.

I risultati ottenuti da questo test portano quindi alle seguenti conclusioni:

- La variazione del parametro EXPAND_AGGRESSIVENESS è effettivamente in grado di modificare la risposta e l'aggressività dell'intelligenza artificiale.
- L'aggressività nel costruire incide effettivamente sul risultato finale della partita, tuttavia è facile raggiungere un punto di saturazione

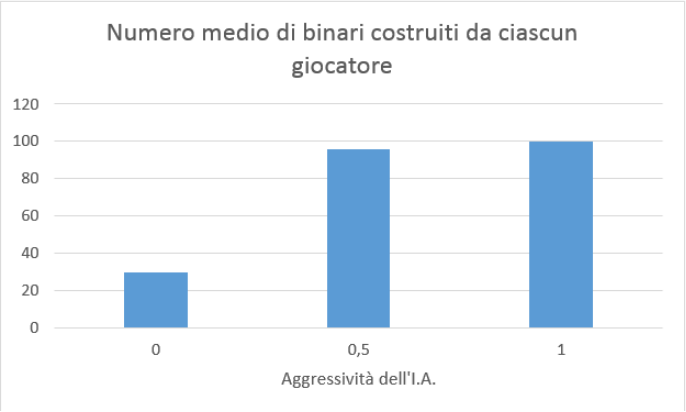


Figura 5.3: Differenza di costruzione al variare del parametro di aggressività

dove ciò non è più sufficiente a garantire una buona competizione. Si devono quindi cercare nuove strategie che permettano all'I.A. di essere ulteriormente competitiva una volta completata la costruzione dell'intera rete.

- Il parametro EXPAND_AGGRESSIVENESS dovrebbe essere regolato nel range 0 - 0.5 poichè valori superiori non sembrano apportare differenze significative.

5.2 Test 2: efficienza degli algoritmi di costruzione

Questi test sono mirati a valutare sia l'efficienza delle scelte attuate in fase di costruzione, sia il legame che può intercorrere tra tali scelte ed il punteggio finale ottenibile.

5.2.1 Test 2.1: valutazione del punto iniziale di costruzione

Gli algoritmi all'interno del modulo di costruzione sono in grado di decidere in che punto iniziare la costruzione della propria rete secondo due politiche (parametro *SMART_CONSTRUCTION_OF_FIRST_HUB*):

1. In modo del tutto casuale, selezionando una delle 38 città disponibili.
2. Dall'analisi della mappa di gioco utilizzando la tecnica del map flooding .

Si ricorda al lettore che, poichè i treni possono essere inseriti solo nei punti contenenti degli hub e che la rete può essere ampliata solo partendo da punti già toccati (a meno di costruire un nuovo hub distaccato), la scelta del

punto iniziale di costruzione può essere molto importante al fine di ottenere il maggior numero di punti. La zona potenzialmente più vantaggiosa viene calcolata utilizzando una tecnica di map flooding come quella descritta nella Sezione 2.3.3 utilizzando la totalità dei soldi e dei punti potenzialmente guadagnabili per ogni città come fattore di influenza da espandere alle caselle adiacenti. Il primo test considera quindi il punteggio finale ottenuto da un singolo giocatore comandato da I.A. in 100 simulazioni della durata di 50 anni di gioco ciascuna, utilizzando prima una scelta casuale della zona di partenza e in un secondo momento la scelta "ponderata". I risultati ottenuti sono i seguenti:

	Scelta casuale	Scelta calcolata
Punteggio (media)	7022	6671
Punteggio (mediana)	7356	6892

Tabella 5.2: Differenza media di punteggio finale al variare della politica di scelta del punto iniziale di costruzione.

I risultati ottenuti appaiono indubbiamente contraddittori, il punteggio medio ottenuto utilizzando il punto di partenza calcolato risulta infatti inferiore. Questo potrebbe aprire nuove riflessioni e la causa potrebbe essere attribuibile a:

- Una troppo rapida espansione da parte del giocatore, che lo porta troppo presto a "disperdersi" dalla zona di partenza indicata.
- Al fatto che, effettivamente, la costruzione partiva sempre dagli stessi punti che, per via della loro posizione, potevano limitare le scelte di costruzione successive e quindi condannare l'I.A. ad ottenere delle configurazioni finali sempre simili e che impedivano di superare una certa soglia di punteggio.
  La scelta casuale garantiva invece più libertà e permetteva di costruire reti che non sarebbe stato possibile costruire cominciando sempre dallo stesso punto.
- Un'effettiva inefficienza dell'algoritmo di sfruttare la zona potenzialmente favorevole, e quindi uno spunto per migliorare il motore di I.A. non tanto nella sezione dedicata alla costruzione, ma in quella dedicata alla gestione delle rotte dei treni.

I valori ottenuti non sono tuttavia da considerare un fallimento dell'algoritmo, va infatti ricordato che nel test precedente l'intelligenza artificiale non

aveva avversari con cui competere, e potrebbe quindi non aver senso far scegliere un punto particolare al giocatore se questo non ha avversari che gli impediscono di sfruttare quello stesso punto in un secondo momento. Per questa ragione è stato effettuato un secondo test introducendo un avversario e valutando se la scelta del primo punto di costruzione potesse o meno rivelarsi un fattore di vantaggio. Sono state quindi effettuate altre 3 serie di test da 100 simulazioni ciascuna e con attiva l'opzione di scelta non casuale del primo punto di costruzione. Alla fine delle simulazioni è stata calcolata la percentuale di vittorie del giocatore che ha eseguito la prima mossa. I risultati sono riassunti in Tabella 5.3:

	Prima serie di test	Prima serie di test	Terza serie di test
% di partite vinte dal giocatore che ha effettuato la prima mossa	**57%**	**59%**	**60%**
% di partite vinte dal giocatore che ha effettuato la seconda mossa	43%	41%	40%

Tabella 5.3: Vantaggio del giocatore che effettua la prima mossa utilizzando una politica intelligente di scelta della posizione iniziale di costruzione.

Dai dati sembrerebbe quindi che effettivamente la scelta della prima posiziona possa fornire un leggero vantaggio sugli altri giocatori, e allo stesso tempo che cominciare da una posizione sfavorevole non pregiudica completamente l'esito della partita. I grafici in Figura 5.4 mostra come effettivamente l'algoritmo condizioni l'I.A. a scegliere bene o male sempre le stesse zone di partenza, tuttavia non c'è forte evidenza che l'ordine di scelta o la scelta vera e propria stabilisca a priori l'esito della partita.

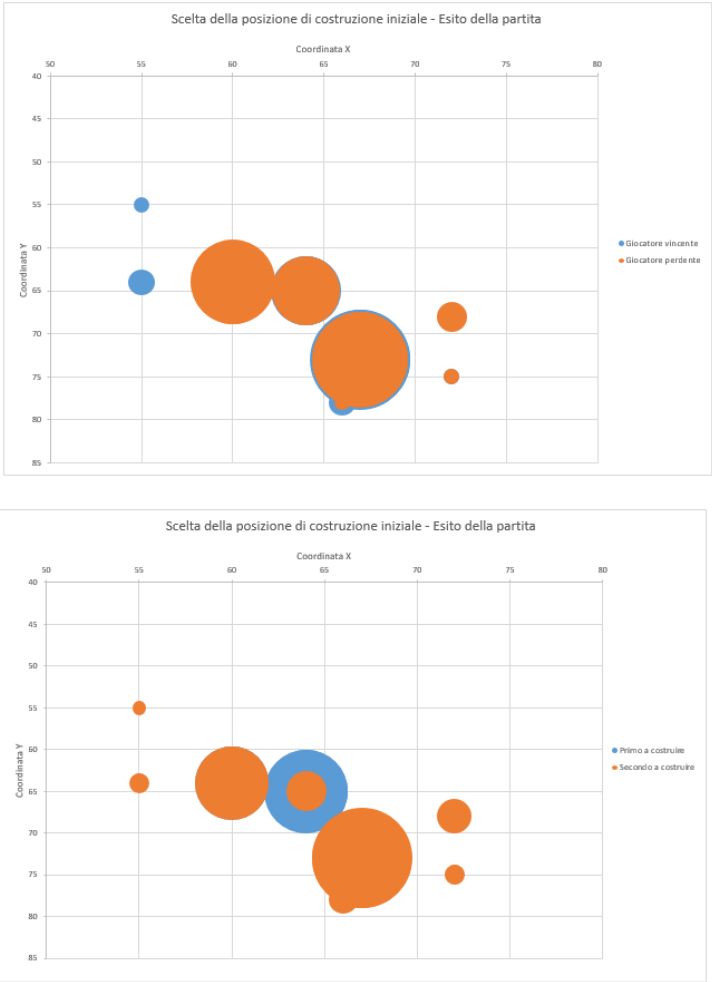


Figura 5.4: Rappresentazione grafica della zona iniziale di costruzione da parte del modulo di intelligenza artificiale e relativo risultato finale.

5.2.2 Test 2.2: legame tra la scelta posizione iniziale e la profondità della ricerca

Poichè dai risultati dei test precedenti non è ancora ben chiaro se la scelta della posizione iniziale rappresenti o meno un effettivo vantaggio per l'intelligenza artificiale, sono stati effettuati altri test considerando questa volta il parametro che determina la profondità della ricerca di una nuova rotta (parametro *MAX_TRACK_CONNECTION_CHAIN*). Tutti i test eseguiti precedentemente sono stati eseguiti con il valore di questo parametro pari a 2, ovvero i collegamenti cercati erano tutti di tipo A-B quindi semplice-

mente da una città ad un'altra. I nuovi test utilizzano un nuovo valore pari a 4, quindi l'algoritmo cercherà dei collegamenti che possono considerare *fino* 4 città contemporaneamente creando dove possibile quindi delle catene di tipo A-B-C-D. La ragione di questo nuovo test è semplice: se è vero che la scelta della posizione iniziale può influire sull'esito finale, occupando già da subito un maggior numero di città vicine alla zona consigliata dovrebbe apportare un ulteriore maggior vantaggio. I dati raccolti tuttavia non evidenziano una considerevole differenza tra il caso precedente (catena lunga 2) ed il nuovo caso (catena lunga 4), seppur sembrino confermare che in effetti la scelta della posizione dia un leggero vantaggio al giocatore che esegue la prima mossa.

	Prima serie di test	Prima serie di test	Terza serie di test
MAX_CONNECTION_CHAIN = 2			
% di partite vinte dal giocatore che ha effettuato la prima mossa	**57%**	**59%**	**60%**
% di partite vinte dal giocatore che ha effettuato la seconda mossa	43%	41%	40%
MAX_CONNECTION_CHAIN = 4			
% di partite vinte dal giocatore che ha effettuato la prima mossa	**58%**	**58%**	**53%**
% di partite vinte dal giocatore che ha effettuato la seconda mossa	42%	42%	47%

Tabella 5.4: Differenza del vantaggio del primo giocatore al variare del parametro MAX_CONNECTION_CHAIN

5.2.3 Test 2.3: costruzione di segmenti distaccati

Come già detto in precedenza, i giocatori possono espandere la propria rete solo partendo da punti in cui essa è già stata costruita. Tuttavia esiste anche la possibilità di poter costruire nuovi hub ferroviari distaccati che permettono di costruire una nuova rete che non interseca quella già costruita. Lo scopo di questo test è quindi quello di verificare se e come la costruzione di reti distaccate rispetto a reti prettamente collegate con il primo punto di costruzione influisca sul risultato finale. Sono state eseguite 3 serie di test da 50 simulazioni ciascuna incrociando i parametri *SMART_CONSTRUCTION_OF_FIRST_HUB* (che determina il punto iniziale di costruzione casuale e non) e *PROBABILITY_TO_BUILD_DETACHED_NETWORKS* che determina la probabilità di costruire una rete distaccata ogni volta che l'I.A. cerca di espandersi. I risultati sono riportati in Tabella 5.5, che elenca le mediane dei punteggi ottenuti.

Prima serie di test					
	Probabilità hub distaccati				
	0.0	**0.25**	**0.50**	**0.75**	**1.00**
Costruzione iniziale casuale	7149	7233	6429	6783	7939
Costruzione iniziale smart	6748	7413	7623	7533	7602
Seconda serie di test					
	Probabilità hub distaccati				
	0.0	**0.25**	**0.50**	**0.75**	**1.00**
Costruzione iniziale casuale	7320	6188	6133	7494	7772
Costruzione iniziale smart	7313	7465	7648	7619	8106
Terza serie di test					
	Probabilità hub distaccati				
	0.0	**0.25**	**0.50**	**0.75**	**1.00**
Costruzione iniziale casuale	7668	6398	6439	7008	8320
Costruzione iniziale smart	6662	7154	7513	8099	7994

Tabella 5.5: Risultati ottenuti incrociando i parametri di costruzione iniziale e probabilità di costruire reti distaccate.

I grafici che riassumono tali risultati sono riportati in Figura 5.5 e mostrano dei comportamenti sicuramente interessanti: da una parte si nota come la costruzione di reti distaccate possa effettivamente garantire dei punteggi finali più alti, dall'altra la particolarità forse più interessante che è possibile apprezzare è che nel caso di una politica di scelta della posizione iniziale casuale l'aumento di punteggio si ottiene solo quando la probabilità di costruire reti distaccate è alta, mentre si ha un'evidente contrazione per valori

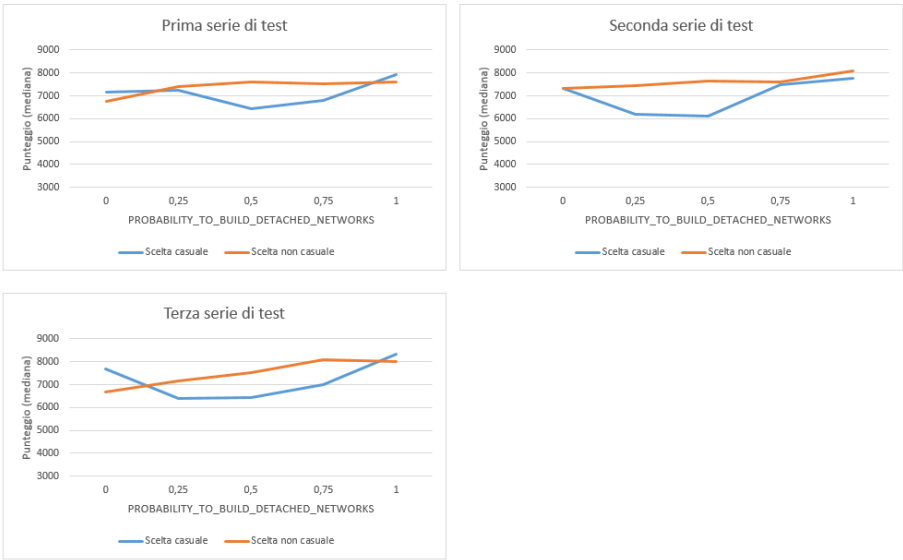


Figura 5.5: Grafici dei dati ottenuti incrociando i parametri di costruzione iniziale e probabilità di costruire reti distaccate.

intermedi. Questi risultati si riveleranno molto utili nella fase finale della definizione dei profili.

5.3 Test 3: politiche di costruzione dei convogli

Come visto nel Capitolo 3 nel gioco sono presenti diverse tipologie di locomotive, ognuna caratterizzata da una diversa velocità di punta ed una diversa capacità di carico. È quindi utile analizzare come la scelta del tipo di locomotiva e del numero di vagoni possa influire sul punteggio finale. Il modulo di intelligenza artificiale dedicato alla gestione della flotta prevede 5 politiche di scelta della locomotiva e 3 politiche di scelta del numero di vagoni[1], esse sono:
Per la scelta delle locomotive:

- **Random**: viene scelta una locomotiva a caso tra le ultime 3 attualmente disponibili. Per ultime 3 si intende le tre al livello tecnologico migliore attualmente sbloccate.
- **MostExpensive**: viene sempre scelta la locomotiva al momento più costosa.

[1]Tutti i test fin'ora eseguiti utilizzavano le stesse politiche: MostExpensive per le locomotive e MaxSpeed per i vagoni

- **BetterLoad**: viene scelta la locomotiva che garantisce il maggior numero di vagoni possibili.
- **BetterSpeed**: viene scelta la locomotiva più veloce tra quelle attualmente disponibili.
- **Balanced**: viene scelta la miglior locomotiva che garantisce equilibrio tra velocità e capacità di trasporto.

Per la scelta del numero di vagoni:

- **Random**: viene selezionato un numero di vagoni casuali che comunque garantisce di non mandare il convoglio in situazioni di sovraccarico.
- **MaxLoad**: viene scelto il maggior numero di vagoni supportati dalla locomotiva corrente.
- **MaxSpeed**: viene scelto un numero di vagoni tali per cui la velocità di punta della locomotiva corrente non venga compromessa.

Sono state quindi eseguite 100 simulazioni della durata di 50 anni di gioco ciascuna per ognuna delle 15 combinazioni possibili, i risultati sono riportati in Tabella 5.6 e riassunti nei grafici in Figura 5.6.

	WagonPolicy		
EnginePolicy	Random	MaxSpeed	MaxLoad
Random	6595	7063	5528
MostExpensive	6567	**7188**	5438
BetterSpeed	6409	6711	4767
BetterLoad	7076	7081	6107
Balanced	6244	6352	5750

Tabella 5.6: Confronto tra le varie politiche di costruzione dei convogli adottate dalla I.A.

I risultati ottenuti sono molto interessanti, da un lato (Figura 5.6) si vede come una maggiore velocità degli spostamenti garantisca un più rapido accumulo di punti. D'altro canto (Figura 5.7) è evidente come esista un grosso tradeoff tra guadagno di punti e guadagno di denaro, e ciò potrebbe rendere ancora più interessanti le scelte strategiche in fase di gioco. Probabilmente sarà tuttavia necessario calibrare i costi in modo da rendere meno marcata questa differenza.

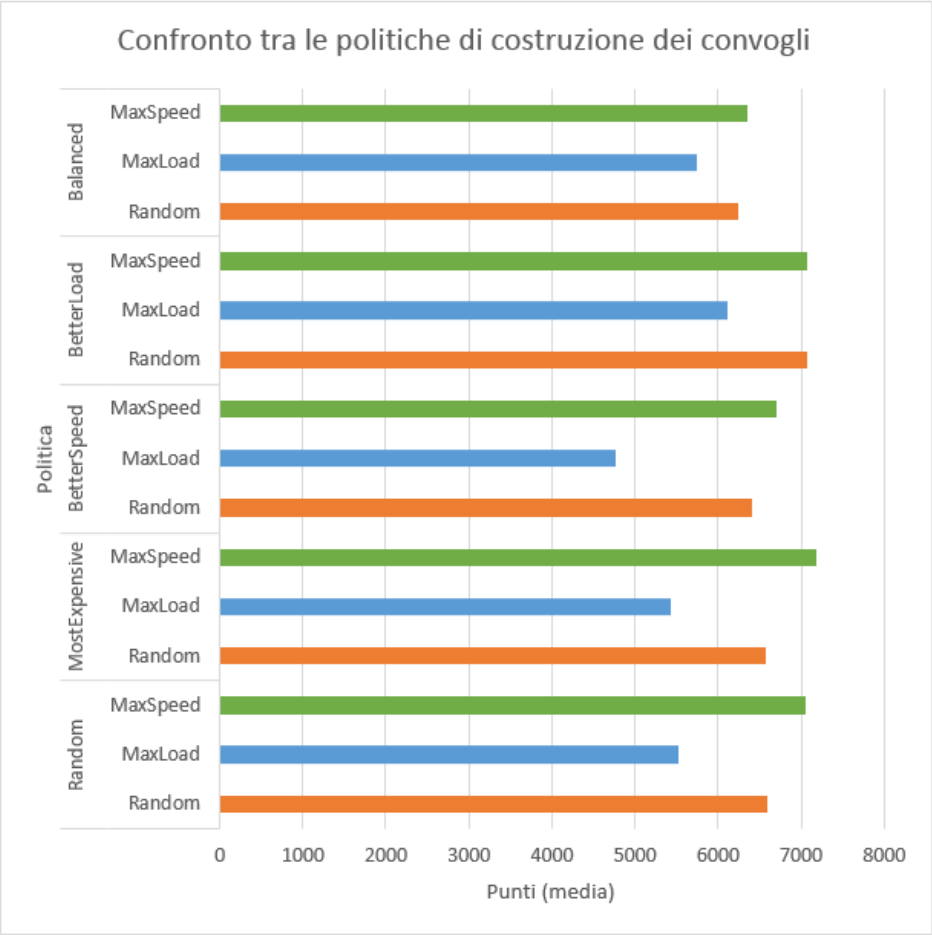


Figura 5.6: Confronto tra le varie politiche di costruzione dei convogli adottate dalla I.A.

I dati rilevati si riveleranno molto utili per due aspetti del gioco:

1. Come visto nella Sezione 3.3.5 uno dei parametri su cui è possibile agire per equilibrare in modo fine il rapporto spesa-guadagno è quello dei costi di esercizio dei treni. Dalle simulazioni è possibile trarre spunto per capire in che modo agire su questo parametro, per esempio si nota come potrebbe essere utile diminuire i costi di esercizio delle locomotive più veloci ed aumentare quello delle locomotive con più capacità di carico in modo da diminuire il dislivello tra le due.

2. Grazie alla conoscenza acquisita sarà possibile implementare degli algoritmi mirati per la scelta ottimale della configurazione dei convogli anche a seconda della situazione corrente del gioco. Per esempio le

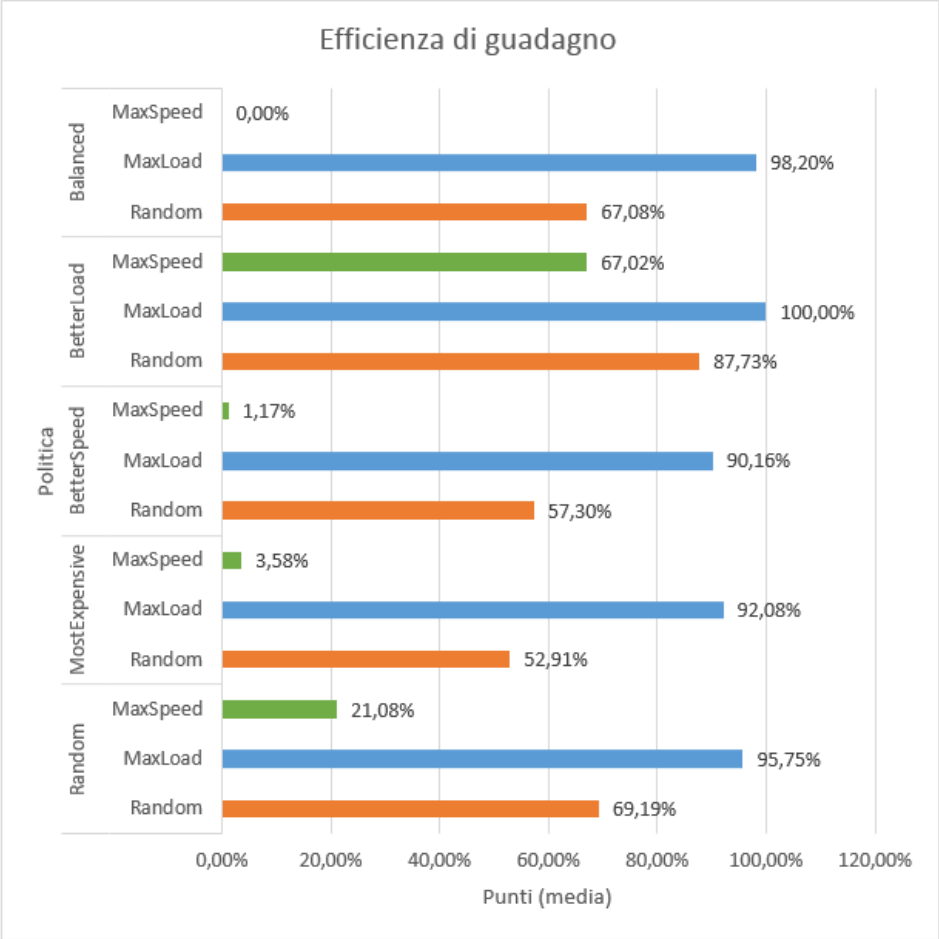


Figura 5.7: Efficienza di guadagno in relazione alle politiche di costruzione dei convogli adottate dalla I.A.

politiche di scelta potranno variare in modo dinamico a seconda di ciò che si vorrà far ottenere dal giocatore artificiale: se l'obiettivo corrente sarà quello di accumulare soldi si potrà decidere di adottare politiche di tipo *load*, mentre un giocatore che si trova in ultima posizione e ha bisogno di recuperare punti verterà verso politiche di tipo *speed*.

5.4 Test 4: livelli di abilità dell'I.A.

Il gioco prevede 4 diversi livelli di abilità dei giocatori comandati da intelligenza artificiale e i dati fin'ora raccolti serviranno ad iniziare a delineare 4 diversi profili che dovrebbero garantire un livello di sfida via via crescente per il giocatore umano che sfida la CPU. Ha quindi senso provare ad effettuare

delle prove per verificare se l'utilizzo di diversi parametri può effettivamente portare all'obiettivo finale di avere diversi livelli di comportamento da parte dell'intelligenza artificiale.

5.4.1 Test 4.1: partita a due giocatori

In questo scenario il test prevede la partecipazione di due giocatori comandati da intelligenza artificiale, una a livello di abilità minimo e l'altro a livello di abilità massimo. I parametri utilizzati (spiegati in Appendice B) sono quindi i seguenti:

Business Module
Giocatore 1 - livello abilità 1
NUMBER_OF_CITIES_VISITED_PER_TURN_MONEY = 8;
NUMBER_OF_CITIES_VISITED_PER_TURN_POINTS = 8;
THRESHOLD_DISTANCE_MAX = 0.10f;
THRESHOLD_DISTANCE_INCREASE_PER_YEAR = 0.03f;
Giocatore 2 - livello abilità 4
NUMBER_OF_CITIES_VISITED_PER_TURN_MONEY = 20;
NUMBER_OF_CITIES_VISITED_PER_TURN_POINTS = 20;
THRESHOLD_DISTANCE_MAX = 0.15f;
THRESHOLD_DISTANCE_INCREASE_PER_YEAR = 0.03f;

Construction Module
Giocatore 1 - livello abilità 1
ACTIONS_PER_TURN = 4;
FIRST_CONSTRUCTION_DELAY = 0;
SMART_CONSTRUCTION_OF_FIRST_HUB = false;
MAX_TRACK_CONNECTION_CHAIN = 2;
MINIMUM_DISTANCE_FROM_OTHERS = 6;
MAINTAIN_DISTANCE_FROM_OTHERS_DURATION = 0.25f;
WISH_TO_EXPAND = 0.01f;
EXPAND_AGGRESSIVENESS = 0.0f;
MAX_STATIONS_COMPARED_TO_OTHERS = 200;
WISH_TO_BUILD_NEW_HUBS = 0.1f;
MAX_HUBS_COMPARED_TO_HUMANS = 0;
PROBABILITY_TO_BUILD_DETACHED_NETWORKS = 0.5f;
UPGRADE_POLICY = AIProfile_ConstructionManager.UpgradePolicy.Gradual;
UPGRADE_MINTHRESHOLD = 0.5f;
UPGRADE_RESPONSE = 0.15f;
Giocatore 2 - livello abilità 4
ACTIONS_PER_TURN = 6;
FIRST_CONSTRUCTION_DELAY = 1;
SMART_CONSTRUCTION_OF_FIRST_HUB = true;
MAX_TRACK_CONNECTION_CHAIN = 4;
MINIMUM_DISTANCE_FROM_OTHERS = 6;
MAINTAIN_DISTANCE_FROM_OTHERS_DURATION = 0.10f;
WISH_TO_EXPAND = 0.04f;
EXPAND_AGGRESSIVENESS = 0.3f;
WISH_TO_BUILD_NEW_HUBS = 0.1f;
MAX_HUBS_COMPARED_TO_HUMANS = 4;
MAX_STATIONS_COMPARED_TO_OTHERS = 10;
PROBABILITY_TO_BUILD_DETACHED_NETWORKS = 0.75f;
UPGRADE_POLICY = AIProfile_ConstructionManager.UpgradePolicy.Smart;
UPGRADE_MINTHRESHOLD = 0.5f;
UPGRADE_RESPONSE = 0.15f;

Fleet Management Module
Giocatore 1 - livello abilità 1
ACTIONS_PER_TURN = 2;
MAX_CONNECTION_CHAIN = 2;
ENGINEPOLICY = AIProfile_FleetManager.EnginePolicy.Random;
WAGONPOLICY = AIProfile_FleetManager.WagonPolicy.MaxLoad;
CONSTRUCTION_TO_TRAIN_RESPONSE = 0.6f;
ROUTE_REVISION_SPEED = 0.05f;

Giocatore 2 - livello abilità 4
ACTIONS_PER_TURN = 4;
MAX_CONNECTION_CHAIN = 2;

```
ENGINEPOLICY = AIProfile_FleetManager.EnginePolicy.BetterLoad;
WAGONPOLICY = AIProfile_FleetManager.WagonPolicy.Random;
CONSTRUCTION_TO_TRAIN_RESPONSE = 0.8f;
ROUTE_REVISION_SPEED = 0.25f;
```

I risultati ottenuti sono riassunti in Figura 5.8. Il fatto che il giocatore con abilità massima non abbia vinto la totalità delle partite è probabilmente imputabile a bug e situazioni di blocco all'interno della partita che impedivano l'accumulo di punti (in alcune simulazioni il secondo giocatore non è stato in grado di superare le poche centinaia di punti). I primi dati appaiono comunque convincenti e fanno pensare che effettivamente la variazione dei parametri possa portare ad avere livelli di difficoltà differenziati. Come ulteriore nota si può aggiungere che probabilmente è necessario intervenire su quest'ultimi in modo da aumentare ulteriormente il distacco di punteggio ottenibile dai due giocatori.

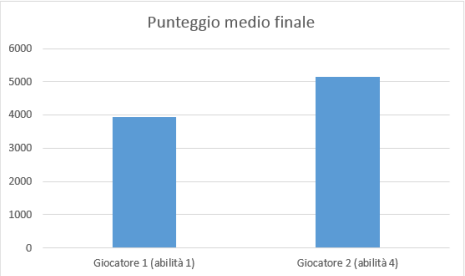


Figura 5.8: Risultati delle simulazioni a 2 giocatori artificiali con diversi livelli di abilità

5.4.2 Test 4.2: partita a quattro giocatori

In questo scenario si vede la partecipazione simultanea di tutti i livelli di abilità previsti, quindi di 4 giocatori comandati da intelligenza artificiale. Di seguito vengono elencati tutti i parametri di partenza:

Business Module
Giocatore 1 - livello abilità 1
NUMBER_OF_CITIES_VISITED_PER_TURN_MONEY = 8;
NUMBER_OF_CITIES_VISITED_PER_TURN_POINTS = 8;
THRESHOLD_DISTANCE_MAX = 0.10f;
THRESHOLD_DISTANCE_INCREASE_PER_YEAR = 0.03f;
Giocatore 2 - livello abilità 2
NUMBER_OF_CITIES_VISITED_PER_TURN_MONEY = 10;
NUMBER_OF_CITIES_VISITED_PER_TURN_POINTS = 10;
THRESHOLD_DISTANCE_MAX = 0.11f;
THRESHOLD_DISTANCE_INCREASE_PER_YEAR = 0.03f;
Giocatore 3 - livello abilità 3
NUMBER_OF_CITIES_VISITED_PER_TURN_MONEY = 15;
NUMBER_OF_CITIES_VISITED_PER_TURN_POINTS = 15;
THRESHOLD_DISTANCE_MAX = 0.15f;
THRESHOLD_DISTANCE_INCREASE_PER_YEAR = 0.03f;
Giocatore 4 - livello abilità 4
NUMBER_OF_CITIES_VISITED_PER_TURN_MONEY = 20;
NUMBER_OF_CITIES_VISITED_PER_TURN_POINTS = 20;
THRESHOLD_DISTANCE_MAX = 0.15f;
THRESHOLD_DISTANCE_INCREASE_PER_YEAR = 0.03f;

Construction Module
Giocatore 1 - livello abilità 1
ACTIONS_PER_TURN = 4;
FIRST_CONSTRUCTION_DELAY = 0;
SMART_CONSTRUCTION_OF_FIRST_HUB = false;
MAX_TRACK_CONNECTION_CHAIN = 2;
MINIMUM_DISTANCE_FROM_OTHERS = 6;
MAINTAIN_DISTANCE_FROM_OTHERS_DURATION = 0.25f;
WISH_TO_EXPAND = 0.01f;
EXPAND_AGGRESSIVENESS = 0.0f;
MAX_STATIONS_COMPARED_TO_OTHERS = 200;
WISH_TO_BUILD_NEW_HUBS = 0.1f;
MAX_HUBS_COMPARED_TO_HUMANS = 0;
PROBABILITY_TO_BUILD_DETACHED_NETWORKS = 0.5f;
UPGRADE_POLICY = AIProfile_ConstructionManager.UpgradePolicy.Gradual;
UPGRADE_MINTHRESHOLD = 0.5f;
UPGRADE_RESPONSE = 0.15f;
Giocatore 2 - livello abilità 2
ACTIONS_PER_TURN = 4;
FIRST_CONSTRUCTION_DELAY = 0;
SMART_CONSTRUCTION_OF_FIRST_HUB = false;
MAX_TRACK_CONNECTION_CHAIN = 3;
MINIMUM_DISTANCE_FROM_OTHERS = 6;
MAINTAIN_DISTANCE_FROM_OTHERS_DURATION = 0.25f;
WISH_TO_EXPAND = 0.02f;
EXPAND_AGGRESSIVENESS = 0.1f;
MAX_STATIONS_COMPARED_TO_OTHERS = 4;
WISH_TO_BUILD_NEW_HUBS = 0.15f;
MAX_HUBS_COMPARED_TO_HUMANS = 1;
PROBABILITY_TO_BUILD_DETACHED_NETWORKS = 0.25f;
UPGRADE_POLICY = AIProfile_ConstructionManager.UpgradePolicy.Gradual;
UPGRADE_MINTHRESHOLD = 0.5f;
UPGRADE_RESPONSE = 0.15f;
Giocatore 3 - livello abilità 3
ACTIONS_PER_TURN = 5;
FIRST_CONSTRUCTION_DELAY = 1;
SMART_CONSTRUCTION_OF_FIRST_HUB = true;
MAX_TRACK_CONNECTION_CHAIN = 3;
MINIMUM_DISTANCE_FROM_OTHERS = 6;
MAINTAIN_DISTANCE_FROM_OTHERS_DURATION = 0.10f;
WISH_TO_EXPAND = 0.03f;
WISH_TO_BUILD_NEW_HUBS = 0.1f;
MAX_HUBS_COMPARED_TO_HUMANS = 3;
EXPAND_AGGRESSIVENESS = 0.2f;
MAX_STATIONS_COMPARED_TO_OTHERS = 6;
PROBABILITY_TO_BUILD_DETACHED_NETWORKS = 0.5f;
UPGRADE_POLICY = AIProfile_ConstructionManager.UpgradePolicy.Smart;
UPGRADE_MINTHRESHOLD = 0.5f;
UPGRADE_RESPONSE = 0.15f;
Giocatore 4 - livello abilità 4
ACTIONS_PER_TURN = 6;
FIRST_CONSTRUCTION_DELAY = 1;
SMART_CONSTRUCTION_OF_FIRST_HUB = true;
MAX_TRACK_CONNECTION_CHAIN = 4;
MINIMUM_DISTANCE_FROM_OTHERS = 6;
MAINTAIN_DISTANCE_FROM_OTHERS_DURATION = 0.10f;
WISH_TO_EXPAND = 0.04f;
EXPAND_AGGRESSIVENESS = 0.3f;
WISH_TO_BUILD_NEW_HUBS = 0.1f;
MAX_HUBS_COMPARED_TO_HUMANS = 4;
MAX_STATIONS_COMPARED_TO_OTHERS = 10;
PROBABILITY_TO_BUILD_DETACHED_NETWORKS = 0.75f;
UPGRADE_POLICY = AIProfile_ConstructionManager.UpgradePolicy.Smart;
UPGRADE_MINTHRESHOLD = 0.5f;
UPGRADE_RESPONSE = 0.15f;

Fleet Management Module
Giocatore 1 - livello abilità 1
ACTIONS_PER_TURN = 2;
MAX_CONNECTION_CHAIN = 2;
ENGINEPOLICY = AIProfile_FleetManager.EnginePolicy.Random;
WAGONPOLICY = AIProfile_FleetManager.WagonPolicy.MaxLoad;
CONSTRUCTION_TO_TRAIN_RESPONSE = 0.6f;
ROUTE_REVISION_SPEED = 0.05f;
Giocatore 2 - livello abilità 2

```
ACTIONS_PER_TURN = 2;
MAX_CONNECTION_CHAIN = 3;
ENGINEPOLICY = AIProfile_FleetManager.EnginePolicy.Random;
WAGONPOLICY = AIProfile_FleetManager.WagonPolicy.Random;
CONSTRUCTION_TO_TRAIN_RESPONSE = 0.7f;
ROUTE_REVISION_SPEED = 0.1f;
Giocatore 3 - livello abilità 3
ACTIONS_PER_TURN = 4;
MAX_CONNECTION_CHAIN = 3;
ENGINEPOLICY = AIProfile_FleetManager.EnginePolicy.BetterLoad;
WAGONPOLICY = AIProfile_FleetManager.WagonPolicy.Random;
CONSTRUCTION_TO_TRAIN_RESPONSE = 0.8f;
ROUTE_REVISION_SPEED = 0.2f;
Giocatore 4 - livello abilità 4
ACTIONS_PER_TURN = 4;
MAX_CONNECTION_CHAIN = 2;
ENGINEPOLICY = AIProfile_FleetManager.EnginePolicy.BetterLoad;
WAGONPOLICY = AIProfile_FleetManager.WagonPolicy.Random;
CONSTRUCTION_TO_TRAIN_RESPONSE = 0.8f;
ROUTE_REVISION_SPEED = 0.25f;
```

Anche in questo caso le prime simulazioni mostrano risultati incoraggianti (Figura 5.9): l'ordine dei punteggi medi finali rispecchia quello del livello di abilità dei giocatori, e la percentuale di vittoria è nettamente favorevole per i giocatori più abili, tuttavia è possibile migliorare ulteriormente i risultati aumentando la differenza di punteggio tra i giocatori.

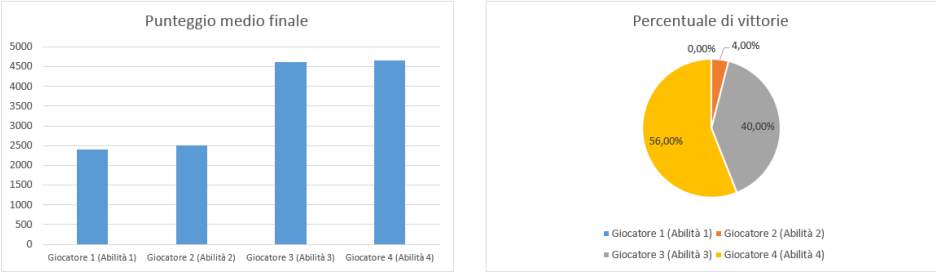


Figura 5.9: Risultati delle simulazioni a 4 giocatori artificiali con diversi livelli di abilità (prima prova)

Per questo motivo viene eseguita una seconda serie di test modificando i parametri come mostrato in Tabella 5.7. Inoltre poichè si è osservato che capitava spesso che le I.A. di livello più alto si trovassero in condizioni di bancarotta che le impediva di continuare con la costruzione [2], si è deciso di disabilitare il conteggio dei soldi per verificare come il comportamento finale dipendesse solo ed esclusivamente dalle politiche adottate.
I dati ottenuti, riassunti nei grafici in Figura 5.10 sono particolarmente buoni: si vede come il punteggio medio ottenuto aumenti in maniera praticamente lineare all'aumentare del livello di abilità dell'intelligenza artificiale

[2] Al momento di effettuare i test il modulo di gestione patrimoniale non era ancora completo del tutto, e i soldi venivano assegnati seguendo semplici politiche statiche che non tengono in considerazione la situazione attuale

	Vecchio valore			
Parametro	1	2	3	4
EXPAND_AGGRESSIVENESS	0	0,1	0,2	0,3
WISH_TO_EXPAND	0,01	0,02	0,03	0,04
CONSTRUCTION_TO_TRAIN_RESPONSE	0,6	0,7	0,8	0,8
MAX_CONNECTION_CHAIN	2	2	3	4
BUILD_ DETACHED_NETWORKS	0,5	0,25	0,5	0,75
*MONEY_ALLOW_EVERYTHING	false	false	false	false
	Nuovo valore			
Parametro	1	2	3	4
EXPAND_AGGRESSIVENESS	0	0,12	0,14	0,5
WISH_TO_EXPAND	0,009	0,02	0,026	0,04
CONSTRUCTION_TO_TRAIN_RESPONSE	0,55	0,7	0,8	0,8
MAX_CONNECTION_CHAIN	2	2	3	3
BUILD_ DETACHED_NETWORKS	0,5	0,25	0,5	0,7
*MONEY_ALLOW_EVERYTHING	true	true	true	true
	Variazione			
Parametro	1	2	3	4
EXPAND_AGGRESSIVENESS	+0,00%	**+20,00%**	**-42,86%**	**+66,67%**
WISH_TO_EXPAND	**-11,11%**	+0,00%	**-15,38%**	+0,00%
CONSTRUCTION_TO_TRAIN_RESPONSE	**-9,09%**	+0,00%	+0,00%	+0,00%
MAX_CONNECTION_CHAIN	+0,00%	+0,00%	+0,00%	**-33,33%**
BUILD_ DETACHED_NETWORKS	+0,00%	+0,00%	+0,00%	**-7,14%**
*MONEY_ALLOW_EVERYTHING	-	-	-	-

Tabella 5.7: Modifica dei parametri dell'I.A. al fine di equilibrare i livelli di difficoltà

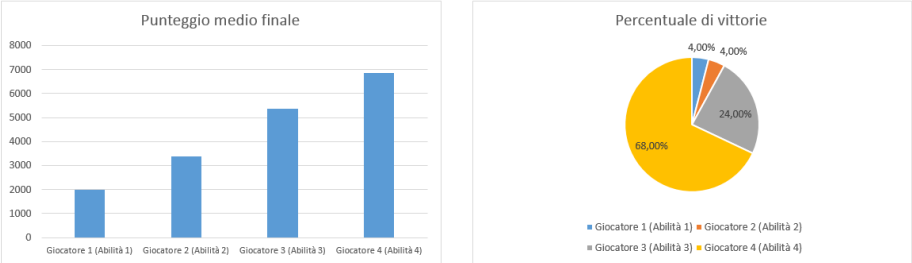


Figura 5.10: Risultati delle simulazioni a 4 giocatori artificiali con diversi livelli di abilità (seconda prova)

e come la maggior parte delle partite venga vinta dal giocatore più abile. Questo tipo di risultato rispecchia certamente ciò che si voleva ottenere dal tipo di intelligenza artificiale, dimostra che il lavoro svolto fin'ora è certamente funzionale e offre un ottimo punto di partenza per sviluppi futuri del progetto.

Capitolo 6

Conclusioni e sviluppi futuri

In quest'ultimo capitolo verrà riassunto il lavoro svolto e verranno fatte alcune riflessioni su ciò che si è riuscito ad ottenere, in particolare riguardo ai dati e all'esperienza raccolta durante le simulazioni. In chiusura verrà presentata una breve riflessione su come il lavoro può essere portato avanti e su come la conoscenza acquisita possa essere utile ad un eventuale sviluppo futuro del progetto.

6.1 Conclusioni

Lo scopo di questa tesi era duplice: da una parte c'è stato il completamento di una parte progettuale non banale che ha portato alla realizzazione di un prodotto videoludico sicuramente interessante, dall'altra si è riusciti a sfruttare questa opportunità per realizzare una parte di analisi e di ricerca nel campo dell'intelligenza artificiale anch'essa piuttosto articolata. Dopo aver analizzato i principali modelli ed i principali strumenti utilizzati in questo settore, si è proceduto nell'implementare un motore di intelligenza artificiale che prendesse spunto proprio da tale analisi. I problemi di movimento, di decisione e di strategia tipici dell'implementazione di un motore di intelligenza artificiale per giochi sono stati affrontati utilizzando sia una serie di

strumenti che ormai possono essere definiti *standard* (Algoritmi A* di pathfinding, macchine a stati, mappe di influenza), sia una serie di soluzioni specifiche per il gioco in questione (manager dei turni, TurnSyncedObject, sistema modulare a scambio di messaggi) ed il risultato ottenuto appare convincente. Ciò appare confermato dai dati raccolti in fase di simulazione che dimostrano come l'intelligenza artificiale implementata sia in grado di giocare autonomamente e di fare quello per cui è stata progettata, ovvero accumulare il maggior numero di punti possibili. Ovviamente, come anche dimostrato dai test, il motore di intelligenza artificiale può essere ulteriormente migliorato e i dati raccolti si riveleranno molto utili a questo scopo. La base di partenza appare tuttavia piuttosto solida. Si è visto, per esempio, come una politica che si basa sull'accumulo di punti grazie all'espansione della rete sia utile fino ad un certo punto e di come sia necessario intervenire ulteriormente sulla micro gestione dei singoli treni, oppure come esista un certo squilibrio tra l'utilizzo di convogli veloci rispetto a convogli in grado di trasportare un numero maggiore di merci. Tutta questa conoscenza verrà sicuramente utilizzata negli sviluppi futuri del progetto, che comprendono, in primis, nel perfezionamento delle problematiche principali che è stato possibile rilevare soltanto grazie ad uno studio approfondito dei dati raccolti.

Bibliografia

[1] Andrew Bacon. Expert system in a gaming enviroment, 2010.

[2] Gordon Calleja. *In-Game: From Immersion to Incorporation*. The MIT Press, 1st edition, 2011.

[3] Alex J. Champandard. 10 reasons the age of finite state machines is over, 2007.

[4] Alex J. Champandard. Flanking total war's ai: 11 tricks to conquer for your game, 2007.

[5] Jaime Griesemer Chris Butcher. The illusion of intelligence, 2002.

[6] Viktor ChvÃ¡tal. Practical application of ai methods in rts games. Master's thesis, Czech Technical University of Prague, 2010.

[7] Roger Davies. Artificial intelligence overview part i - importance of heuristics and teaching machines to think, 2009.

[8] Luke Dicken. Game ai vs traditional ai, 2011.

[9] Max Dyckhoff and LLC Bungie. Decision making and knowledge representation in halo 3. In *Presentation at the Game Developers Conference*, 2008.

[10] Inc. Epic Games. Navigation mesh reference, 2012.

[11] Ross Graham. Real-time agent navigation with neural networks for computer games. Master's thesis, Institute of Technology Blanchardstown, 2006.

[12] Ross Graham, Hugh McCabe, and Stephen Sheridan. Pathfinding in computer games. *ITB Journal*, 8, 2003.

[13] Stephen Grand, Dave Cliff, and Anil Malhotra. Creatures: Artificial life autonomous software agents for home entertainment. In *Proceedings of the first international conference on Autonomous agents*, pages 22–29. ACM, 1997.

[14] Johan Hagelbäck and Stefan J Johansson. Using multi-agent potential fields in real-time strategy games. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 2*, pages 631–638. International Foundation for Autonomous Agents and Multiagent Systems, 2008.

[15] Chris Hecker. My liner notes for spore/spore behavior tree docs, 2009.

[16] John Funge Ian Millington. *Artificial Intelligence for Games*. Morgan Kaufmann, second edition, 2009.

[17] Damian Isla. Handling complexity in the halo 2 ai. 2005.

[18] Aleks Jakulin. Artificial intelligence in games: Food for thought series, 2003.

[19] Donald Kehoe. Designing artificial intelligence for games, 2009.

[20] Sven Koenig and Maxim Likhachev. D* lite. In *AAAI/IAAI*, pages 476–483, 2002.

[21] PopTop Software ltd. Railroad tycoon 3 manual, 2003.

[22] Roberto Lucchetti. *A Primer in Game Theory*. Editrice Esculapio, 2011.

[23] Jamey Pittman. The pacman dossier, 2011.

[24] Thomas Randall, Peter Cowling, Roderick Baker, and Ping Jiang. Using neural networks for strategy selection in real-time strategy games. In *Symposium on AI &amp*, 2009.

[25] Stefano Ruschetta. Sviluppo dell'intelligenza artificiale per il gioco da tavolo magnifico: Da vinci's art of war. Master's thesis, Politecnico di Milano, 2011.

[26] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *science*, 317(5844):1518–1522, 2007.

[27] David Silver. Cooperative pathfinding. In *AIIDE*, pages 117–122, 2005.

[28] Alan Skorkin. Why developers never use state machines, 2011.

[29] Kenneth O Stanley, Bobby D Bryant, and Risto Miikkulainen. Evolving neural network agents in the nero video game. 2005.

[30] Anthony Stentz. The focussed dˆ* algorithm for real-time replanning. *IJCAI*, 95:1652–1659, 1995.

[31] Nathan R Sturtevant and Michael Buro. Improving collaborative pathfinding using map abstraction. In *AIIDE*, pages 80–85, 2006.

[32] Moshe Tennenholtz. Game theory and artificial intelligence. In *Foundations and Applications of Multi-Agent Systems*, pages 49–58. Springer, 2002.

[33] OpenTTD wiki community. Openttd wiki - game mechanics, 2014.

[34] Geogios N. Yannakakis. Game ai revisited. In *Proceedings of the 9th Conference on Computing Frontiers*, CF '12, pages 285–292, New York, NY, USA, 2012. ACM.

Appendice A

Algoritmi e Pseudocodice

A.1 Algoritmo A*

Tipico algoritmo di pathfinding.

```
initialize openset (unexplored nodes) with starting node s;
initialize the empty list closedset (explored nodes);
while openset not empty do
    current ← node n in openset where the cost function f(n) is minimum;
    if current is goal then
        return solution;
    else
        remove current from openset;
        add current to closedset;
        expand current, generating all its successors succ, with a pointer back to current;
        for all successors succ of node current do
            calculate the cost function f(succ);
            if succ not in openset and succ not in closedset then
                add succ to openset;
                assigne the value of f(succ) to node succ;
            else
                if f(succ) ≤ the previous value then
                    update the value of node succ with the new value of f(succ) and its
                    predecessor with node current;
                end if
                if succ is in closedset then
                    move back succ to openset;
                end if
            end if
        end for
    end if
end while
No solution found;
```

A.2 Dijkstra

Utilizzabile sia come algoritmo di pathfinding, si rivela anche molto utile anche in tecniche di ricerca strategico/tattica.

```
for all vertex v in Graph do
    distance[v] ← infinity;
    previous[v] ← null;
end for
distance[s] ← 0, with s starting node;
initialize openset with all the nodes in the Graph;
while openset not empty do
    current ← node n in openset with the minimum distance in distance[];
    remove current from openset;
    if distance[current] is infinite then
        BREAK, all remaining nodes are unreachable from the current;
    end if
    for all neighbour n of node current do
        newCost ← distance[current] + distance_between(current, n);
        if newCost ≤ distance[n] then
            distance[n] ← newCost;
            previous[n] ← current;
            reorder node n in openset (decrease index);
        end if
    end for
end while
return the list of all distances from node s: distance[], and the list of predecessors:
previous[];
```

A.3 Alpha-Beta Pruning

Tecnica di potatura dell'albero usata per diminuire il numero di nodi valutati dalla tecnica minimax (Vedi Sezione 2.4.2)

```
function alphabeta(node, depth, a, b, maximizingPlayer)
if depth = 0 or node is a terminal node then
   return the heuristic value of the node;
end if
if maximizingPlayer then
   for all child of node do
      a ← max(a,alphabeta(child,depth − 1,a,b,false);
      if b ≤ a then
         Break (potatura Beta);
      end if
      return a;
   end for
else
   for all child of node do
      b ← min(b,alphabeta(child,depth − 1,a,b,true);
      if b ≤ a then
         Break (potatura Alpha);
      end if
      return b;
   end for
end if
end function
Chiamata iniziale:
alphabeta(nodoOrigine,depth,−infinity,+infinity,true);
```

Appendice B

Lista dei parametri utilizzati dall'intelligenza artificiale

In quest'appendice vengono elencati tutti i parametri che descrivono il comportamento dell'intelligenza artificiale implementata. Per ogni parametro viene inoltre fornita una breve descrizione.

Modulo di gestione della flotta		
Parametro	**Tipo**	**Descrizione**
ACTIONS_PER_TURN	int	Determina quante azioni il modulo è in grado di eseguire per ogni turno
MAX_CONNECTION_CHAIN	int	Profondità della ricerca nei nuovi collegamenti 2:A-B 3:A-B-C 4:A-B-C-D etc.
CONSTRUCTION_TO_TRAIN_RESPONSE	float	Più il valore è alto, più rapidamente l'IA reagirà alla costruzione di nuovi segmenti acquistando nuovi treni
ENGINE_POLICY	enum	Politica nella scelta delle locomotive: Random, MostExpensive, BetterLoad, BetterSpeed, Balanced (Vedi Sezione 5.3)
WAGON_POLICY	enum	Politica nella scelta dei vagoni: Random, MaxSpeed, MaxLoad (Vedi Sezione 5.3)
ROUTE_REVISION_SPEED	float	Più il valore è alto, più frequentemente l'IA revisionerà la validità delle rotte correnti dei treni

Tabella B.1: Lista dei parametri utilizzati dall'intelligenza artificiale (Modulo di gestione della flotta)

Modulo di costruzione		
Parametro	**Tipo**	**Descrizione**
ACTIONS_PER_TURN	int	Determina quante azioni il modulo è in grado di eseguire per ogni turno
FIRST_CONSTRUCTION_DELAY	int	Turni di attesa prima di effettuare la prima costruzione
SMART_CONSTRUCTION_OF_ FIRST_HUB	bool	Scelta del primo punto di costruzione FALSE: scelta casuale TRUE: scelta ponderata
MAX_TRACK_CONNECTION_CHAIN	int	Profondità della ricerca nei nuovi collegamenti 2:A-B 3:A-B-C 4:A-B-C-D etc.
MINIMUM_DISTANCE_FROM_OTHERS	int	Quanto l'IA deve tenersi distante dagli altri giocatori quando vuole costruire nuovi segmenti
MAINTAIN_DISTANCE_FROM_ OTHERS_DURATION	float	Per quanto tempo l'IA deve tenersi distante dagli altri giocatori (% della durata totale della partita)
MAX_STATIONS_COMPARED_TO_ OTHERS	int	L'IA non cercherà di espandersi ulteriormente se ha già pidì questo numero di stazioni rispetto agli altri giocatori
WISH_TO_EXPAND	float	Determina il desiderio di espansione dell'IA
EXPAND_AGGRESSIVENESS	float	Aggressività nella costruzione in risposta alla posizione attuale del giocatore e alla differenza di binari costruiti rispetto agli altri giocatori
PROBABILITY_TO_BUILD_ DETACHED_NETWORKS	float	Probabilità di costruire segmenti di rotta distaccati
WISH_TO_BUILD_NEW_HUBS	float	Determina la volontà di migliorare le stazioni già esistenti costruendo nuovi hub
MAX_HUBS_COMPARED_TO_HUMANS	int	I nuovi hub non verranno costruiti (vedi sopra) se l'IA ne possiede già più di questo numero rispetto ai giocatori umani
TRANSPORT_POLICY	enum	Tipo di collegamento preferito: OnlyGoods, PreferGoods, PreferPassengers, PreferMail, OnlyPassengers, OnlyMail
UPGRADE_POLICY	enum	In che modo l'IA effettua l'upgrade dei binari: Gradual, ImmediateMax, Smart
UPGRADE_MINTHRESHOLD	float	L'IA ignorerà i binari con un valore di congestione inferiore a questa soglia
UPGRADE_RESPONSE	float	Più questo valore è alto, minore sarà il tempo di reazione nell'upgrade dei binari congestionati

Tabella B.2: Lista dei parametri utilizzati dall'intelligenza artificiale (Modulo di costruzione)

Modulo di business		
Parametro	**Tipo**	**Descrizione**
CITIES_VISITED_PER_TURN_MONEY	int	Determina quante città vengono visitate ogni turno al fine di trovare dei collegamenti utili a guadagnare denaro
CITIES_VISITED_PER_TURN_POINTS	int	Determina quante città vengono visitate ogni turno al fine di trovare dei collegamenti utili a guadagnare punti
THRESHOLD_DISTANCE_MIN	float	I collegamenti con distanza al di sotto di questa soglia verranno ignorati (% rispetto al collegamento massimo possibile)
THRESHOLD_DISTANCE_MAX	float	I collegamenti con distanza al di sopra di questa soglia verranno ignorati
THRESHOLD_DISTANCE_INCREASE	float	Quanto aumenta (per ogni anno di gioco) il parametro sopra
THRESHOLD_MONEY_MIN	int	I collegamenti che non garantiscono almeno questa somma di denaro verranno ignorati
THRESHOLD_SCORE_MIN	int	I collegamenti che non garantiscono almeno questa somma di punti verranno ignorati
RESEARCH_CONSIDER_GOODS	bool	La ricerca deve tener conto delle merci?
RESEARCH_CONSIDER_PASSENGERS	bool	La ricerca deve tener conto dei passeggeri?
RESEARCH_CONSIDER_MAIL	bool	La ricerca deve tener conto della posta?

Tabella B.3: Lista dei parametri utilizzati dall'intelligenza artificiale (Modulo di business)

Indice analitico

A

AB pruning, *vedi* Potatura Alfa-Beta
Alberi comportamentali, 13, 25, 26
Algoritmo
    A*, 9, 21, 29, 42, 69, 76, 115
    cooperative A*, 22
    definizione, 21
    Dijkstra, 9, 23, 116
    hierarchical A*, 23
    local repair A*, 22

B

Backward Induction, *vedi* Induzione a Ritroso
Behavior tree, *vedi* Alberi comportamentali
Binari, 47
    aggiornamento, 47
    calcolo dell'utilità, 59
    livelli, 47
    scambio, 47
    segmento, 47
Board games, 9, 15, 28, 29, 32

C

Città, 39, 43, 55, 73
    domanda e offerta, 44
    livello, 45, 63, 64
    passeggeri e posta, 45
    premi, 46, 51, 63
    produzione, 44

D

Debug, 88
Distribuzione di probabilità, 80

E

Economia, 52
Esecuzione out of order, 77
Euristica, 16, 19
    Most constrained variable, 19

F

Finestra percettiva, 6
Finite State Machine, *vedi* Macchine a stati finiti

G

Giochi
    a informazione perfetta, 28, 33
    a somma zero, 28, 34
    in forma estesa, 33
Goal, 80

H

Hex board, *vedi* Mappa esagonale
Hub ferroviario, 48, 51, 94

I

Induzione a ritroso, 28, 33
Intelligenza artificiale, 65
    accademica, 6, 36
    intelligibile, 17
    modello implementativo, 8, 66
    moduli, 69, 76

profili, 69
Iron Duke, 2, 29, 32, 37, 55, 65

L
Locomotiva, *vedi* Treno

M
Macchine a stati finiti, 25, 84
Manager dei turni, 38, 66, 87
Map flooding, 15, 24, 94, 95
Mappa esagonale, 30, 38, 41, 42
Mappe di influenza, 15, 33
Merci, 38, 43–45, 48, 51, 55
Mesh navigazionali, 10
Minmax, 28, 34
Multithreading, 67, 79, 86

N
Navmesh, *vedi* Mesh navigazionali

P
Passeggeri, 38, 43, 45, 48, 51, 57, 63
Pathfinding, 8–10, 18, 19, 21, 42, 69
cooperativo, 75
Playtest, 3
Posta, 45, 48, 52, 57, 63
Potatura Alfa-Beta, 28, 35, 117
Premi, 38, 46

R
Reservation table, 23, 75
Reti neurali, 12

S
Scalo ferroviario, 61
Sistemi esperti, 12
Spritesheet, 42
Stazioni, 48, 61

T
Teoria dei giochi, 2, 13, 28, 33
Time unit, 38, 39
Treno, 39
locomotiva, 48
manutenzione, 61
manutenzione straordinaria, 48
vagoni, 48, 51
TurnManager, *vedi* Manager dei turni
TurnSyncedObject, 38, 39, 44, 48, 66, 77

V
Von Neumann, 34

W
Waypoint tactic, 14

X
XML, 42, 44