



Eötvös Loránd Tudományegyetem

Informatikai Kar

Komputeralgebra Tanszék

---

# Gépi tanulás a legnagyobb sebzés eléréséhez

Dr. Vatai Emil

Adjunktus

Komputeralgebra tanszék

ELTE IK

Szalóki Sándor

Programtervező informatikus BSc.

Budapest, 2018

# Tartalomjegyzék

<b>1. Bevezető</b>	<b>2</b>
<b>2. Felhasználói dokumentáció</b>	<b>3</b>
2.1. Feladat . . . . .	3
2.1.1. Játék modell . . . . .	3
2.1.2. Gépi tanulás . . . . .	4
2.1.3. Grafikus felület . . . . .	4
2.2. Módszerek . . . . .	4
2.2.1. Játék modell . . . . .	5
2.2.2. Gépi tanulás . . . . .	5
2.2.3. Grafikus felület . . . . .	5
2.3. Rendszer-követelmények, telepítés . . . . .	6
2.4. A program használata . . . . .	6
<b>3. Fejlesztői dokumentáció</b>	<b>11</b>
3.1. Specifikáció . . . . .	11
3.2. Módszerek . . . . .	11
3.2.1. Játék modell . . . . .	11
3.2.2. Gépi tanulás . . . . .	18
3.2.3. Grafikus felület . . . . .	22
3.3. Szerkezet . . . . .	24
3.3.1. Külső függőségek . . . . .	24
3.3.2. Adatszerkezetek . . . . .	24
3.3.3. Modulok . . . . .	29
3.3.4. Karakterek bővítése . . . . .	30
3.4. Tesztelés . . . . .	38
<b>4. Irodalomjegyzék</b>	<b>42</b>

# 1. Bevezető

Legfőbb célom a szakdolgozattal a gépi tanulás mélyebb megértése, tanulása egy olyan példán keresztül, amely előfordulhat problémaként a való életben is.

A probléma, amivel itt foglalkozunk kellően bonyolult ahhoz, hogy a megoldáshoz gépi tanulást kelljen alkalmazni és megoldása ne legyen triviális egy egyszerű algoritmussal.

Ahhoz, hogy a feladat több legyen, mint egy egyszerű út keresés, minden lépés hatással van a környezetére, illetve a környezettől függően egy-egy lépés hatása, értéke változhat.

Erre egy kellő nehézségű probléma a Final Fantasy XIV harcrendszerének szimulálása, majd ebben a szimulációban való tanulás.

## 2. Felhasználói dokumentáció

### 2.1. Feladat

#### 2.1.1. Játék modell

Első feladat a játéknak egy kellő pontosságú modellezése. A megoldás hatékonysága érdekében törekedni kell arra, hogy ne legyen felesleges adat, viszont minden, ami szükséges, az elérhető legyen. A harcrendszer szabályai pontosan meghatározhatóak, így a szimulációtól elvárható a pontos eredmény is. Két fő részre osztható a modell: Karakterek és ezek képességei.

**Karakterek** Minden karakternek három meghatározó eleme van:

**Gyorsaság** Ez adja meg a karakter globális időzítőjét

**Támadás** A képesség, ami a játékos irányítása nélkül történik a karakter gyorsaságától függően

**MP/TP** Magic Points, illetve Tactial Points, ezek a képességek erőforrásai

**Képességek** Egy képesség leírásához hét tulajdonság szükséges:

**Potenciál** A sebzés mértékének alapja

**Időzítő** Meghatározza, hogy az adott képesség a globális időzítőt, vagy a sajátját használja

**Erőforrás** A használatának ára MP, vagy TP-ben

**Combo** Képességek adott sorrendben használata esetén más-más lehet egy képesség hatása

**Típus** Fegyver, vagy varázslat

**Feltétel** A használatára vonatkozó feltétel

**Hatás** Használat után adott ideig hatással lehet a következő képességére, a hős állapotára, vagy adott időközönként plusz sebzést okozhat

A karakter tulajdonságai és képességeinek listája alapján egy adott képesség sorozatból készíthető egy szimuláció, amely megadja, hogy a sorozatot hány tizedmásodperc végrehajtani, illetve végrehajtás után ennek mekkora a teljes sebzési értéke.

Fontos, hogy a szimuláció bármely pontján tudjuk, hogy mi a játékos pontos állapota: Erőforrásainak, globális időzítőjének, képességeinek saját időzítőjének állapota, az aktív hatások, illetve ha van, akkor a Combo álltal meghatározott következő képesség.

Ez az állapot megfelel annak, amit a játékos játék közben a képernyőn láthatna.

### **2.1.2. Gépi tanulás**

A feladat ebben a szimulációban egy olyan képesség sorozat keresése, amely eléri a felhasználó által megadott időt, és ezalatt az idő alatt minél hatékonyabb legyen, azaz a legnagyobb sebzés elérésére törekszik.

A probléma tér elég nagy ahhoz, hogy egy egyszerű kereső algoritmus futási ideje túl nagy legyen. Ennek megoldásához megerősítéses tanulást használ a program.

Ehhez szükséges, hogy a játék állapotáról egy kellően pontos képet kapjunk, erre szolgál a játék szimulációjából tudható képszerű adat. Ezt az adatot állapotnak, a képességeket pedig akcióknak használhatjuk fel.

### **2.1.3. Grafikus felület**

Szükséges egy egyszerű grafikus felület, amely megkönnyíti a program használatát, átláthatóságát. A képességek sorozatának egy megjelenítését, a gépi tanulás eredményének grafikus feldolgozását.

Az algoritmus eredményessége jól bemutatatható néhány egyszerű grafikonon, például a tudás alapján egy sebzés-idő arányokat mutató grafikon, illetve egy, amely az elmúlt generációk tudását jelöli.

## **2.2. Módszerek**

A feladat megoldása funkcionális programozással történik.

### 2.2.1. Játék modell

Az említett két fő elemet (Karakterek és Képességek) megfelelő típusokkal vesszük, majd egy-egy tetszőleges karaktert a képességeivel együtt már egyszerűen előállíthatunk. Miután kellő adat birtokában vagyunk, a képességekből összeállíthatunk egy sorozatot, amit szimulálni szeretnénk.

A játék harcrendszerének szimulációja három lépésben történik.

1. A képességek sorozatát validálni kell. Ellenőrizni kell, hogy a képesség időzítője szerint lehetséges-e már a használata, rendelkezésre áll-e megfelelő típusú és mennyiségű erőforrás. Illetve néhány képesség rendelkezhet saját feltétellel, amit szintén ellenőrizni kell. Amennyiben minden rendelkezésre áll, a képességet meg kell tartani és a megfelelő hatásokat, illetve időzítőket be kell állítani.
2. A validált sorozaton végig kell menni és minden ponton az adott állapot szerint ki kell számolni, hogy abban a tizedmásodpercen mennyi sebzés történt, majd ezeket összegezni.

### 2.2.2. Gépi tanulás

A megfelelő algoritmus választásához több tényezőt is figyelembe kell venni. A játék harcrendszerének teljes működése kellően bonyolult, így a működést kihasználó algoritmusok használata bonyolult, ha lehetséges. Az akciók értéke az állapottól függ, illetve az akciók változtathatják az állapotot, emiatt az a feladat nem vezethető vissza egy egyszerű többkarú rabló problémára sem.

A probléma megoldásához Monte-Carlo-módszer használható. Az algoritmusnak nincs szüksége a háttérben működő szimuláció pontos leírására, elég ha tudunk adni egy pontos képet az állapotról, illetve az adott állapotban választott akciók értékéről.

A szimulációt olyan módon működik, hogy megfelelő képet tudjon adni az állapotról, az akció értéke pedig megadható az képesség sebzésének függvényében.

### 2.2.3. Grafikus felület

A program nem igényel bonyolult grafikus felületet. Egyszerű MVVM architektúra használatával épül fel WPF segítségével. Karakterenként a kilistázzuk a képességek képeit, amelyek majd választhatóak lesznek szimulációra. Lehetőséget kell adni az

eredmény sorozatnak kilistázására, illetve grafikonok rajzolására. Fontos, hogy a számolás a háttérben fusson, hiszen ez egy idő után nagy számításokkal járhat, ami zárolhatja az ablakot.

### 2.3. Rendszer-követelmények, telepítés

A program futtatásához Windows operációs rendszer szükséges. Telepítés nem szükséges, viszont a futtatható állomány mellé el kell helyezni a (programmal együtt kiadott) dinamikus könyvtárakat is.

**Ajánlott hardware:** Quad-Core 2.3 Ghz processzor, 2 GB RAM

**Software követelmény:** Microsoft .NET Framework 4.6.1

### 2.4. A program használata

A program indításához a futtatható állomány szükséges. Ennek indításával rögtön egy ablak fogad (1). Az ablakon található egy menüsor rendre Add, Start, illetve Cancel gombokkal. Alatta található a program futásához szükséges beállítások mezői. Egy listában találhatóak a karakterek és hozzájuk rendelt képességek listája. Az ablak közepén kerül majd megjelenítésre a tanult képesség sorozat, majd alatta a két grafikon. Továbbá található még egy státusz sor a képernyő legalján. Ebben a következő információk találhatóak:

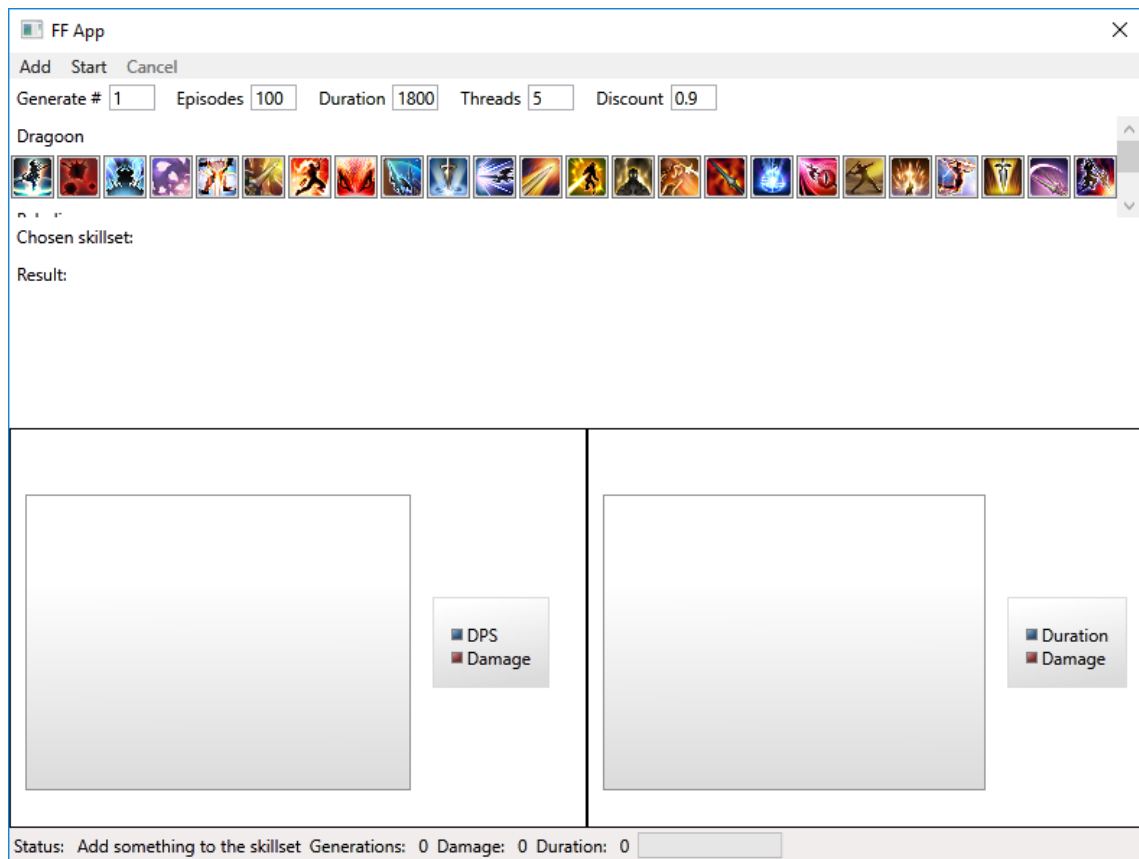
**Status** Rövid üzenet a program állapotáról

**Generations** A jelenlegi generáció száma

**Damage** A tanult képesség sorozat által elért sebzés értéke

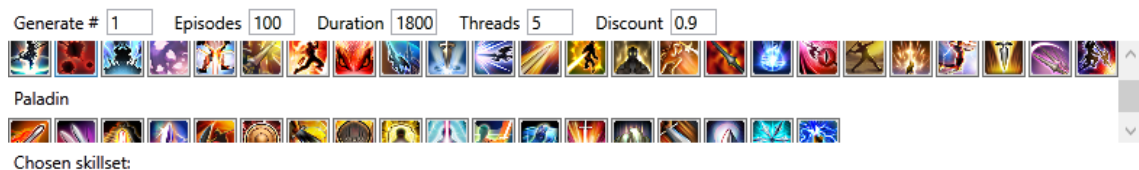
**Duration** A tanult képesség sorozat által elért idő tizedmásodpercben

**Bar** Töltő csík, amely mutatja, ha éppen számolás történik a háttérben



1. ábra. Kezdő képernyő

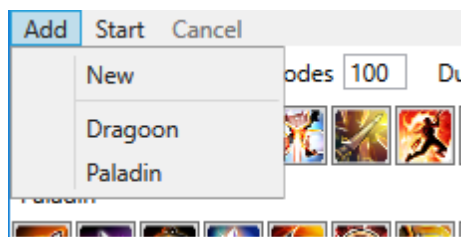
Mielőtt elindíthatnánk a tanulást, választanunk kell egyet a megadott karakterek képességlisái közül (2). Minden képesség lista fölött megtalálható a hozzá tartozó karakter neve. A kurzor egy képességen tartása után megjelenik a képesség neve, rövid leírása.



2. ábra. Lehetséges választható listák

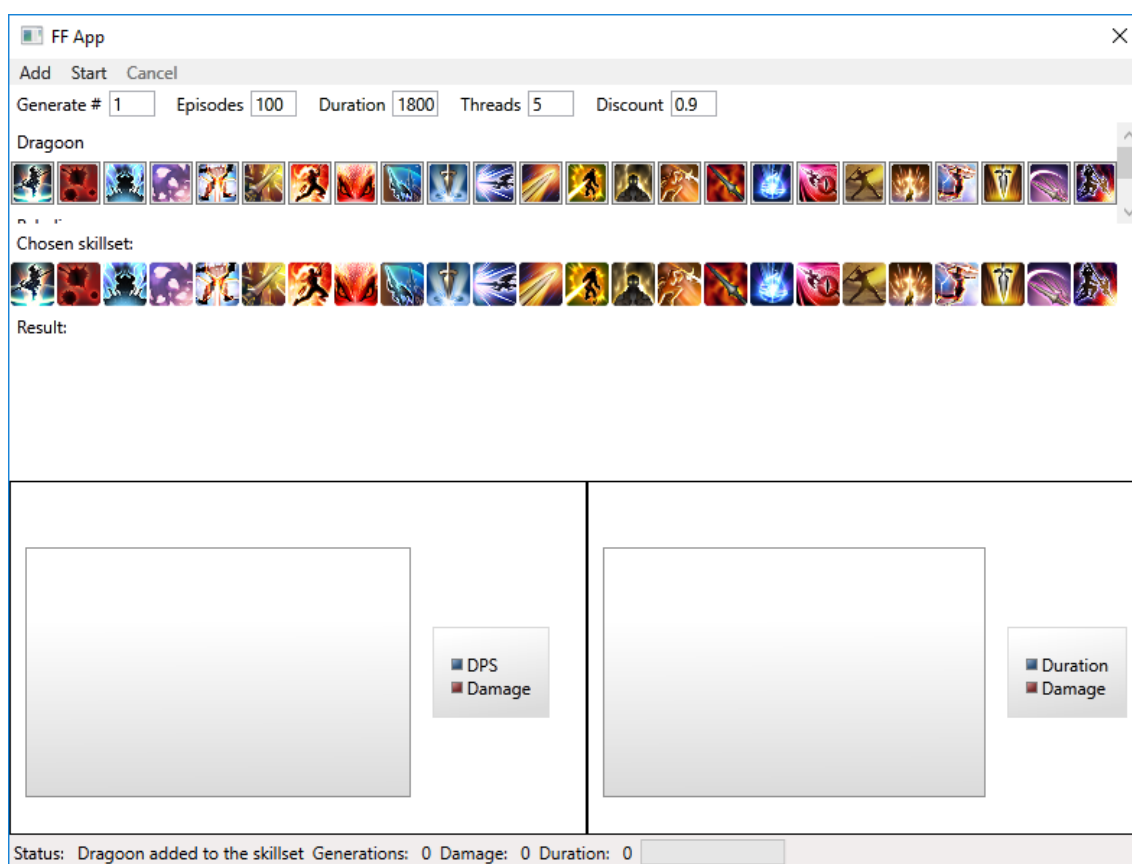
Ha megtaláltuk a megfelelő listát, akkor az Add menüpontban (3) választhatjuk ki a szükséges listát név alapján. Választás után szabadon cserélhetjük a listákat. A New gombra kattintva a programot alaphelyzetbe állíthatjuk.





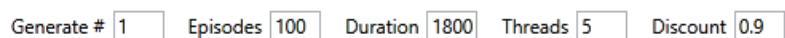
3. ábra. Lehetséges választható listák az Add menüpont alatt

A lista kiválasztása után erről információt kapunk, a választott lista megjelenik a Chosen skillset alatt (4). Ezen a ponton a tanulás elindítható a Start gombbal.



4. ábra. A választott lista megjelenik a képernyőn

Indítás előtt lehetőségünk van néhány beállítás változtatására (5).



5. ábra. Futási beállítások

A beállítások sorra:

**Generate** A kívánt generációk számát adhatjuk meg (Értéke: 1..)

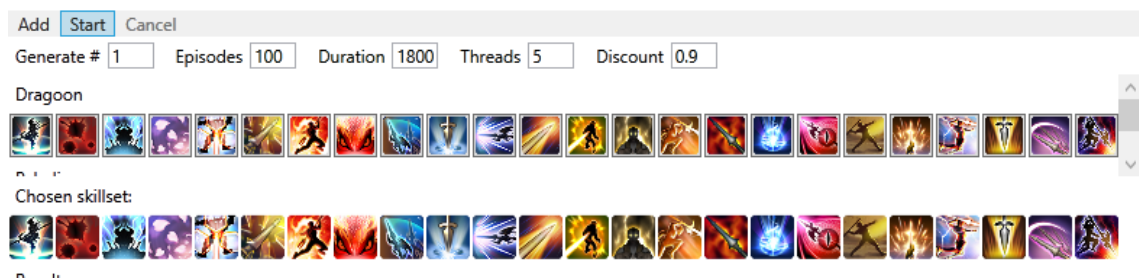
**Episodes** Generációnként létrehozott szimulációk száma (Értéke: 1..)

**Duration** Szimulációk hossza tizedmásodpercben (Értéke: 1..)

**Threads** Párhuzamosítás során használható magok száma (Értéke: 1..)

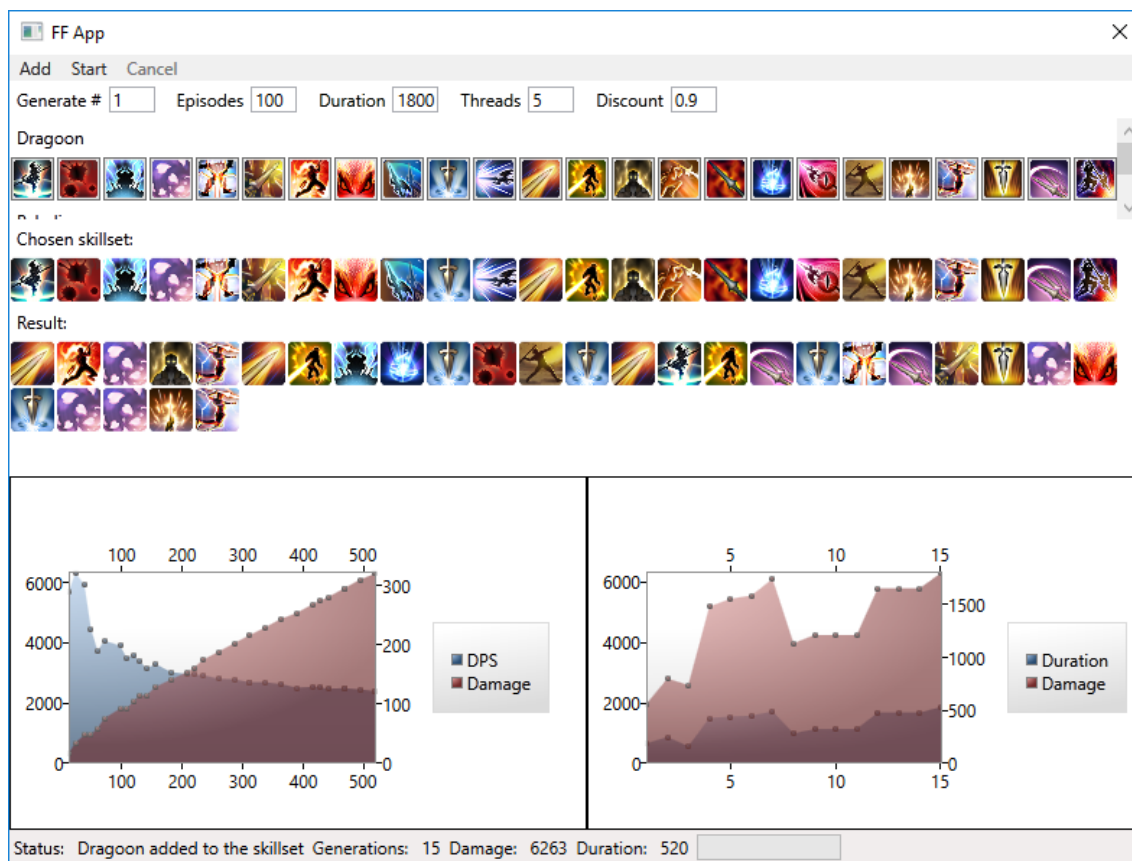
**Discount** Ez a szám jelöli egy-egy akció értékének hatásának arányát az előtte lévő akciókra (Értéke: 0.0 .. 1.0)

Amennyiben a beállításokat megfelelőnek találjuk, a Start gomb (6) lenyomásával elindítható a beállított számú generációk generálása.



6. ábra. Generálások indítása

A program a generációkat egyesével állítja elő és rajzolja őket ki a képre (7). Eközben a beállítási mezők, illetve az Add és Start gombok nem elérhetőek, viszont a generálás bármikor leállítható a Cancel gombra kattintva. Leállítást kérése esetén az éppen futó számolás még befejeződik, ennek eredménye megjelenik a képernyőn, majd az eddig letiltott mezők, gombok feloldásra kerülnek.



7. ábra. Eredmény 15 generáció futtatása után

Az eredményről három féle információt kapunk.

**Result** A jelenlegi tudás alapján generált képesség sorozat

**Bal grafikon** A Result-ban szereplő képesség sorozat szimulálása során kapott sebzés értékek másodpercre osztott értékei. Kék színnel és Damage névvel jelölt terület mutatja az adott másodpercben elért teljes sebzést. Piros színnel és DPS névvel jelölt terület mutatja az adott másodpercben elért sebzés és idő hányadosát.

**Jobb grafikon** Nyomon követhetjük minden generáció tudásának sebzés értékét

### 3. Fejlesztői dokumentáció

#### 3.1. Specifikáció

A feladat egy karakter és annak képességeinek listájából létrehozni a legnagyobb sebzés értékű képesség sorozatot. A megfelelő specifikáció és az egyszerűség érdekében az alábbi két típust vezetjük be a játék modelljéből:

$\mathcal{K}$  : Képesség

$\mathcal{H}$  : Karakter

Ezek segítségével már megfogalmazhatóak a programra a következő állítások:

$$\begin{aligned} A &= (\mathcal{H}_{Karakter} \times \mathcal{K}^*_{Képességek} \times \mathcal{K}^*_{Eredmény}) \\ B &= (\mathcal{H}_{Karakter'} \times \mathcal{K}^*_{Képességek'}) \\ Q &= (Karakter = Karakter' \wedge Képességek = Képességek') \\ R &= (Sebzés(Karakter', Eredmény) = \underset{ks \in \mathcal{P}(Képességek')}{argmax} Sebzés(Karakter', ks)) \\ \text{Sebzés: } \mathcal{H} \times \mathcal{K}^* &\rightarrow \mathbb{N} \end{aligned}$$

Az állapottér tartalmazza a karaktert és annak használható képességeit, illetve az eredményt. Paramétertérnek felvesszük a karakter és képességeinek listájának egy kezdő állapotát. Előfeltételként lerögzítjük a karakter és képességeinek listájának értékét. Utófeltétel, hogy az eredmény által meghatározott képesség sorozat sebzés értéke az adott karakterrel maximális. Ennek meghatározása a képességek számával arányosan exponenciálisan nő, ezért fontos egy olyan algoritmus használata, amely ezt hatékonyan képes kiszámolni.

#### 3.2. Módszerek

##### 3.2.1. Játék modell

A játék modellezéséhez szükséges a két alaptípus: Karakter és Képesség. Ezeket egy-egy a leírásnak megfelelő típussal könnyen le is írhatjuk.

**Skill** A Skill típus reprezentálja a Képességet. Ennek a leírása a következő:

---

```
type Skill =
{
    Name          : string
    Potency       : int
    Action        : ActionType
    CooldownType  : CooldownType
    CostType      : CostType
    CastType      : CastType
    SkillType     : SkillType
    Combo         : Combo list option
    Condition     : (Buff list -> bool) option
    ID            : int
}
```

---

Ahhoz, hogy a képesség modellje elég pontos legyen a játék szimulációjához, az alábbi típusokkal jellemezzük a tulajdonságait:

---

```
type CooldownType =
| GlobalCooldown
| OffGlobalCooldown of int
```

```
type CostType =
| Free
| MP of int
| TP of int
```

```
type CastType =
| Instant
| Time of int
```

```
type SkillType =
| Ability
| Weaponskill
| Spell
```

```

type ActionType =
| Damage of Buff list option
| DamageOverTime of potency:int * duration:int * Buff option
| Buff of Buff list
| Debuff of Buff
| Heal of Buff option
and BuffType =
| Job of (Job -> Job)
| Skill of (Skill -> Skill)
| BuffDuration of (Buff -> (Buff * int * int) option)
and Buff =
{
    Name      : string
    Effect     : BuffType
    Duration   : int
    Condition  : Condition
    Stacks     : int
    EndEffect  : (Buff -> Buff) option
    ID         : int
}
and Condition =
| NotLimited
| LimitedUses of uses:int * contidion:(Skill * Job * Buff list -> bool)
and Combo =
{
    Name           : string
    Target         : string
    Effect         : Skill -> Skill
    DistruptedByGCD : bool
}

```

---

**Job** A Job típus valósítja meg a karaktert. Ennek jellemzése hasonló módon történik. Tulajdonságai jól megfeleltethetők egy-egy típusnak:

---

```
type Job =
{
    Name      : string
    Speed     : int
    AutoAttack : Skill
    MaxMP     : int
    MP        : int
    TP        : int
}
```

---

**Rotation** Ezek segítségével a szimulációt reprezentáló típust nehézéges nélkül hozhatjuk létre a következő módon:

---

```
type GCD =
| Cooldown of int
| Available

type Tick =
{
    ActiveBuffs      : (Buff * int) list
    ActiveDebuffs    : (Buff * int) list
    OGCDTimers       : (Skill * int) list
    GCDtick          : GCD
    ActiveSkill       : Skill option
    ActiveDoTs        : (Skill * (int * int)) list
    ActiveCombo       : (Combo list * int) option
}

type Rotation =
| Rotation of (Tick list) * Job
```

---

A képesség sorozat végleges szimulációja a Rotation típusban jelenik meg. Az ebben szereplő Tick lista reprezentálja az idővonalat, egy Tick egy tizedmásodpercet jelöl. Ezzel a megoldással a szimuláció bármely pontján kérhetünk információt az adott tizedmásodperc adatairól. Tudjuk az éppen aktív pozitív, negatív hatásokat, minden képesség saját időzítőjét (amennyiben rendelkezik saját időzítővel), a globális időzítő állását, az éppen használt képességet (ha létezik), az aktív időszakos sebzéseket, illetve tudjuk, ha létezik éppen aktív combo (akár több is).

**Szimuláció** Ahhoz, hogy a Rotation Tick listájába adat kerüljön, szükséges egy függvény, amivel a szimulációhoz úgy adhatunk új képességeket, hogy az validálva legyen, illetve a lehető legnagyobb pontossággal tükrözze a való életben történő eseményeket. Ezt a feladatot a Rotation Add függvénye végzi el.

---

```
static member add (skill: Skill) (rotation: Rotation) =  
    match rotation with  
    | Rotation (ticks, job) ->  
        match ticks with  
        | [] -> failwith "Failed to add skill to the rotation"  
        | ticks ->  
            let ticks = ticks |> List.indexed  
            let lastindex, last = ticks |> List.last  
            let job = Rotation.updateJob job lastindex last  
            let originalSkill = skill  
            .  
            .  
            .
```

---

**Global Cooldown - GCD** Miután feloldottuk a típust és megszereztük az eddigi Tick listát, elkezdhetjük előállítani a következő Tick adatait. Először megnézzük, hogy az előző állapotban mi volt a globális időzítő értéke és beállítjuk az állapota alapján az megfelelő értékre.

**Skill** Következő lépésben az éppen használni kívánt képességet az aktív hatások és combo alapján frissítjük. Kihasználjuk, hogy a hatások típusa (Skill → Skill), így



ezek kompozíciójával egyszerűen számolhatunk.

**Combo** Miután az aktív képességet a hatások által meghatározott állapotára hoztuk, beállíthatjuk ez alapján az combo értékét is. Sok esetben egy képesség csak akkor kap combo tulajdonságot, ha már ő maga is az előző combo célja volt, ezért fontos, hogy ez a lépés a frissítés után történjen. Amennyiben a képesség nem rendelkezik ilyen tulajdonsággal, az előző állapotot kell tovább vinnünk. Ha az előző állapotban megadott Combo időzítője lejárt, ezt törölni kell.

**Buff** A Combo tulajdonsághoz hasonlóan a képességhez tartozó új hatások is más hatások eredménye képpen kerülhetnek csak elő. Ezeket a megfelelő időzítővel ellátva rakjuk bele hatások listájába. Ezeken a listákon végig kell menni és ellenőrizni, hogy az időzítőjük alapján törölni kell-e őket.

**Cast Time** Fontos, hogy egy képesség használata mennyi időt vesz igénybe, hiszen e szerint kell használat után további "üres" Tick-ekkel feltölteni a szimulációt, amivel garantáljuk az idő múlását.

**Damage over Time - DoT** Képességeknek lehetőségük van nem csak egyszeri sebzést okozni, de megadott ideig időközönkénti sebzést okozni. Ezeket az időközönkénti sebzéseket is külön tárolnunk kell egy listában. A listát ellenőrizni kell, amennyiben egy időzítő lejárt, a hozzá tartozó elemet törölni kell.

**Off Global Cooldown - OGCD** A saját időzítővel rendelkező képességek időzítőinek listáján is végig kell menni és a lejárt időzítőket törölni kell.

**Condition** Képességeket az időzítőjük típusától függően különböző feltételekkel tehetjük csak be a szimulációba. Közös működés történik amennyiben a képességhez szükséges erőforrás nem áll rendelkezésre (ekkor addig várunk, amíg megfelelő mennyiségű erőforrás keletkezik), illetve amikor a képesség saját feltétele (ha van) nem teljesül. Ha a saját feltétel nem teljesül, akkor a képesség használata nem szabályos, nem kerül be a szimulációba.

**GCD** Az egyszerűbb eset, amikor a globális időzítőt használja a képesség. Ekkor megvárjuk a globális időzítő végét, majd a képességet hozzá adjuk a szimulációhoz és újra indítjuk a globális időzítőt.

**OGCD** Ebben az esetben ellenőrizni kell, hogy a képesség saját időzítője már benne van-e az aktív időzítők listájában. Amennyiben benne van, a képességet nem tudjuk használni. Ellenkező esetben a képesség időzítőjét elindítjuk, a képességet hozzá adjuk a szimulációhoz.

**Üres Tick** Sokszor van szükség "üres" időtöltésre, ekkor a szimulációt ugyan ezek a szabályok alapján, de aktív képesség nélkül töljük fel megfelelő számú új Tick-el.

**Sebzés számlálás** Miután a képesség sorozatból megkaptuk a megfelelő validált szimulációt, lehetőségünk van ezen végig menni és megszámlolni a sorozat teljes sebzés értékét. A szimulált sorozatban minden képesség sebzés értéke már rendelkezik a hatások módosításaival, tehát ezt nem kell újra számolni. Minden karakter rendelkezik egy sebesség tulajdonsággal, illetve egy alapvető támadással, ezt itt kell nyomon követnünk. A sebesség által meghatározott időközönként ezt a támadást hozzá kell adnunk a szimulációban található sebzésekhez. Ez a támadás szintén élvezheti a hatások módosításait, tehát ezeket is itt kell hozzá adnunk az előbb használt módon. Található még a szimulációban időközönkénti sebzés is, amit szintén itt kell nyomon követnünk.

Ebből a három tényezőtől már egy számszerű sebzés érték rendelhető minden Tick-hez. Ezeket az értékeket összegezve megkapjuk a szimuláció teljes sebzési értékét.

---

```
let ToDamage (rotation: Rotation) : int =  
  match rotation with  
  | Rotation (rotation, job) ->  
    let time = (List.length rotation) / 10  
    rotation  
    |> List.indexed  
    |> List.fold (fun dps (index, tick) ->  
      .  
      .
```

### 3.2.2. Gépi tanulás

A választott algoritmus az Off-Policy Monte Carlo módszer az optimális  $\pi \approx \pi_*$  stratégia eléréséhez.

Az eredeti algoritmus ehhez (forrás: [1]) a következő:

Kezdetben  $\forall s \in \mathcal{S} : \forall a \in \mathcal{A}(s) :$

$Q(s, a) \in \mathbb{R}$  (tetszőleges)

$C(s, a) \leftarrow 0$

$\pi(s) \leftarrow \operatorname{argmax}_a Q(s, a)$

Végtelen ciklus (minden szimulációra) :

$b \leftarrow$  tetszőleges gyenge stratégia

Szimuláció létrehozása  $b$ -t használva:  $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

$W \leftarrow 1$

Ciklus a szimuláció minden lépésére,  $t = T - 1, T - 2, \dots, 0 :$

$G \leftarrow \gamma G + R_{t+1}$

$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$

$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$

$\pi(S_t) \leftarrow \operatorname{argmax}_a Q(S_t, a)$

Ha  $A_t \neq \pi(S_t)$ : Ciklus vége

$W \leftarrow W \frac{1}{b(A_t|S_t)}$

Mivel a program funkcionális nyelven készült, az algoritmust, ennek megfelelően kellett megváltoztatni. Az algoritmus megvalósításához szükséges néhány változtatás. A használt típusok és függvények leírásai:

$$\mathcal{S} : \mathbb{N}^* \times \mathbb{N}^* \times \mathbb{N}^* \times \text{string}^* \times \mathbb{N} \times \mathbb{N}$$

$$\mathcal{A} : \mathbb{N}$$

$$\mathcal{Q} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$$

$$\mathcal{C} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$$

$$\pi : \mathcal{S} \rightarrow \mathbb{R}^*$$

$$\mathcal{E} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}$$

$$\text{Episode} : \mathcal{E}^*$$

$$\text{GenerateEpisode} : \pi \rightarrow \text{Episode}$$

$$\text{CreateRandomPolicy} : \pi$$

$$\text{CreateGreedyPolicy} : \mathcal{Q} \rightarrow \pi$$

$$\text{CreateEpsilonGreedyPolicy} : \mathcal{Q} \times \varepsilon \rightarrow \pi$$

$$\text{MonteCarlo} : \mathcal{Q} \times \mathcal{C} \times \pi \rightarrow \mathcal{Q} \times \mathcal{C} \times \pi$$

A fenti típusok használata segítségével az algoritmus átírható egy annak megfelelő, funkcionális szemlélettel előállított függvényre. Ciklusok helyett foldl, illetve foldr műveletek használhatóak. Ahhoz, hogy a módosítások ne vesszenek el, mindezt, amit az eredeti függvény változóiban tárol, a fold függvények állapotában kell tárolnunk. A függvény kimenete az algoritmus által előállított Q, C függvények, illetve az ezekből előállítható mohó stratégia. A fenti algoritmus a  $\pi$  mohó stratégiát lépésenként állítja elő, ezt az alábbi függvényben csak a legvégén tesszük meg.

```

MonteCarlo :  $\mathcal{Q} \times \mathcal{C} \times \pi \rightarrow \mathcal{Q} \times \mathcal{C} \times \pi$ 
MonteCarlo Q C b =
  episodeFolder :  $\mathbb{R} \times \mathcal{Q} \times \mathcal{C} \times \mathbb{R} \rightarrow \mathcal{E} \rightarrow \mathbb{R} \times \mathcal{Q} \times \mathcal{C} \times \mathbb{R}$ 
  episodeFolder (G, Q, C, W) E =
    (S, A, R) = E
    g =  $\gamma G + R$ 
    c (S, A) =  $C(S, A) + W$ 
    q (S, A) =  $q(S, A) + \frac{W}{C(S, A)}[G - Q(S, A)]$ 
    w =  $W \frac{1}{b(S)|A}$ 
    (g, q, c, w)
  episodes : Episode*
  episodes = [GenerateEpisode b|x ← [1..]]
  episodesFolder :  $\mathcal{Q} \times \mathcal{C} \rightarrow \text{Episode} \rightarrow \mathcal{Q} \times \mathcal{C}$ 
  episodesFolder (Q, C) E =
    (g, q, c, w) = foldl episodeFolder (0, Q, C, 1) E
    (q, c)
  (q, c) = foldr episodesFolder (Q, C) episodes
  q, c, CreateGreedyPolicy q

```

Az itt található GenerateEpisode feladata az adott stratégia használatával létrehozni egy-egy szimulációt. Ehhez szükséges néhány új függvény a játék modell és a tanulás közötti átmenet miatt:

```

Random :  $\mathbb{R}^* \rightarrow \mathcal{A}$ 
ToSkill :  $\mathcal{A} \rightarrow \text{Skill}$ 
Add :  $\text{Skill} \times \text{Rotation} \rightarrow \text{Rotation}$ 
FromRotation :  $\text{Rotation} \rightarrow \mathcal{S}$ 
ToDamage :  $\text{Rotation} \rightarrow \mathbb{N}$ 
Time :  $\text{Rotation} \rightarrow \mathbb{N}$ 
EmptyRotation :  $\text{Rotation}$ 
Duration :  $\mathbb{N}$ 

```

Ezek használatával a szimuláció már egyszerűen létrehozható az alábbi módon:

```

GenerateEpisode :  $\pi \rightarrow Episode$ 
GenerateEpisode b =
  Generate :  $Rotation \times \mathcal{S} \times \mathbb{L} \times Episode$ 
  Generate Rot S l Ep =
    Ha l: Ep
    A :  $\mathcal{A}$ 
    A = Random (b S)
    (s, r, l, rot) :  $\mathcal{S} \times \mathbb{R} \times \mathbb{L} \times Rotation$ 
    (s, r, l, rot) =
      skill = ToSkill A
      rotation = Add skill Rot
      state = FromRotation rotation
      reward =
        Ha (Time Rot) = (Time rotation): -50
        Ha (Time rotation) > Duration: 1000
        Különben:  $\frac{ToDamage(rotation)}{Time(rotation)}$ 
      finished = (Time Rot) = (Time rotation)  $\vee$  (Time rotation) > Duration
      (state, reward, finished, rotation)
    Generate rot s l (Ep + (S, A, r))
  Generate EmptyRotation (FromRotation EmptyRotation) False []

```

**Reward** Ezen a ponton szükséges minden állapothoz egy jutalom választása. A folyamatos magas sebzésre törekvés érdekében a jutalom az aktuális sebzés értéke és az eltelt idő hányadosa. Szükséges a rossz működés büntetése is. Amennyiben az akciót követően a szimuláció ideje nem változik, az adott akció egy nem megengedett akció volt. Ha a szimuláció időtartama eléri az előre megjelölt cél időt, a jutalom sokszorosát kapja.

**Generációk** A hatékonyság növelése érdekében a MonteCarlo  $\pi$  értéke egy  $\varepsilon$ -mohó stratégia. További hatékonyság növelést érünk el, ha a generált szimulációk számát

korlátozzuk, majd a tanult tudás alapján az algoritmust újra indítjuk úgy, hogy az új  $\pi$ -t ez alapján a tudás alapján hozzuk létre egy már kisebb  $\varepsilon$  értékkel. Kezdetben az algoritmust egy igazságos véletlen szerű startégiával indítjuk el. Ezeket a meneteket hívjuk itt generációknak.

Egy generációs algoritmust ezek alapján a következő éppen fogalmazhatunk meg az eddig ismert típusok és függvények használatával:

GeneticMonteCarlo :  $\mathcal{Q} \times \mathcal{C} \times \pi \rightarrow \mathcal{Q} \times \mathcal{C} \times \pi$

GeneticMonteCarlo Q C b =

Folder :  $\mathcal{Q} \times \mathcal{C} \times \pi \rightarrow \mathbb{N} \rightarrow \mathcal{Q} \times \mathcal{C} \times \pi$

Folder (Q C b) i =

epsilon =  $\frac{1}{5 \times \ln i + 5}$

(q, c, pi) = MonteCarlo Q C b

(q, c, CreateEpsilonGreedyPolicy q epsilon)

(q, c, pi) = foldl Folder [1..] (Q, C, b)

q, c, CreateGreedyPolicy q

### 3.2.3. Grafikus felület

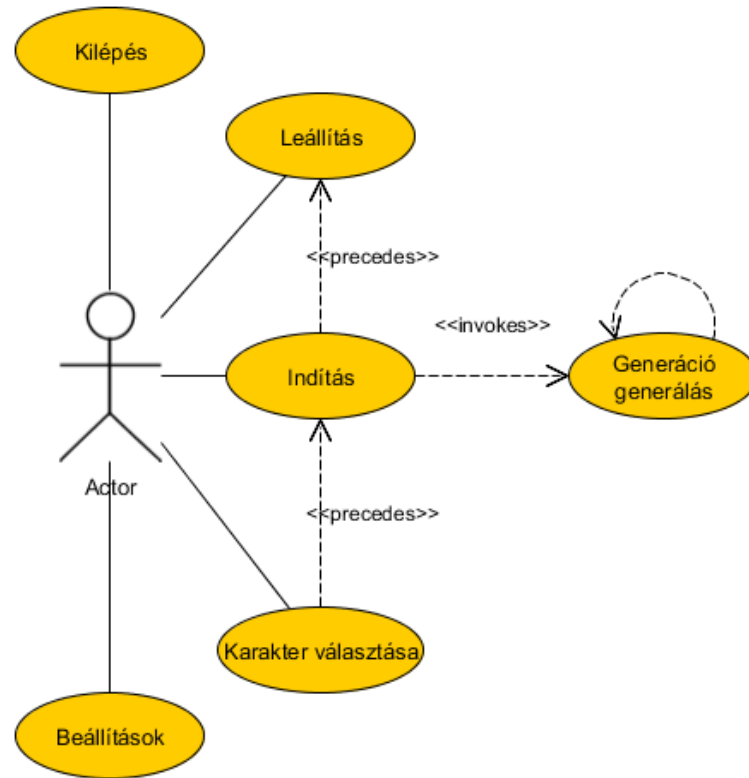
Grafikus felületet WPF segítségével könnyen adhatunk a számolások fölé. Szerkezetében MVVM architektúrát használunk, ahol a Model a játék modellje, illetve a gépi tanulás algoritmusai. View egy egyszerűbb ablakból áll, melyhez tartozik a számolásokat kezelő ViewModel.

A felhasználónak lehetősége van a programból kilépni, beállításokat állítani (8. oldal: 5). A tanítás indítása előtt szükséges egy karakter választása (7. oldal: 2). Ekkor az indítás elérhető válik. Indítást követően a program automatikusan elkezdi generálni a beállításokban megadott számú és tulajdonságú generációkat. Ez a folyamat bármikor megszakítható.

Minden generáció előállítása után a felület megjeleníti az adott generáció tudását. A tanult stratégia alapján létrehozunk egy képesség sorozatot, a továbbiakban ennek tulajdonságait jelenítjük meg. Ezt három féle módon tesszük meg:

1. A sorozat elemeit kilistázzuk a megfelelő képekkel

2. Egy grafikonon ábrázoljuk a sorozat minden állapotában elért sebzés értékét, illetve annak idővel vett hányadosát.
3. Egy másik grafikonon vezetjük az elmúlt generációk sebzés, illetve idő értékét



8. ábra. A grafikus felület Use-Case diagrammja

Mivel a számítások idő és erőforrás igénye magas lehet, ezeket egy másik, háttérben futó szálon indítjuk el. Ezt egy `BackgroundWorker` segítségével valósítjuk meg. Minden generáció előállítása végén ez a szál jelet küld a grafikus felület kezelő szálnak, illetve átadja a szükséges kiírandó információkat. Ekkor a grafikus felület kirajzolja a képernyőre a megfelelő új adatokat, a háttér szál tovább számolja az újabb generációkat (amennyiben ezekre még igény van). A háttér szál futása közben a beállítások mezői, illetve a leállítás gombjának kivételével a gombok le vannak tiltva, így a felhasználó nem tud beleszólni az éppen futó számolás alapvető beállításába.



### 3.3. Szerkezet

#### 3.3.1. Külső függőségek

A program két fő részre oszlik: Model (játék és gépi tanulás számítások), illetve Grafikus felület. Közös követelmény az F# fordító, a Microsoft.FSharp könyvtárak.

A program Model része további külső függőségekre nem támaszkodik.

A grafikus felület működéséhez szükséges az F# és WPF közötti átjáráshoz:

1. FsXaml.Wpf - XAML fájlok kezeléséhez F# kódból
2. FSharp.Desktop.UI, FSharp.ViewModel.Core - MVVM architektúrát segítő könyvtárak
3. System.Windows.Interactivity.WPF - Reaktív felülethez szükséges könyvtár
4. System.Windows.Controls.DataVisualization.Toolkit - Grafikonok rajzolása
5. Rx-Core, Rx-Interfaces, Rx-Linq, Rx-Main, Rx-PlatformServices - További könyvtárak reaktív felülethez

#### 3.3.2. Adatszerkezetek

**Játék modell** A játék modelljének megvalósításához szükséges fő típusok:

1. Skill - Képesség
2. Job - Karakter
3. Rotation - Szimuláció

Ezek megfelelő pontosságú leírásához szükség van további típusokra.

---

```
type CooldownType =  
    | GlobalCooldown  
    | OffGlobalCooldown of int  
  
type CostType =  
    | Free  
    | MP of int
```

```

| TP of int

type CastType =
| Instant
| Time of int

type SkillType =
| Ability
| Weaponskill
| Spell

type ActionType =
| Damage of Buff list option
| DamageOverTime of potency:int * duration:int * Buff option
| Buff of Buff list
| Debuff of Buff
| Heal of Buff option
and BuffType =
| Job of (Job -> Job)
| Skill of (Skill -> Skill)
| BuffDuration of (Buff -> (Buff * int * int) option)
and Buff =
{
    Name      : string
    Effect     : BuffType
    Duration   : int
    Condition  : Condition
    Stacks     : int
    EndEffect  : (Buff -> Buff) option
    ID         : int
}
and Condition =
| NotLimited
| LimitedUses of uses:int * contidion:(Skill * Job * Buff list -> bool)

```

```

and Combo =
{
    Name          : string
    Target        : string
    Effect        : Skill -> Skill
    DistruptedByGCD : bool
}

and Skill =
{
    Name          : string
    Potency       : int
    Action        : ActionType
    CooldownType  : CooldownType
    CostType      : CostType
    CastType      : CastType
    SkillType     : SkillType
    Combo         : Combo list option
    Condition     : (Buff list -> bool) option
    ID            : int
}

and Job =
{
    Name          : string
    Speed         : int
    AutoAttack    : Skill
    MaxMP         : int
    MP            : int
    TP            : int
}

type GCD =
| Cooldown of int
| Available

```

```

type Tick =
{
    ActiveBuffs      : (Buff * Int) list
    ActiveDebuffs    : (Buff * Int) list
    OGCDTimers       : (Skill * Int) list
    GCDtick          : GCD
    ActiveSkill       : Skill option
    ActiveDoTs        : (Skill * (Int * Int)) list
    ActiveCombo       : (Combo list * Int) option
}

type Rotation =
| Rotation of (Tick list) * Job

```

---

**Gépi tanulás** A gépi tanuláshoz egy további típusra van szükség. A State típus jelöli az állapotot, amit a szimuláció egy pontjából kapunk megfigyelés után:

```

type State =
(Int list) * (Int list) * (Int list) * (string list) * Int * Int

```

---

Megfigyelés a következő képpen történik: Lekérdezzük az aktuális Tick adatait, majd ezeket a gyorsabb számolás érdekében egyszerűbb, de egyértelmű típusokkal reprezentáljuk. Ahol lehetséges egy teljes bonyolult típus helyett csak a hozzá tartozó ID értéket tároljuk, ha ez nem létezik, akkor a nevét. Az erőforrások értékét átalakítás nélkül tárolhatjuk.

```

let fromRotation (r: Rotation) : State =
    match r with
    | Rotation (ticks, j) ->
        let last = List.last ticks
        let buffs =
            last.ActiveBuffs
            |> List.map (fun (b, _) -> b.ID)
            |> List.sort
        let ogcd =

```

```

last.OGCDTimers
|> List.map (fun (b, _) -> b.ID)
|> List.sort

let dots =
    last.ActiveDoTs
    |> List.map (fun (b, _) -> b.ID)
    |> List.sort

let combo =
    match last.ActiveCombo with
    | None -> []
    | Some combos -> (fst combos) |> List.map (fun c -> c.Target)
        |> List.sort

(bufffs, ogcd, dots, combo, j.TP, j.MP)

```

---

További hatékonyság növelés érdekében az (Állapot  $\rightarrow$  Értékek) tárolását kettő szintes Map típusban valósítjuk meg. Ez az Állapot (State) bonyolult típus a gyors összehasonlításhoz, de ezen segíthetünk, ha egy hash függvény alapján más-más Map szerkezetekben tároljuk őket. Erre a célra egy megfelelő hash függvény (a legtöbbet változó) első két lista hosszának összege:

```

let toKey (s: State) : int =
    let bufffs, ogcd, _, _, _, _ = s
    bufffs.Length + ogcd.Length

```

---

**Grafikus felület** Szükséges a képességek kezeléséhez egy megfelelő View Model. Ezt valósítja meg a SkillView típus:

```

type SkillView (s: FFDPSMeter.Model.Skill, place: int) =
    member this.Name = s.Name
    member this.Potency = s.Potency
    member this.ID = s.ID
    member this.Image = "/Resources/Skills/" + (s.Name.Replace(" ",
        "")) + ".png"
    member this.Place = place

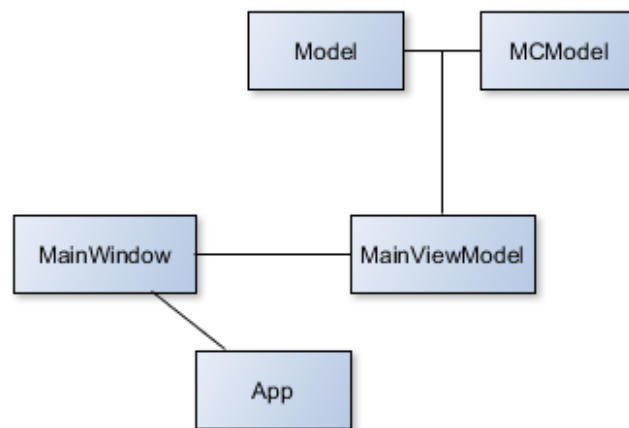
```

```
member this.ToMLModel (s: FFDPSMeter.Model.Skill list) =  
    s  
    |> List.find (fun skill -> skill.ID = this.ID)
```

---

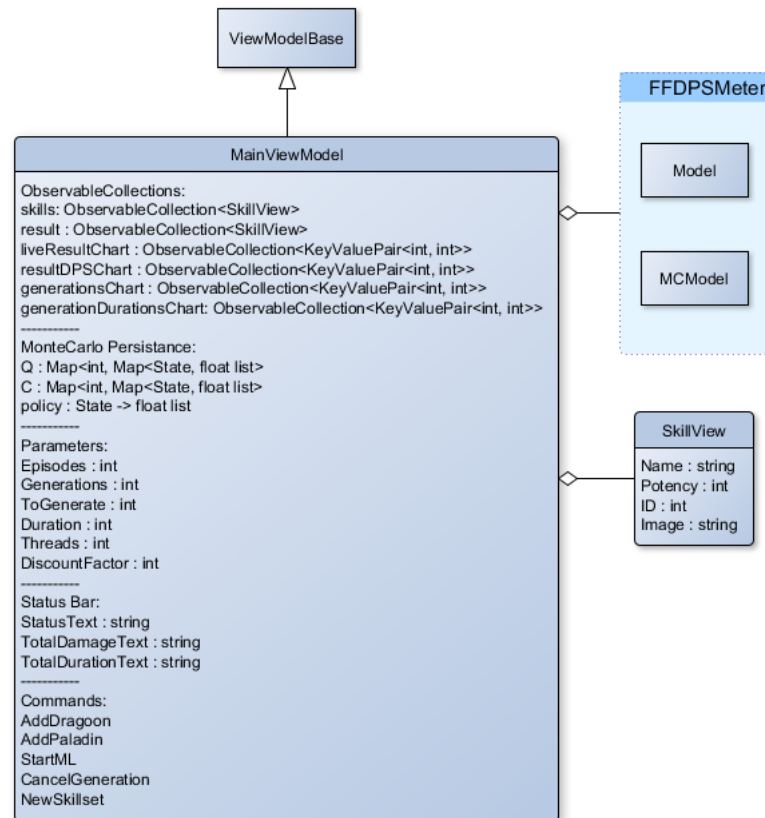
A reaktív felület elérése érdekében a ViewModel-ben szereplő adatok ObservableCollection-ben találhatóak.

### 3.3.3. Modulok



9. ábra. Modulok kapcsolata

A modulok helyes együttműködését az App biztosítja. Az egyetlen ablak felületet a MainWindow valósítja meg, illetve az ehhez tartozó MainViewModel ().



10. ábra. View Model felépítése

### 3.3.4. Karakterek bővítése

További képességek, karakterek hozzáadása a típusrendszer megszorításaival egyszerűen megoldható.

Egy karaktert az átláthatóság kedvéért egy saját modulba rendezünk, ekkor a modul neve legyen a karakter neve.

**Job** Első lépésként létre kell hozni a karakter típusával (Job) magát a karaktert, ahol a következő értékeket kell megadni:

---

```

type Job =
{
    Name      : string
    Speed     : int
    AutoAttack : Skill
    MaxMP     : int
  }
  
```

```

    MP          : int
    TP          : int
}

```

---

**Name**            A karakter neve

**Speed**           Gyorsasága tizedmásodpercben

**AutoAttack**    Alapvető támadás (Skill), amit automatikusan Speed időközönként alkalmaz

**MaxMP**           Magic Points maximális értéke

**MP, TP**           MP, illetve TP kezdő értéke (TP maximális értéke mindig 1000)

**Skill**    Képességek megadásához szükséges ismerni a Skill típust is:

---

```

type Skill =
{
    Name          : string
    Potency       : int
    Action        : ActionType
    CooldownType  : CooldownType
    CostType      : CostType
    CastType      : CastType
    SkillType     : SkillType
    Combo         : Combo list option
    Condition     : (Buff list -> bool) option
    ID            : int
}

```

---

**Name**            A képesség neve

**Potency**        Sebzési értéke

**Action**         Akció típusa

**CooldownType**   Időzítő típusa



<b>CostType</b>	Erőforrás típusa
<b>CastType</b>	Használatához szükséges idő
<b>SkillType</b>	Képesség típusa
<b>Combo</b>	Lehetséges Combo listája
<b>Condition</b>	Használat feltétele
<b>ID</b>	Képesség egyéni száma

Ez a típus több másik típusra is támaszkodik, a típus teljes megértéséhez szükséges ezen típusok ismerete is.

**ActionType** Képesség akció típusa szerint más-más hatással lehet a szimulációra.

---

```

type ActionType =
| Damage of Buff list option
| DamageOverTime of potency:int * duration:int * Buff option
| Buff of Buff list
| Debuff of Buff
| Heal of Buff option

```

---

**Damage** Egyszerű sebzés, rendelkezhet további hatásokkal (Buff)

**DamageOverTime** Egyszerű sebzés mellett adott ideig időközönkénti sebzést is okozhat, rendelkezhet további hatásokkal (Buff)

**Buff** Egyszerű hatások (Buff) hozzáadása

**Debuff** Hasonló az előző Buff típushoz, de ez negatív hatásokat tartalmaz

**Heal** Karakter életének növelése (jelenlegi szimulációban ezzel nem foglalkozunk)

**Buff** Az előző típus leírásához szükséges a Buff típus ismerete is. Ez felelős egy-egy hatás megvalósításáért.

---

```
type Buff =  
  {  
    Name      : string  
    Effect     : BuffType  
    Duration   : int  
    Condition  : Condition  
    Stacks     : int  
    EndEffect  : (Buff -> Buff) option  
    ID         : int  
  }
```

---

<b>Name</b>	A hatás neve
<b>Effect</b>	Hatás típusa szerint különböző dolgokat változtathat meg
<b>Duration</b>	Hatás időtartama
<b>Condition</b>	Feltétel a hatás időtartamára
<b>Stacks</b>	Egy-egy hatásnál lehetőség van belső számláló használatára
<b>EndEffect</b>	Lehetőség az időtartam végén, vagy feltétel sérülés esetén átalakítani a hatást
<b>ID</b>	Egyéni azonosító szám

Ehhez továbbá szükséges az alábbi két típus:

---

```
type BuffType =  
  | Job of (Job -> Job)  
  | Skill of (Skill -> Skill)  
  | BuffDuration of (Buff -> (Buff * int * int) option)
```

---

<b>Job</b>	Karakterre vonatkozó módosítások
<b>Skill</b>	Képességre vonatkozó módosítások

**BuffDuration** Másik hatásra, vagy annak időtartamára vonatkozó módosítások  
(Ennek eredménye az új hatás, az időtartam növelésének mértéke, illetve a maximális időtartam)

---

```
and Condition =  
  | NotLimited  
  | LimitedUses of uses:int * contidion:(Skill * Job * Buff list -> bool)
```

---

**NotLimited** Nincs feltétel

**LimitedUses** Megadható egy feltétel függvény, illetve egy számláló, ami egyel csökken, ha a feltétel igaz. Amennyiben a számláló eléri a nullát, a hatásnak vége van

**CooldownType** Képességek időzítőjük szerint két csoportba sorolhatóak: globális, illetve saját időzítővel rendelkező képességek.

---

```
type CooldownType =  
  | GlobalCooldown  
  | OffGlobalCooldown of int
```

---

**GlobalCooldown** Globális időzítő

**OffGlobalCooldown** Saját időzítő és annak mértéke tizedmásodpercben

**CostType** Erőforrás használata alapján három csoport létezik: ingyenes, MP és TP.

---

```
type CostType =  
  | Free  
  | MP of int  
  | TP of int
```

---

**Free** Ingyenes

**MP** Magic Points használata és annak mértéke

**TP** Tactical Points használata és annak mértéke

**CastType** Egy képesség használata időt vesz igénybe. Ez alapján két csoport létezik: azonnali használat, vagy várakozni kell a használata előtt.

---

```
type CastType =  
  | Instant  
  | Time of int
```

---

**Instant** Azonnali használat (1 másodpercet vesz igénybe)

**Time** Adott tizedmásodperc várakozás a képesség használata előtt

**SkillType** Képességeket három csoportba lehet sorolni típusuk szerint: tehetség, fegyver alapú, vagy varázslat.

---

```
type SkillType =  
  | Ability  
  | Weaponskill  
  | Spell
```

---

**Ability** Tehetség

**Weaponskill** Fegyver alapú

**Spell** Varázslat

**Combo** Képességek adott sorrendben használata esetén változtathatunk a következő képesség tulajdonságain. Ezt valósítja meg a Combo:

---

```
type Combo =  
  {  
    Name          : string  
    Target        : string  
    Effect        : Skill -> Skill  
    DistruptedByGCD : bool  
  }
```

---

**Name** Combo neve

**Target** A sorrendben következő képesség neve

**Effect** Target-ben szereplő képesség módosítása

**DistruptedByGCD** Ha az értéke igaz, minde globális időzítővel rendelkező képesség megszakítja a Combo-t

**Saját karakter befejezése** A típusok megértése után már mindent tudunk egy új karakter megírásához:

---

```
module UjKarakter =
  let Karakter : Job =
    let AutoAttack : Skill =
      {
        Name = "Auto Attack"
        Potency = 72
        Action = ActionType.Damage None
        CooldownType = CooldownType.OffGlobalCooldown 30
        CostType = CostType.Free
        CastType = CastType.Instant
        SkillType = SkillType.Weaponskill
        Combo = None
        Condition = None
        ID = 0
      }
    {
      Name      = "Uj Karakter"
      Speed     = 25
      AutoAttack = AutoAttack
      MaxMP     = 5000
      MP        = 5000
      TP        = 1000
    }

  let Combo1 : Skill =
    let combo : Combo =
```

```

{
    Name = "Combo 1"
    Target = "Combo 2"
    Effect = (fun (s: Skill) -> {s with Potency = int ((float
        s.Potency) * 1.1)}))
    DistruptedByGCD = true
}

{
    Name          = "Combo 1"
    Potency       = 40
    Action        = ActionType.Damage None
    CooldownType  = CooldownType.GlobalCooldown
    CostType      = CostType.TP 100
    CastType      = CastType.Instant
    SkillType     = SkillType.WeaponSkill
    Combo         = Some [combo]
    Condition     = None
    ID = 1
}

let Combo2 : Skill =
    let buff =
        {
            Name = "Buff"
            Effect = Skill (fun (s: Skill) -> {s with Potency =
                s.Potency + 50}))
            Duration = 150
            Condition = Condition.NotLimited
            Stacks = 0
            EndEffect = None
            ID = 2
        }
    {
        Name          = "Combo 2"

```

```

    Potency          = 70
    Action           = ActionType.Damage (Some [buff])
    CooldownType     = CooldownType.GlobalCooldown
    CostType         = CostType.MP 500
    CastType         = CastType.Time 150
    SkillType        = SkillType.Spell
    Combo            = None
    Condition        = None
    ID = 2
}

let Skillset : Skill =
[
    Combo1
    Combo2
]

```

---

Ekkor az `UjKarakter.Karakter`, illetve az `UjKarakter.Skillset` értékekkel már könnyedén hozhatunk létre új szimulációkat.

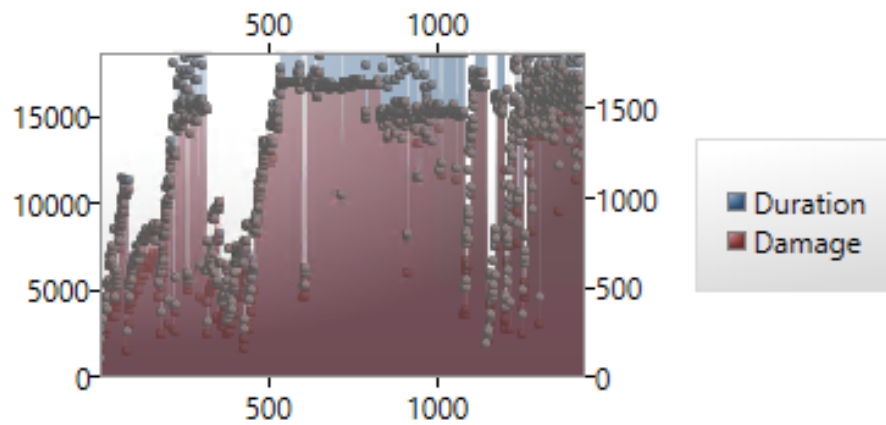
### 3.4. Tesztelés

A program tesztelése a gépi tanulás sajátosságai miatt nehéz. Az algoritmusok alapja a véletlenszerű szám generálás, illetve egy valószínűség szerinti akció választás. A végleges algoritmus megengedi, hogy belső paramétereit futás közben is változtathassuk, így ezek hatékonyságának tesztelése lehetséges.

**Reward** Nehéz kérdés a jutalom függvény választása az adott akciókhoz. Jelenleg a jutalom az akció által létrehozott állapot sebzés értékének és idejének hányadosa. ezzel elérhető, hogy az algoritmus egy stabilis sebzés/másodperc arányra törekedjen. A büntetés feltétele és mértéke is kérdéses ilyenkor. A jelenlegi jutalom-büntetés rendszer egy működő, bár nem a legoptimálisabb rendszer. Ennek teszteléséhez további kutatás szükséges.

**Discount Factor** Az algoritmusban egy-egy akció értékének összegzésénél jelen van egy  $\gamma$  szorzó. Ez felelős azért, hogy ha egy szimuláció végtelen hosszú, akkor is egy konvergens sorozatot kapjunk az értékekből. Végtes sorozat esetében ezzel az értékkel adhatjuk meg, hogy egy-egy akció értéke mennyire legyen hatással az előtte lévő akciók értékére.  $\gamma$  optimális értéke is kérdéses lehet.

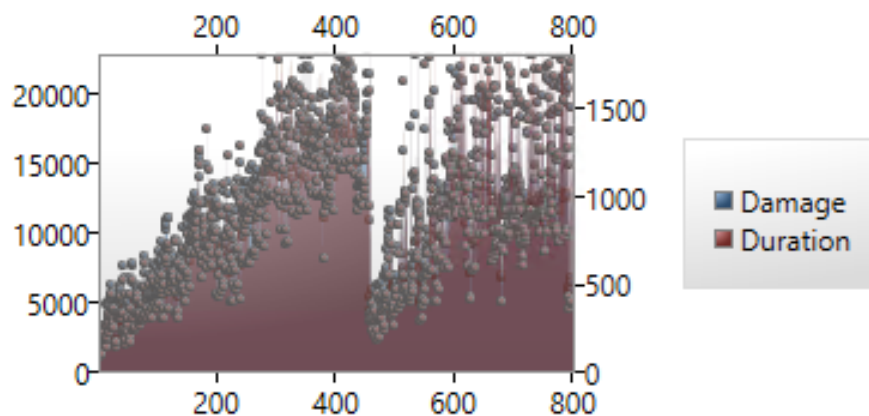
$\gamma = 0.9$  Amikor  $\gamma$  értéke közel van egyhez, látható, hogy az algoritmus sok változtatást nem végez a már megtanult lépéseken. Amikor egy látszólag jó útra tér, annak értéke magas. Ahogy látható az ábrán (11) a tudása hullámokban növekszik, amíg el nem ér egy stabil állapotot, ahol elkezdődik a nagyobb felfedezés.



11. ábra. Generációk grafikonja ( $\gamma = 0.9$ , 1400 generáció, 400 episode/generáció, 180 másodperc)

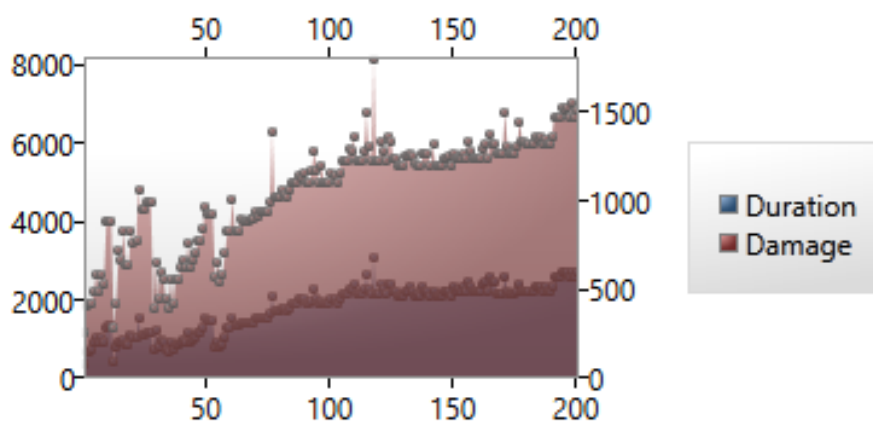
$\gamma = 0.1$  Ha  $\gamma$  értéke kicsi, egy-egy akció értéke szinte csak a közvetlen előtte lévő akció értékére van hatással. Ilyenkor a generációk grafikonja jelentősen változik (12).



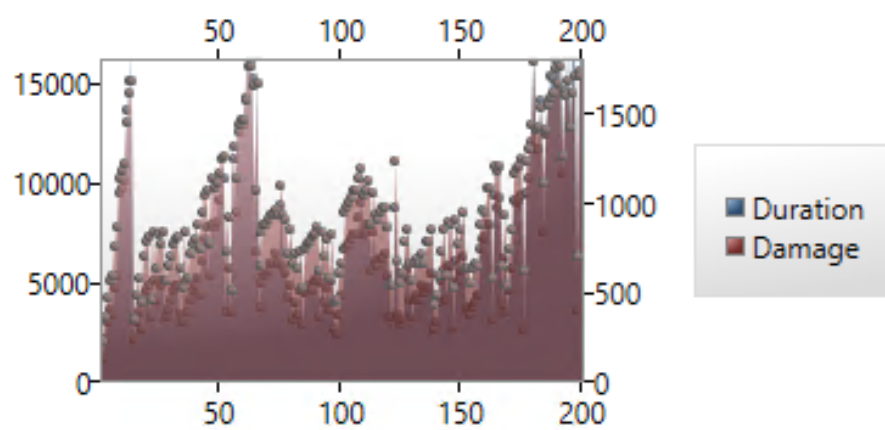


12. ábra. Generációk grafikonja ( $\gamma = 0.1$ , 800 generáció, 500 episode/generáció, 180 másodperc)

**Episodes** Meggondolandó a generációnkénti szimulációk generálásának száma is. Túl kevés esetén (13) az  $\varepsilon$ -mohó startégia felfedezésére nem adatik elég lehetőség. Túl sok esetén (14) a futás ideje és a program memória igénye megnő. Mivel a stratégia, amivel létrehozuk a szimulációkat csak generációváltás alkalmával frissül, a tanulás folyamata jelentős mértékkel nem gyorsul.



13. ábra. Generációk grafikonja ( $\gamma = 0.9$ , 200 generáció, 10 episode/generáció, 180 másodperc)



14. ábra. Generációk grafikonja ( $\gamma = 0.9$ , 200 generáció, 1000 episode/generáció, 180 másodperc)

## 4. Irodalomjegyzék

[1] Richard S. Sutton and Andrew G. Barto: Reinforcement Learning: An Introduction, The MIT Press, 2018, [528], ISBN-978-0-262-19398-6