Politecnico di Torino

Dipartimento di Ingegneria Gestionale e della Produzione

Corso di Laurea Triennale in Ingegneria Gestionale

Classe L8 – Ingegneria dell'Informazione

A.A. 2023/2024

Generatore di build con gli emblemi di Pokémon Unite

Relatore

Prof. Fulvio Corno

Candidato

Andrea Marietti

S281374

Anno Accademico 2023-2024

INDICE

Capitolo 1 – Proposta di Progetto

1.1 - Studente proponente

s281374 Marietti Andrea

1.2 - Titolo della proposta

Generatore di build con gli emblemi di Pokémon Unite

1.3 - Descrizione del problema proposto

Pokémon Unite è un videogioco del genere MOBA in cui due squadre si scontrano per segnare il maggior numero di punti nelle basi avversarie. I giocatori hanno a disposizione dei personaggi standard per tutti, ma di cui possono aumentare (o diminuire) i 7 parametri tramite degli emblemi ottenibili con l'avanzare del livello.
Ogni emblema ha tre livelli: Bronzo, Argento e Oro. Più alto è il livello (dal metallo di valore inferiore al più prezioso) maggiori saranno le variazioni dei parametri, sia in positivo che in negativo. Tra la community, tuttavia, è caldamente sconsigliato utilizzare i livelli Bronzo e Argento, dati gli scarsi vantaggi che vengono forniti al giocatore.
La creazione di una combinazione di emblemi è complessa in quanto bisogna trovare la combinazione migliore dati 10 slot da riempire. L'applicazione avrà quindi lo scopo di facilitare la ricerca della combinazione date delle richieste del giocatore tramite un algoritmo ricorsivo.

1.4 - Descrizione della rilevanza gestionale del problema

I parametri che gli emblemi vanno a modificare hanno un grande impatto nel gioco a livello competitivo, dove anche una piccola variazione può segnare l’esito di uno scontro. Un personaggio improntato alla difesa, ad esempio, richiederà sicuramente elevati parametri difensivi, mentre potrà non necessitare di rilevante velocità di movimento.
Nell’ambito gestionale, la ricerca di una soluzione che massimizza un parametro non è da considerarsi sempre come la soluzione migliore, ma fornisce informazioni utili per avere una stima dei vantaggi e degli svantaggi che conseguono a una serie di scelte combinate tra loro.

1.5 - Descrizione dei data-set per la valutazione

Non esistendo un data-set scaricabile con tutte le informazioni, i dati sono ottenuti effettuando webscraping dal sito: https://game8.co/games/Pokémon-UNITE/archives/383688. Per effettuare questo procedimento è stata usata una estensione per browsers basati su Chromium chiamata "Web Scraper", che permette di estrarre le informazioni in automatico previa specifica degli elementi HTML in cui presenti e di inserirle in un file csv, facilmente convertibile in SQL.
Il database è composto da un'unica tabella formata da cinque colonne:
-Name: è nome del Pokémon che identifica l'emblema;
-Color: è il colore dell'emblema;
-Bronze, Silver, Gold: per ogni colonna sono indicati sia il parametro che viene aumentato sia quello viene diminuito.

1.6 - Descrizione preliminare degli algoritmi coinvolti

Per effettuare la ricerca della combinazione richiesta è necessario utilizzare un algoritmo ricorsivo con vari filtri al fine di soddisfare tutte le richieste.
Verranno proposte all'utente due opzioni per la creazione della build: automatica e manuale.
Nella prima, l'utente potrà scegliere il parametro da massimizzare e, se desidera, un parametro da non portare a valore negativo. In questo caso l'algoritmo ricorsivo partirà con un insieme parziale vuoto e dovrà, quindi, esplorare tutte le possibili combinazioni. Mentre nel secondo caso sarà possibile scegliere manualmente un numero a piacere di emblemi (fino a 10) e il parametro che si vuole massimizzare. L'insieme parziale partirà con al suo interno gli emblemi scelti e l'algoritmo si occuperà di completare nel modo migliore la combinazione.
Per permettere entrambe le opzioni senza intaccare l'esperienza dell'utente, si sfrutta il polimorfismo, ovvero verranno create due funzioni omonime che implementano un algoritmo ricorsivo. I due algoritmi presenteranno delle differenze dettate dalla necessità di rendere il processo di ricerca il più efficiente possibile.
Il problema principale è processare tutte le possibili combinazioni dati 233 emblemi e 10 posizioni. I tempi di esecuzione sarebbero molto grandi, data l'elevata quantità di risorse richieste. Per ovviare a questo problema si potrebbero usare dei parametri di soglia che devono necessariamente essere superati prima del completamente della combinazione, al fine di scartare tutte le combinazioni che non soddisfano gli standard richiesti. Per di creare varietà

negli output, non verranno usati tutti gli emblemi, ma un sottoinsieme di 30 elementi estratti casualmente; ciò ridurrà notevolmente anche i tempi di esecuzione.

1.7 - Descrizione preliminare delle funzionalità previste per l'applicazione software

L'applicazione avrà una interfaccia semplice e minimale. Saranno presenti due schede, una per l'opzione manuale e una per quella automatica, così da prevenire ogni possibile errore dell'utente. In entrambe le schede, sotto le opzioni di scelta, sarà presente un'area di testo in cui verrà mostrato l'output composto dall'elenco degli emblemi scelti e dal calcolo dei valori ottenuti.
Nella schermata per l'opzione automatica saranno presenti due comboBox; una per la scelta del parametro da massimizzare e una, il cui utilizzo è a discrezione dell'utente, per selezionare il parametro da non modificare sotto lo zero.
Nell'altra, invece, sarà sempre presente una comboBox per massimizzare una statistica, ma anche altre 10 per permettere all'utente di scegliere manualmente gli emblemi.
Viene ipotizzata la possibilità di utilizzare questa funzione anche solo per calcolare rapidamente il risultato ottenibile e quindi vengono lasciate 10 opzioni per l'inserimento dei dati, rendendo di fatto inutile la comboBox per il parametro da massimizzare.

Capitolo 2 – Descrizione dettagliata del problema affrontato

2.1 – Contesto aziendale

Il problema di trovare la migliore combinazione per massimizzare un parametro nel minore tempo possibile è un problema diffuso. All'aumentare della complessità, si predilige una soluzione buona alla soluzione migliore, poiché quest'ultima potrebbe non esistere o essere troppo onerosa in termini di risorse per essere ottenuta. Ad esempio, vengono utilizzate delle euristiche per gestire le ordinazioni di prodotti per rifornire il magazzino (per esempio l'euristica *Lot4Lot* o *Silver-Meal*); non sempre queste forniscono la soluzione migliore, ma forniscono una soluzione in modo efficiente.

2.2 – Analisi del problema affrontato

Nella ricerca della build che massimizza un parametro, quindi, non necessariamente si otterrà sempre la migliore combinazione, ma quella che, a parità di risorse, fornisce la soluzione migliore.

Se si prendessero tutti gli emblemi e si provassero tutte le combinazioni possibili al fine di trovare la migliore che massimizza il parametro "Attack", bisognerebbe utilizzare 233 emblemi e combinarli avendo a disposizione 10 slot (senza combinazioni identiche con solo ordine differente) e il numero di combinazioni possibili sarebbe troppo elevato per rendere i calcoli attuabili in tempi ragionevoli. Se a ciò si aggiunge la volontà di non ridurre un parametro secondario, ritenuto rilevante, i tempi si allungherebbero ulteriormente.

Gli input sono variabili e dipendono dalle richieste dell'utente; può venire passato un solo parametro o due, possono venire anche indicati degli emblemi manualmente. L'output sarà, in caso di successo, una lista di dieci emblemi coi cui creare la propria build in gioco. La criticità più rilevante è la quantità di possibili combinazioni da valutare; tuttavia, la creazione del software permetterà agli utenti di risparmiare una notevole quantità di tempo in fase di creazione in gioco.

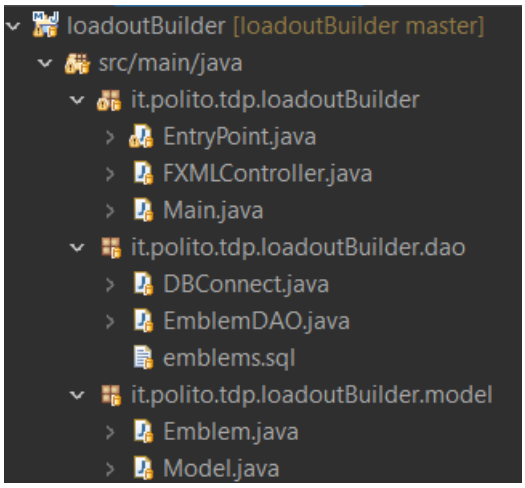
Capitolo 3 – Descrizione del dataset

Il dataset *emblems* è formato da un'unica tabella omonima, a sua volta divisa in cinque colonne.

campo	***descrizione***	***variabile***
Name	Nome del Pokémon rappresentato sull'emblema	VARCHAR
Color	Colore dell'emblema	VARCHAR
Bronze	Valori dei parametri per l'emblema livello *bronzo*	VARCHAR
Silver	Valori dei parametri per l'emblema livello *argento*	VARCHAR
Gold	Valori dei parametri per l'emblema livello *oro*	VARCHAR

Capitolo 4 – Descrizione delle strutture dati e algoritmi coinvolti

Il software è stato scritto in linguaggio Java, rispettando i patterns di programmazione MVC e DAO.



- Il package it.polito.tdp.loadoutBuilder contiene le classi: Main, classe principale del progetto, EntryPoint, che lancia l'interfaccia grafica, e FXMLController, che collega l'interfaccia grafica alla classe Model;
- Il package it.polito.tdp.loadoutBuilder.dao contiene le classi che consentono al software di accedere ai dati del database ed il file .sql relativo ad esso;
- In it.polito.tdp.loadoutBuilder.model è contenuta la classe (logica) che elabora tutti i dati, Model, e la classe (oggetto) Emblem.

Il costruttore della classe Model, ***Model()***, inizializza le variabili dichiarate in precedenza e chiama le funzioni ***creaMappe()*** e ***aggiungiSoglie(),*** che popolano le mappe con dei valori predefiniti.

```
public Model() {
    this.dao = new EmblemDAO();
    this.idMap = new HashMap<>();
    this.reverseIdMap = new HashMap<>();
    this.mappaSoglie = new HashMap<>();

    this.var_stop=true;
    this.creaMappe(this.idMap, this.reverseIdMap);
    this.aggiungiSoglie(this.mappaSoglie);

}
```

La funzione ***creaBuild()*** è utilizzata due volte, ma con parametri input diversi, sfruttando la proprietà del polimorfismo. È la funzione che si occupa di preparare i dati che saranno poi necessari durante la ricorsione, quali id dei parametri, l'inizializzazione della lista *parziale*, il richiamo della funzione ***randomPull()***, che serve a creare una lista di 30 emblemi casuali su cui lavorare e, nel caso di due parametri in input, controllare se il valore secondario è nullo e, quindi, richiamare la funzione ricorsiva corretta.

```java
/**
 *  Funzione per creare la build manuale
 *  Riceve il nome del parametro scelto e gli emblemi selezionati
 *
 * **/
public void creaBuild(String paramPrincipale, List<Emblem> scelti) {

    this.idParam = this.reverseIdMap.get(paramPrincipale);
    this.bestScore = 0.0;

    List<Emblem> parziale = new ArrayList<>(scelti);

    if (parziale.size() == 10) {
        this.buildFinale = new ArrayList<>(parziale);
        return;
    }else {

        List<Emblem> emblemi = new ArrayList<Emblem>();
        emblemi.addAll(this.randomPull(this.listaEmblemi));

        this.cerca(paramPrincipale,parziale,emblemi);
    }
}
```

Le funzioni ricorsive chiamate dentro *creaBuild()* si chiamano entrambe ***cerca()***, e proprio come in precedenza viene sfruttata la proprietà del polimorfismo con parametri di input diversi. Sebbene la logica ricorsiva sia la stessa, le due funzioni presentano delle differenze; queste sono una conseguenza della ottimizzazione che è stata fatta per rendere il processo il più rapido possibile.

La funzione ***cerca()*** che si occupa di lavorare con un solo parametro e, se inseriti, degli emblemi di partenza presenti nel *parziale*, presenta come condizione di terminazione della ricerca il raggiungimento di 10 elementi nella lista. Il *parziale* ottenuto viene controllato tramite dei filtri *if()* che si basano sul valore ottenuto del parametro che si vuole massimizzare, mediante la funzione ***calcolaAttuale()***, confrontandolo con la variabile *bestScore*, a cui è assegnato il valore migliore ottenuto in precedenza (0 in partenza). Se il risultato ottenuto è migliore del precedente viene aggiornata la lista *buildFinale* con i valori contenuti nel *parziale*.

La ricerca avviene iterando con un ciclo *for()* gli elementi della lista *emblemi*, ovvero i 30 elementi estratti causalmente i quali, se non contenuti nel *parziale*, vengono aggiunti e rimossi durante il processo ricorsivo. Per ridurre i tempi, è

stato inserito un filtro *if()*, con condizione *parziale.size()==8*, che controlla, tramite la funzione *calcolaAttuale()*, il valore del parametro principale e scarta l'ultimo emblema inserito (di fatto chiudendo il ramo che si stava esplorando) se il valore ottenuto è inferiore al valore minimo richiesto.

```
private void cerca(String paramPrincipale, List<Emblem> parziale, List<Emblem> emblemi) {

    // condizione di terminazione
    if(parziale.size()==10) {
        // calcola il valore del parametro nel parziale attuale
        double valoreAttualePrinc = this.calcolaAttuale(parziale,this.idParam);
        if(valoreAttualePrinc<=0) {
            return;
            }
        if(this.bestScore==0) {
            this.bestScore = valoreAttualePrinc;
            this.buildFinale = new ArrayList<>(parziale);
            }
        if(this.bestScore>=valoreAttualePrinc) {
            return;
        }else {
            this.bestScore = valoreAttualePrinc;
            this.buildFinale = new ArrayList<>(parziale);
            return;
            }
    }

    for(int i=0;i<emblemi.size();i++) {
        Emblem e = emblemi.get(i);
        if(!parziale.contains(e) ) {
            parziale.add(e);

            if(parziale.size()==8) {
                if(this.calcolaAttuale(parziale, this.idParam)<this.mappaSoglie.get(paramPrincipale)) {
                    parziale.remove(parziale.size()-1);
                    return;
                }
            }else {
                this.cerca(paramPrincipale,parziale, emblemi);
                parziale.remove(parziale.size()-1);
            }
        }
    }
```

L'altra funzione ***cerca()*** ha come condizione di terminazione una dimensione del *parziale* pari a 5, questa restrizione serve a garantire bassi tempi di esecuzione; verrà richiamata due volte dalla funzione *creaBuild()*, utilizzando due liste con parametri estratti diversi (maggiore varietà di output) e poi uniti insieme. Per consentire questa duplice invocazione, negli *if()* di controllo è presente, nel caso in cui la lista ottenuta contenga un valore del parametro principale migliore (e per quello secondario nullo o maggiore di 0), una varabile booleana *var_stop* che permette, alla funzione *creaBuild()* di decidere quali delle due liste parziali, che comporranno *buildFinale*, deve essere utilizzata e riempita.

Gli altri filtri presenti in questa parte di codice sono analoghi a quelli dell'altra funzione omonima, ovvero controllano che i valori siano accettabili, aggiornano

i valori migliori. A differenza della precedente, anche il parametro secondario è tenuto in considerazione nei controlli (nell'altra funzione non è presente).

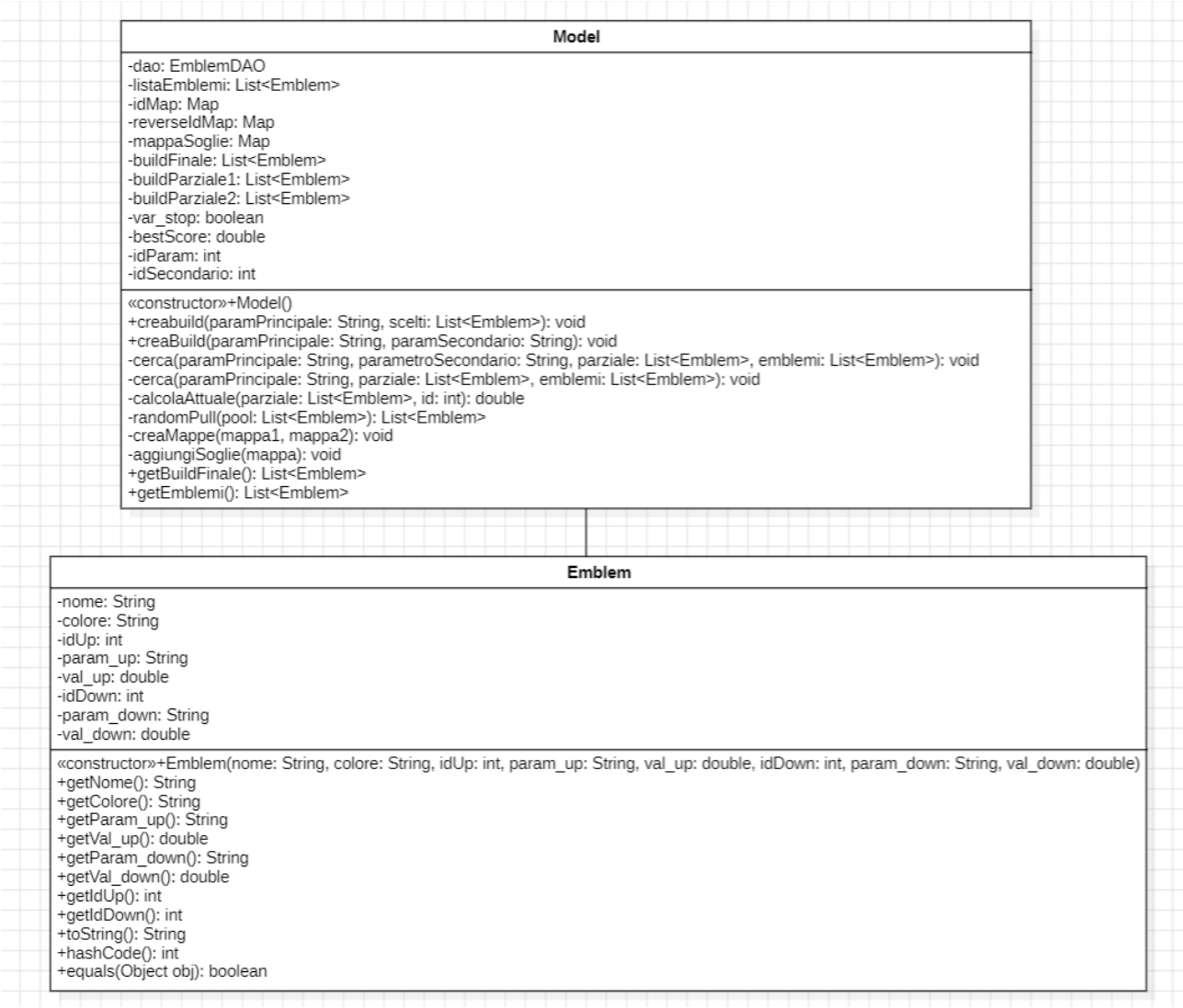
La parte di ricerca è formata da un ciclo *for()* e da un *if()* che controlla se l'emblema estratto durante l'iterazione è già contenuto nel parziale.

```
private void cerca(String paramPrincipale, String parametroSecondario, List<Emblem> parziale, List<Emblem> emblemi) {

    // condizione di terminazione
    if(parziale.size()==5) {
        // calcola il valore del parametro nel parziale attuale
        double valoreAttualePrinc = this.calcolaAttuale(parziale,this.idParam);
        double valoreAttualeSecond = this.calcolaAttuale(parziale,this.idSecondario);

        // caso valori troppo bassi
        if(valoreAttualePrinc<=0 || valoreAttualeSecond<=0) {
            return;
        }

        // caso bestscore migliore del valore attuale
        else if(this.bestScore>=valoreAttualePrinc) {
            return;
        }
        // caso inizializzatore -> setto il valore migliore SE il secondario è >=0
        else if(this.bestScore==0 && valoreAttualeSecond>=0) {
            this.bestScore = valoreAttualePrinc;
            if(this.var_stop) {
                this.buildParziale1 = new ArrayList<Emblem>(parziale);
            } else {
                this.buildParziale2 = new ArrayList<Emblem>(parziale);
            }
            return;
        }

        // caso valore attuale migliore del bestscore e secondario accettabile
        else if(this.bestScore<valoreAttualePrinc && valoreAttualeSecond>=0) {
            System.out.println("Valore attuale: "+valoreAttualePrinc+"  "+"Valore migliore: "+this.bestScore);
            this.bestScore = valoreAttualePrinc;

            if(this.var_stop) {
                this.buildParziale1 = new ArrayList<Emblem>(parziale);
            } else {
                this.buildParziale2 = new ArrayList<Emblem>(parziale);
            }
            return;
        }
    }

    for(int i=0;i<emblemi.size();i++) {
        Emblem e = emblemi.get(i);
        if(!parziale.contains(e) ) {
            parziale.add(e);
            this.cerca(paramPrincipale,parametroSecondario, parziale, emblemi);
            parziale.remove(parziale.size()-1);
            }
        }
```

```
private double calcolaAttuale(List<Emblem> parziale, int id) {

    double value=0.0;
    for(Emblem e : parziale) {
        if(e.getIdUp()==id) {
            value += e.getVal_up();
        }else if(e.getIdDown()==id) {
            value +=e.getVal_down();
        }
    }
    return value;
}
```

I parametro id è stato ricavato dalle mappe apposite nella funzione *creaBuild()* e corrisponde all'id del parametro di cui si vuole calcolare il valore complessivo dati gli emblemi del *parziale*.

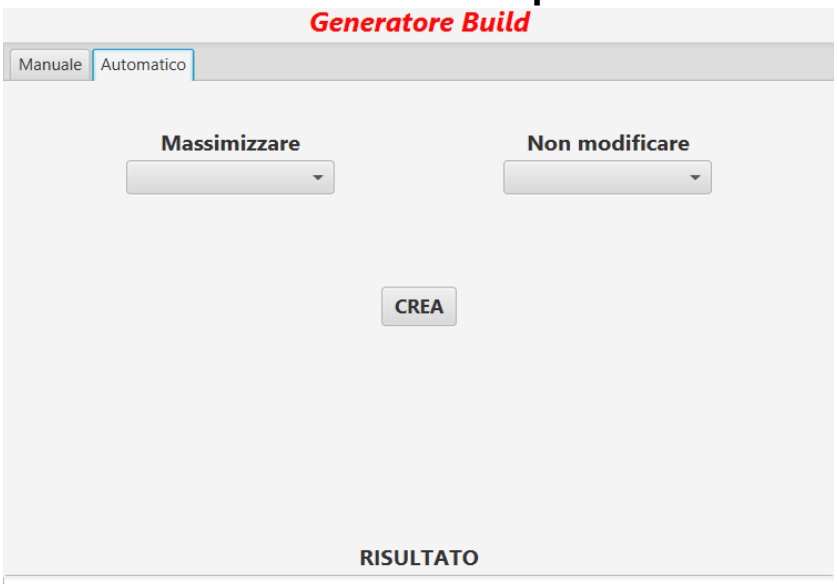
Capitolo 5 – Diagramma delle classi delle parti principali



Capitolo 6 – Videata dell'applicazione e link

6.1 – Creazione manuale con completamento

6.2 – Creazione automatica due parametri



6.3 – Video di presentazione e dimostrazione funzionamento
https://youtu.be/hGSeeiFzU6E

Capitolo 7 – Risultati sperimentali ottenuti

I test per i 3 possibili utilizzi sono stati effettuati, ognuno, su 30 esecuzioni usando come parametro principale *Attack*.

Opzione	Tempi medi	Note
Manuale (5 emblemi in input manuale *coerenti*)	≈1s	Nel 6% dei casi non è stato in grado di trovare una build
Automatico - 1 parametro	≈1s	Nel 3% dei casi non è stato in grado di trovare una build
Automatico - 2 parametri (HP secondario)	≈3s	Nel 17% dei casi non è stato in grado di trovare una build

Ekans Attack: 2.0 Defense: -5.0
Arbok Attack: 2.0 Sp.Def: -5.0
Rattata Defense: 5.0 HP: -50.0
Raticate Defense: 5.0 Sp.Atk: -3.0
Machamp Attack: 2.0 Sp.Atk: -3.0
Geodude Defense: 5.0 Sp.Def: -5.0
Fearow Attack: 2.0 Sp.Atk: -3.0
Quagsire HP: 50.0 Sp.Def: -5.0
Weepinbell Attack: 2.0 HP: -50.0
Phanpy Attack: 2.0 HP: -50.0

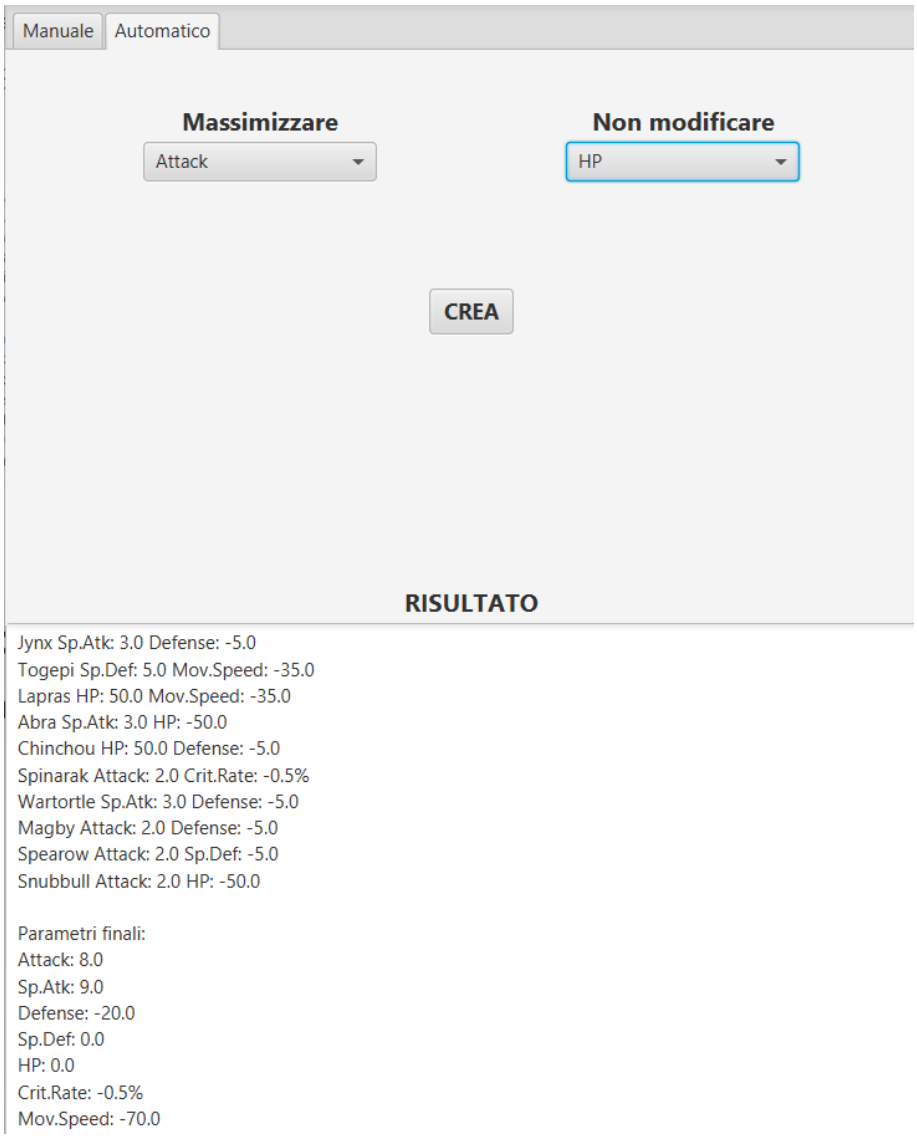
Parametri finali:
Attack: 12.0
Sp.Atk: -9.0
Defense: 10.0
Sp.Def: -15.0
HP: -100.0
Crit.Rate: 0.0%
Mov.Speed: 0.0

Parametri usati per il primo test

Jynx Sp.Atk: 3.0 Defense: -5.0
Togepi Sp.Def: 5.0 Mov.Speed: -35.0
Lapras HP: 50.0 Mov.Speed: -35.0
Abra Sp.Atk: 3.0 HP: -50.0
Chinchou HP: 50.0 Defense: -5.0
Spinarak Attack: 2.0 Crit.Rate: -0.5%
Wartortle Sp.Atk: 3.0 Defense: -5.0
Magby Attack: 2.0 Defense: -5.0
Spearow Attack: 2.0 Sp.Def: -5.0
Snubbull Attack: 2.0 HP: -50.0

Parametri finali:
Attack: 8.0
Sp.Atk: 9.0
Defense: -20.0
Sp.Def: 0.0
HP: 0.0
Crit.Rate: -0.5%
Mov.Speed: -70.0

Parametri usati per il secondo test



Parametri usati per il terzo test

Capitolo 8 – Valutazioni sui risultati ottenuti e considerazioni conclusive

Le build create dal programma hanno lo scopo di velocizzare ed aiutare l'utente in questo processo. Da questo punto di vista, le risposte fornite sono soddisfacenti. Nonostante l'estrazione casuale degli emblemi, le build che vengono create si adattano in modo coerente ai personaggi del gioco.

Il programma rimane limitato dai 30 emblemi estratti; se si potessero esplorare tutte le combinazioni, utilizzando quindi tutti gli emblemi, sicuramente si potrebbero ottenere risultati migliori, ma ad un costo di risorse computazionali molto più alto. Riguardo all'utilizzo delle risorse necessarie, tuttavia, i risultati sono stati molto soddisfacenti: nelle prime versioni, i tempi di esecuzione superavano i due minuti per la modalità manuale e non era possibile ottenere un output in tempi ragionevoli per quella automatica.

L'aggiornamento del dataset non influirebbe in alcun modo sul programma; inoltre, esso risulta facilmente aggiornabile, in caso di implementazione di nuove meccaniche di gioco.

In conclusione, l'applicazione rispetta le premesse con cui era stata ideata, discostandosi poco dal progetto originale.

Licenza