Progetto di Fondamenti di Intelligenza Artificiale

ArmorPickerAI

Autore	Matricola
Andrea Ruggiero	0512114732

Indice

1 Introduzione

Abbiamo ipotizzato di aver a che fare con un videogame in linea con il famoso loot & shooter **Destiny 2** prodotto dalla software house Bungie. Proprio come in Destiny 2, in questo gioco che chiameremo **DungeonMasterAI** tutte le attività che siano dungeon, raid o semplici esplorazioni del mondo sono volte al collezionamento di nuove armi o armature utili al potenziamento del personaggio per permettergli di accedere a zone e contenuti di più alto livello. Focalizzandoci sulle armature esistono 5 tipologie di pezzo:

- Elmo
- Guanti
- Corpetto
- Pantaloni
- Mantello

Ognuno di questi pezzi ha delle statistiche suddivise in 6 categorie **Mobilità, Resilienza, Recupero, Disciplina, Intelletto e Forza** oltre che una rarità tra **Comune, Non Comune, Raro, Leggendario ed Esotico**

Figure 1: Immagine di un pezzo di armatura da Destiny 2

Le statistiche di ogni singolo pezzo hanno un totale (nella Figure 1 evidenziato in rosso) che rappresenta la somma delle statistiche del pezzo, queste sono inoltre suddivise in due macro categorie una formata da **Mobilità, Resilienza e Recupero** (nella Figure 1 evidenziato in verde) e l'altra formata da **Disciplina, Intelletto e Forza** (nella Figure 1 evidenziato in blu). Quando un pezzo viene assegnato al giocatore come ricompensa dal completamento di qualsivoglia attività ogni macro categoria ha una soglia massima di 34 punti suddivisi tra le categorie della stessa, pertanto ogni pezzo può avere delle statistiche totali di massimo 68 punti. Il personaggio avrà anche esso le 6 categorie prodotte ognuna dalla somma dei punti di quella categoria di ognuno dei 5 pezzi di armatura indossati. Massimizzare i punti categoria è fondamentale per la progressione del personaggio del mondo di gioco in quanto ogni categoria a suo modo migliora le capacità del persoaggio migliorandone ad esempio il recupero di salute, la resistenza ai danni, la potenza di abilità corpo a corpo o della granata. Ricordando però che tutti i punti della singola statistica che eccedono il 100 non porteranno nessun beneficio al giocatore.

2 Definizione del problema

L'obiettivo di **ArmorPickerAI** è quello di prendere tutte le armature che un giocatore può riporre nel suo inventario (per un massimo di 50 unità) o nel suo deposito (per un massimo di 600 unità) e trovare una giusta conbinazione di pezzi per la creazione di un armor set che massimizzi le statistiche del personaggio in generale per tutte le caratteristiche e che nello specifico possa focalizzarsi su una caratteristica per macro categoria che il giocatore vuole massimizzare per adattare l'utilizzo del suo personaggio al suo stile di gioco.

2.1 Specifica PEAS

2.1.1 Performance

Le prestazioni dell'agente sono valutate attraverso:

- Scalabilità: L'agente deve adattare il suo funzionamento a qualsivoglia account in termini di ricchezza di inventario e deposito assumendo tuttavia che nelle prime fasi di gioco un giocatore è più focalizzato sul progredire all'interno del mondo piuttosto che massimizzare le proprie statistiche, inoltre sarebbe insensato l'utilizzo di un software esterno per la generazione del prefetto armor set quando si hanno a disposizione pochi pezzi tra i quali sarebbe possibile scegliere autonomamente. L'utilizzo dell'algoritmo è più che altro attribuito a giocatori navigati che vogliono potenziarsi per poter affrontare sfide di "End game" pertanto si vuole che l'agente sia particolarmente performante con account molto ricchi senza però precludere il corretto funzionamento anche con account cosiddetti "Early game".

- Tempo di esecuzione: L'agente deve terminare la sua ricerca di migliori armor set in una tempistica accettabile.

- Convergenza finale: Alla fine dell'esplorazione si vuole avere una scelta formata dai migliori armor set componibili con i pezzi di armatura a disposizione dell'utente, che tuttavia, non siano tutti uguali tra loro per permettere cosi al giocatore di scegliere quello che più lo aggrada.

2.1.2 Environment

- Completamente osservabile: L'agente una volta recuperato il contenuto degli account conosce tutte le caratteristiche di ogni singolo pezzo di armatura presente in esso.
- Agente singolo: L'agente opera nell'ambiente da solo.
- Stocastico: L'agente non può determinare lo stato successivo in base allo stato corrente a causa dell'elemento di randomicità di crossover e mutazione.
- Discreto: L'agente si interfaccia con un ambiente che ha un insieme finito e contabile di stati, azioni e pezzi di armatura.
- Statico: L'agente opera in un ambiente che una volta caricato non cambia se non per mano dell'agente stesso
- Sequenziale: L'agente ad ogni generazione dipende dalle precedenti e contribuisce al processo evolutivo complessivo.

2.1.3 Actuators

- Mostrare in output a schermo un insieme di armor set ottimali.

2.1.4 Sensors

- Dai log di gioco possiamo reperire un file di testo dove sono registrati tutti i pezzi di armatura appartenenti al giocatore. L'agente dovrà quindi prendere in input questo file ed estrarne le informazioni.

3 Definizione della soluzione

Definito il problema è ovvia la necessità di un algoritmo che ricerchi i migliori armor set all'interno del nostro dominio. Rientriamo dunque in un caso di **Ottimizzazione**. Optando per utilizzo della meta-euristica dell'**Algoritmo Genetico** creeremo un algoritmo di ricerca che si adatta al nostro problema non solo perchè un armor set formato dai 5 diversi pezzi di armatura sopracitati costituisce perfettamente un individuo ma soprattutto in quanto il nostro problema si adatta al concetto del miglioramento naturale di una specie, basterebbe infatti immaginare i diversi pezzi di armatura come parti differenti del corpo di un animale (squame, artigli, pelo, ecc...) dove gli individui con tratti ereditabili che ben si adattano all'ambiente tendono a sopravvivere.

3.1 Creazione dell'algoritmo

Creeremo dunque un algoritmo di ricerca seguendo la meta-euristica dell'algoritmo genetico andando a ricostruire tutti i punti cardine di quest'ultimo Andando ad utilizzare per ogni punto le

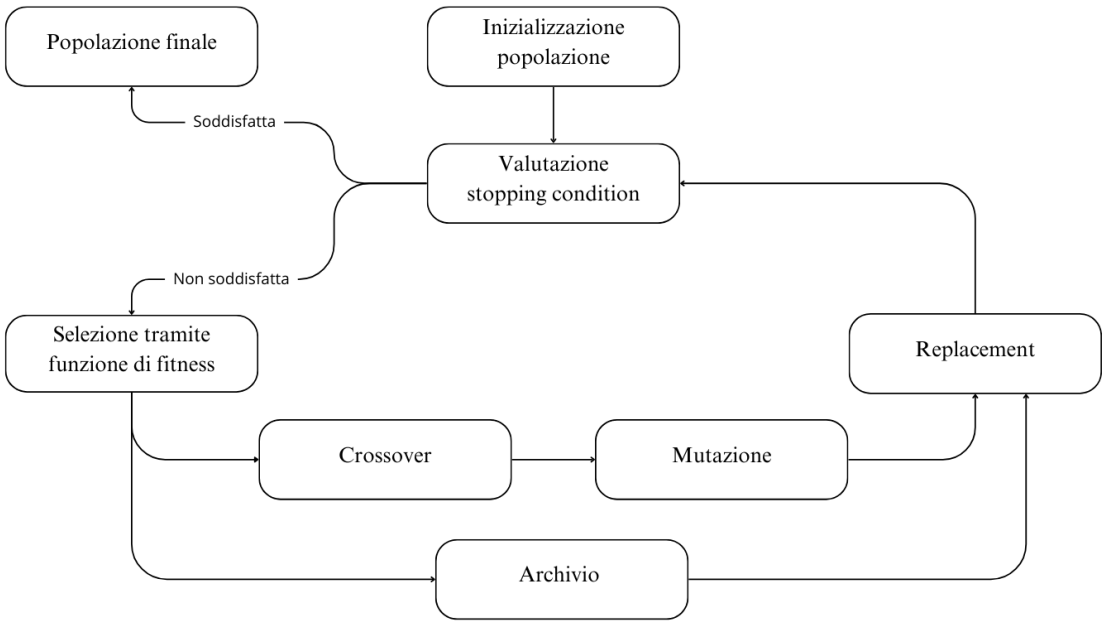

Figure 2: Definizione del core dell'algoritmo

tecniche che più si adattano alla risoluzione del nostro problema fornendo anche degli improvement in aiuto a questo algoritmo **cieco** quali l'uso di **euristiche problem-specific** e **archive strategy**.

3.1.1 Inizializzazione della popolazione

Come abbiamo detto il nostro agente prenderà in input in file contenente i pezzi di armatura posseduti dal giocatore. La creazione degli armor set sarà un compito affidato solo e soltanto a questa parte che quindi assume molta importanza in quanto è l'unica parte del nostro algoritmo in cui il ragionamento si basa sui pezzi di armatura e non ancora sugli armor set. E bene dunque favorire i pezzi di armatura con un totale delle statistiche più alto ma senza precludere la possibilità a pezzi di armatura con un basso totale l'ingresso quantomeno nella popolazione iniziale, questo perchè alcuni pezzi benchè abbiano un basso totale delle statistiche potrebbero avere una o entrambe le statistiche favorite nelle macro catogorie dal giocatore rendendolo un pezzo appetibile.

3.1.1.1 Implementazione

Per la realizzazione della popolazione iniziale abbiamo optato per l'utilizzo di una **soglia fissa** impostata a 10000 individui che la comporranno. La scelta potrebbe risultare estremamente dispendiosa per account poco ricchi in termini di inventario e deposito, tuttavia essendo l'elaborazione di questo numero di elementi accettabile per tempo di elaborazione e non avendo alcun limite nell'utilizzo della memoria abbiamo optato per un numero di individui che potesse favorire la riuscita della localizzazione degli armor set perfetti in account più ricchi. Successivamente, per la costituzione degli armor set che compongono la popolazione, in primis andiamo a definire un MIN ovvero il numero di pezzi dello stesso tipo che si ripete meno volte, ed infine per ogni armor set che compone la popolazione seglieremo ogni singolo componente con un 50% di possibilità dai primi MIN elementi della tipologia corrispondente al pezzo che si sta scegliendo (ordinati già di default appena caricati dal file di testo in senso decrescente rispetto al totale delle statistiche) e con un 50% di possibilità dall'intera lista di pezzi della tipologia corrispondente al pezzo che si sta scegliendo.

3.1.2 Funzione di fitness

La funzione di fitness è una funzione in grado di associare un valore a ogni armor set, questa misura il livello di adeguatezza degli individui rispetto al problema considerato e guida il processo di selezione. Nel nostro caso nella funzione di fitness bisogna considerare il peso di una categoria per macro categoria alla quale viene attribuito un peso diverso in base a quanto il giocatore desidera prediligerla rispetto alla massimizzazione totale delle statistiche in generale e una penalizzazione per gli armor set che portano una statistica ad eccedere il 100.

3.1.2.1 Implementazione

Per l'implementazione abbiamo optato dunque per una **media pesata con penalizzazione**. Sommati i punti di ogni pezzo per ogni determinata statistica otteniamo i punti dell'armor set della stessa, dopo una verifica dell'eccesso ed eventuale conseguente penalizzazione che consiste nel dimezzare i punti della stessa, si effettua una moltiplicazione tra i punti della statistica dell'armor set e il suo peso (il peso base della statistica non prediletta sarà 1 mentre per la statistica prediletta in ogni macro categoria il suo peso potrà variare da 2 a 5 in base alla preferenza dell'utente). Il risultato verrà immagazzinato in una variabile che sommerà tutti i risultati. Ripetuto questo passaggio per ogni statistica otterremo la nostra media pesata moltiplicando la variabile prodotta dalla somma di tutte le iterazioni per la somma dei pesi.

```
for (int i = 0; i < statistiche.size(); i++) {
    if (statistiche.get(i) > 100) {
        int valorePenalizzato = statistiche.get(i) / 2;
        statistiche.set(i, valorePenalizzato);
    }
    sommaPonderata += statistiche.get(i) * pesi.get(i);
    sommaPesi += pesi.get(i);
}
System.out.println(sommaPonderata / sommaPesi);
return sommaPonderata / sommaPesi;
```

Figure 3: Implementazione calcolo fitness all'interno del codice

3.1.3 Selezione

Nella fase di selezione vogliamo scegliere i candidati alla riproduzione. Mentre nel caso dell'inizializzazione abbiamo dato opportunità anche ai pezzi meno performanti di entrare nella popolazione per le motivazioni sopracitate ora abbiamo la funzione di fitness che tenendo in conto di tutte le variabili necessarie a soddisfare le esigenze del giocatore ci permette di eliminare gli armor set non soddisfacenti.

3.1.3.1 Implementazione

Per l'implementazione abbiamo optato per una tecnica **Tournament Truncation** dove si compie un ordinamento totale dei candidati, la selezione non avviene su base casuale quanto con una selezione rigida dei migliori M individui dove M nel nostro caso corrisponde ai $\frac{3}{4}$ della popolazione. Questo torneo andrà ad ammettere al mating pool gli elementi più promettenti escludendo gli ultimi del torneo.

```
public ArrayList<ArmorSet> populationSelection(ArrayList<Integer> pesi, ArrayList<ArmorSet> popolazione) {
    ArrayList<ArmorSet> popolazioneValutata = new ArrayList<>();
    ArrayList<ArmorSetFitness> armorSetFitnesses = new ArrayList<>();
    for(int i=0; i<popolazione.size(); i++) {
        ArmorSetFitness temp = new ArmorSetFitness(popolazione.get(i), fitnessCalculator(popolazione.get(i), pesi));
        armorSetFitnesses.add(temp);
    }
    ArmorSetFitnessMergeSort asfms = new ArmorSetFitnessMergeSort();
    asfms.mergeSort(armorSetFitnesses, left: 0, right: armorSetFitnesses.size() - 1);
    int postiTorneo = (3 * popolazione.size()) / 4;
    for(int i=0; i<postiTorneo; i++) {
        popolazioneValutata.add(armorSetFitnesses.get(i).getArmorSet());
    }
    return popolazioneValutata;
}
```

Figure 4: Implementazione processo di selezione all'interno del codice

3.1.4 Crossover

La fase di crossover è la fase in cui gli individui che hanno passato la selezione e che quindi fanno parte del mating pool si "accoppiano" creando un nuovo individuo prendendo parte delle caratteristiche da un genitore e parte dall'altro.

3.1.4.1 Implementazione

Per l'implementazione di questa fase abbiamo optato per una variante della tecnica **Uniform** dove ciascun gene i-esimo viene scelto casualmente tra i due geni i-esimi dei genitori dove però non è possibile che l'interezza dei geni provenga da un unico genitore generando dunque un clone di uno dei due genitore deve dunque esservi almeno un gene per ogni genitore.

3.1.5 Mutazione

La fase di mutazione è la fase in cui il figlio appena generato può ricevere o meno, secondo una percentuale indicata dall'utente, una mutazione ad un gene donatogli dai genitori

3.1.5.1 Implementazione

Per l'implementazione di questa fase abbiamo optato per la tecnica **Random Resetting** dove, in base alla percentuale scelta dall'utente che varierà dal 10% al 100% con scatti di 10, ci sarà la possibilità che venga effettuato il cambio casuale di un gene ad un altro valore ammissibile. Ad esempio con una percentuale del 20% ci sarà il 20% di possibilità che uno dei cinque pezzi di armatura donati dai genitore all'armor set figlio venga scambiato con uno casuale della stessa categoria di pezzo all'interno dell'account, limitando tuttavia la casualità della scelta ai primi MIN migliori elementi.

```
public ArmorSet mutationSon(ArmorSet figlio, int percentualeMutazione) {
    Random random = new Random();
    if(random.nextInt( origin: 0,  bound: 100) <= percentualeMutazione) {
        int pezzo = random.nextInt( origin: 0,  bound: 5);
        if(pezzo == 0) {
            figlio.setHelmet(armorPieceDAO.helmets.get(random.nextInt( origin: 0, min)));
        }
        if(pezzo == 1) {
            figlio.setGloves(armorPieceDAO.gloves.get(random.nextInt( origin: 0, min)));
        }
        if(pezzo == 2) {
            figlio.setChestplate(armorPieceDAO.chestplates.get(random.nextInt( origin: 0, min)));
        }
        if(pezzo == 3) {
            figlio.setLeggings(armorPieceDAO.leggings.get(random.nextInt( origin: 0, min)));
        }
        if(pezzo == 4) {
            figlio.setCloack(armorPieceDAO.cloaks.get(random.nextInt( origin: 0, min)));
        }
    }
    return figlio;
}
```

Figure 5: Implementazione processo di mutazione all'interno del codice

3.1.6 Archivio

La archive strategy è una tecnica che ci permette di preservare i migliori individui al proseguire delle generazioni.

3.1.6.1 Implementazione

Subito dopo la selezione vengono scelti i primi N migliori individui scelti tra la popolazione selezionata quindi quella già sottoposta alla selezione e gli individui già presenti in archivio. N equivarrà a $\frac{1}{2}$ della popolazione selezionata andando, col passare delle generazioni a diminuire in numero in funzione della Tournament Truncation effettuata nella selezione. Alla fine della generazione di

una nuova generazione gli elementi in archivio verranno riversati nella nuova popolazione permettendo così all'algoritmo di usufruire dei geni di questi particolari individui molto performanti ma al tempo stesso rimarranno anche salvati in archivio in modo da essere preservati da eventuali mutazioni svantaggiose.

3.1.7 Stopping Condition

La stopping condition è la condizione secondo la quale decidiamo di interrompere il progredire dell'evoluzione o continuare.

3.1.7.1 Implementazione

Nel nostro caso la stopping condition è semplicemente scaturita dal numero di armor set finali che vogliamo sottoporre al giocatore la raffinazione continua dunque finche la generazione non si riduce a 10 individui.

```
public ArrayList<ArmorSet> populationEvolution(int percentualeMutazione, String account, ArrayList<Integer>pesi) {
    ArrayList<ArmorSet> popolazione = populationInizializer(account);
    ArrayList<ArmorSet> archivio = new ArrayList<>();
    Random random = new Random();
    while(popolazione.size() > 10) {
        ArrayList<ArmorSet> popolazioneSelezionata = populationSelection(pesi, popolazione);
        archivio = ArchiveStrategy(popolazioneSelezionata, archivio, pesi);
        ArrayList<ArmorSet> nuovaGenerazione = new ArrayList<>();
        while (nuovaGenerazione.size() < popolazioneSelezionata.size()) {
            ArmorSet genitore1 = popolazioneSelezionata.get(random.nextInt( origin: 0, popolazioneSelezionata.size()));
            ArmorSet genitore2 = popolazioneSelezionata.get(random.nextInt( origin: 0, popolazioneSelezionata.size()));
            while(genitore1.equals(genitore2)) {
                genitore2 = popolazioneSelezionata.get(random.nextInt( origin: 0, popolazioneSelezionata.size()));
            }
            ArmorSet figlio = crossoverParents(genitore1, genitore2);
            nuovaGenerazione.add(mutationSon(figlio, percentualeMutazione));
        }
        popolazione = nuovaGenerazione;
        popolazione.addAll(archivio);
        ArrayList<ArmorSet> temp = new ArrayList<>();
        for (ArmorSet armorSet : popolazione) {
            if (!temp.contains(armorSet)) {
                temp.add(armorSet);
            }
        }
        popolazione = temp;
    }
    return popolazione;
}
}
```

Figure 6: Implementazione del core dell'algoritmo

4 Testing e analisi finale

La nostra soluzione è stata prodotta in Java con un una GUI personalizzata con la quale l'utente può interagire inserendo il percorso del file da analizzare, la statistica per macro categoria che vuole prediligere in caso lo voglia e la percentuale di effettuare mutazioni. Con l'aiuto di **ChatGPT** abbiamo generato vari file contenti pezzi di armatura che rispecchiassero le indicazioni indicate nell'introduzione ogni account ha indicato il numero di pezzo di armatura che lo compongono (le tipologie generate non sono equamente distribuite). Dai test effettuati l'algoritmo restituisce sempre e con ogni account testato i migliori armor set possibile questo grazie alla grande generazione delle popolazione iniziale che ci permette di avere una buona fase di **Simulated Annealing** nella prima fase di costruzione, grazie alla riduzione iterativa "a cono" della popolazione nel corso delle generazioni fino ad un massimo di 10 elementi che con l'aiuto dell'archivio porta a valle sempre i migliori elementi trovati durante l'analisi, infine grazie alla funzione di fitness bilanciata in modo da far contare ovviamente la media totale delle statistiche senza lasciare però che gli armor set con statistiche superiori a 100 riescano a prevaricare nonostante un eventuale peso alterato della statistica in quanto portare a valle un individuo con una statistica a 120 implicherebbe togliere 20 punti alle altre due statistiche appartenenti alla macro categoria.

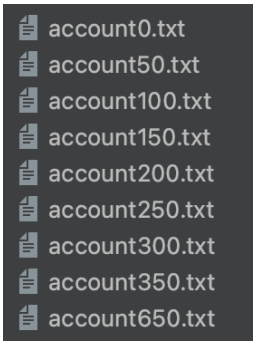

Figure 7: Account generati sul quale abbiamo testato l'algoritmo

5 Glossario

- **Armor set**: Un insieme di pezzi di armatura che, combinati, forniscono specifici vantaggi o statistiche a un personaggio in un gioco. Di solito composto da più elementi come casco, petto, guanti, pantaloni e stivali.
- **End game**: La fase avanzata di un gioco, dove i giocatori affrontano contenuti più difficili e complessi, tipicamente per ottenere ricompense di alto livello e perfezionare le proprie abilità o equipaggiamento.
- **Early game**: La fase iniziale di un gioco, in cui i giocatori sono impegnati con contenuti più semplici, imparano le meccaniche di base e iniziano a sviluppare il proprio personaggio o strategia.
- **Improvement**: Un processo o azione volta a migliorare una statistica, abilità, strategia o risultato all'interno di un gioco o di un algoritmo, per ottenere prestazioni migliori.
- **Underfitting**: Un problema in cui un modello o un algoritmo non riesce a catturare le relazioni nei dati, risultando in prestazioni scarse. In ambito di machine learning, è il contrario di overfitting e spesso si verifica quando il modello è troppo semplice.
- **Mating pool** :In un algoritmo genetico, è l'insieme di soluzioni selezionate per la riproduzione (o "crossover") con lo scopo di generare una nuova popolazione con caratteristiche migliori.
- **Core**: Il cuore o la parte principale di un sistema o di un'applicazione, spesso riferito ai componenti centrali di un software o a un elemento cruciale in un gioco.
- **ChatGPT**: Un modello di linguaggio sviluppato da OpenAI, progettato per comprendere e generare testo naturale, rispondendo a domande, fornendo spiegazioni e aiutando in diverse attività.
- **Java**: Un linguaggio di programmazione orientato agli oggetti, ampiamente utilizzato per lo sviluppo di applicazioni web, mobile e desktop, noto per la sua portabilità e robustezza.
- **GUI**: Acronimo di "Graphical User Interface" (Interfaccia Grafica Utente). Una modalità di interazione con un computer o un'applicazione che utilizza elementi visivi (come finestre, icone e bottoni) per facilitare l'interazione.
- **Simulated Annealing**: Un processo di ottimizzazione ispirato al processo di ricottura dei metalli. Cerca di trovare la soluzione ottimale introducendo perturbazioni casuali e riducendo gradualmente la probabilità di accettare soluzioni peggiori, per evitare minimi locali e trovare l'ottimo globale.