Eötvös Loránd Tudományegyetem

Informatikai Kar

Komputeralgebra Tanszék

Gépi tanulás a legynagyobb sebzés eléréséhez

Dr. Vatai Emil

Adjunktus

Komputeralgebra tanszék

ELTE IK

Szalóki Sándor

Programtervező informatikus BSc.

Budapest, 2018

Tartalomjegyzék

1. Bevezető **2**

2. Felhasználói dokumentáció **3**

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 5

2.2.1. Játék model 5

2.2.2. Gépi tanulás 5

2.2.3. Grafikus felület 6

2.3. Rendszer-követelmények, telepítés 6

2.4. A program használata 7

3. Fejlesztői dokumentáció **12**

3.1. Specifikáció 12

3.2. Módszerek 12

3.2.1. Játék model 12

3.2.2. Gépi tanulás 19

3.2.3. Grafikus felület 23

3.3. Szerkezet 23

3.3.1. Külső függőségek 23

3.3.2. Adatszerkezetek 23

3.3.3. Modulok 23

3.3.4. Képességek 23

3.4. Tesztelés 23

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 megoldásához 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 képességere, 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, képességeinak 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 álltal 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 bemutatható néhány egyszerű grafikonon.

2.2. Módszerek

A feladat megoldása funkcionális programozással történik.

2.2.1. Játék model

Az említett két fő elemet (Karakterek és Képességek) megfelelő típusokkal felvesszük, 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 mennyiségú 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. Karaketerenként 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:

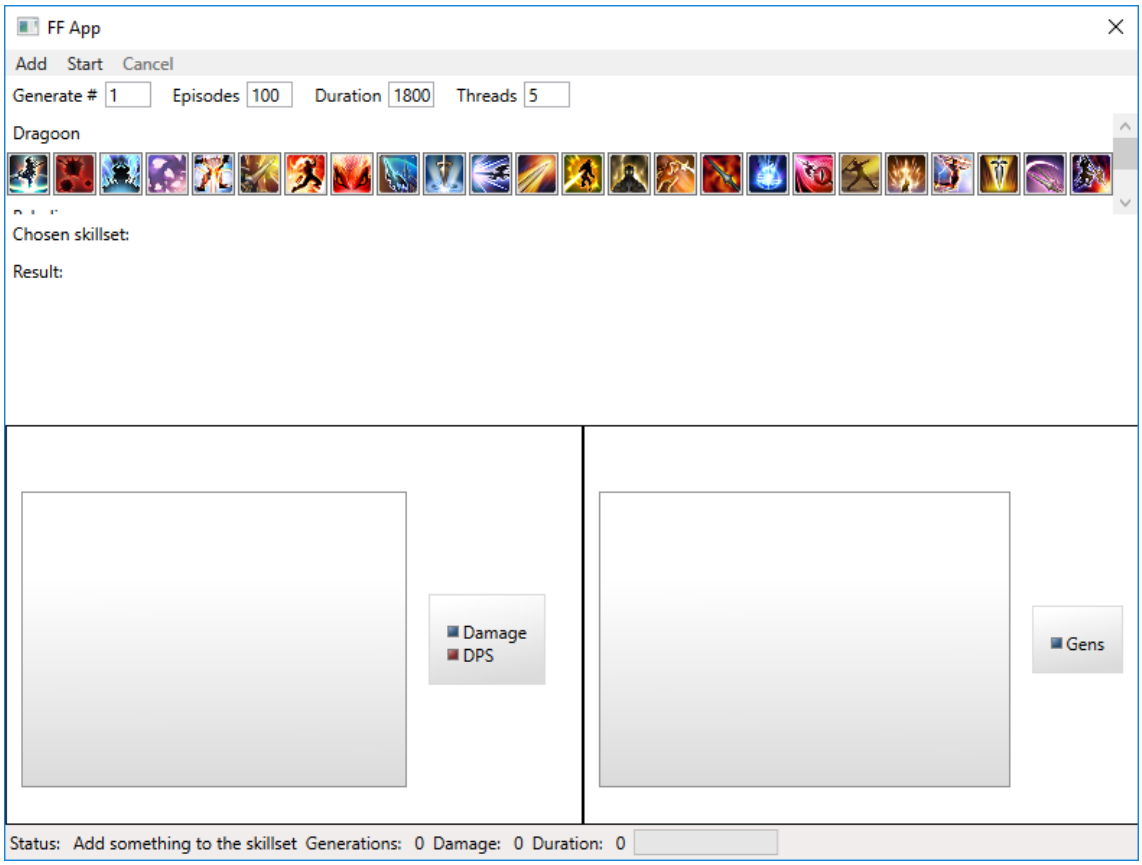
Status Rövid üzenet a program állapotáról

Generations A jelenlegi generáció száma

Damage A tanult képesség sorozat álltal elért sebzés értéke

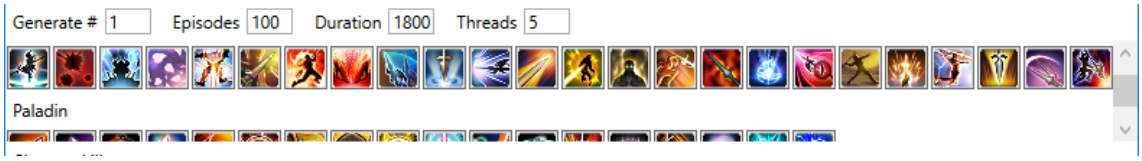
Duration A tanult képesség sorozat álltal elért idő tizedmásodpercben

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



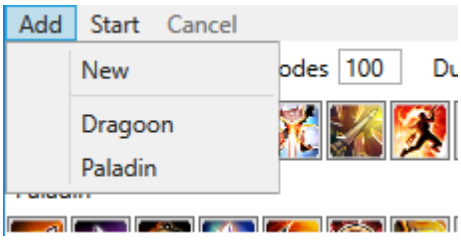
1. ábra. Kezdő képernyő

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



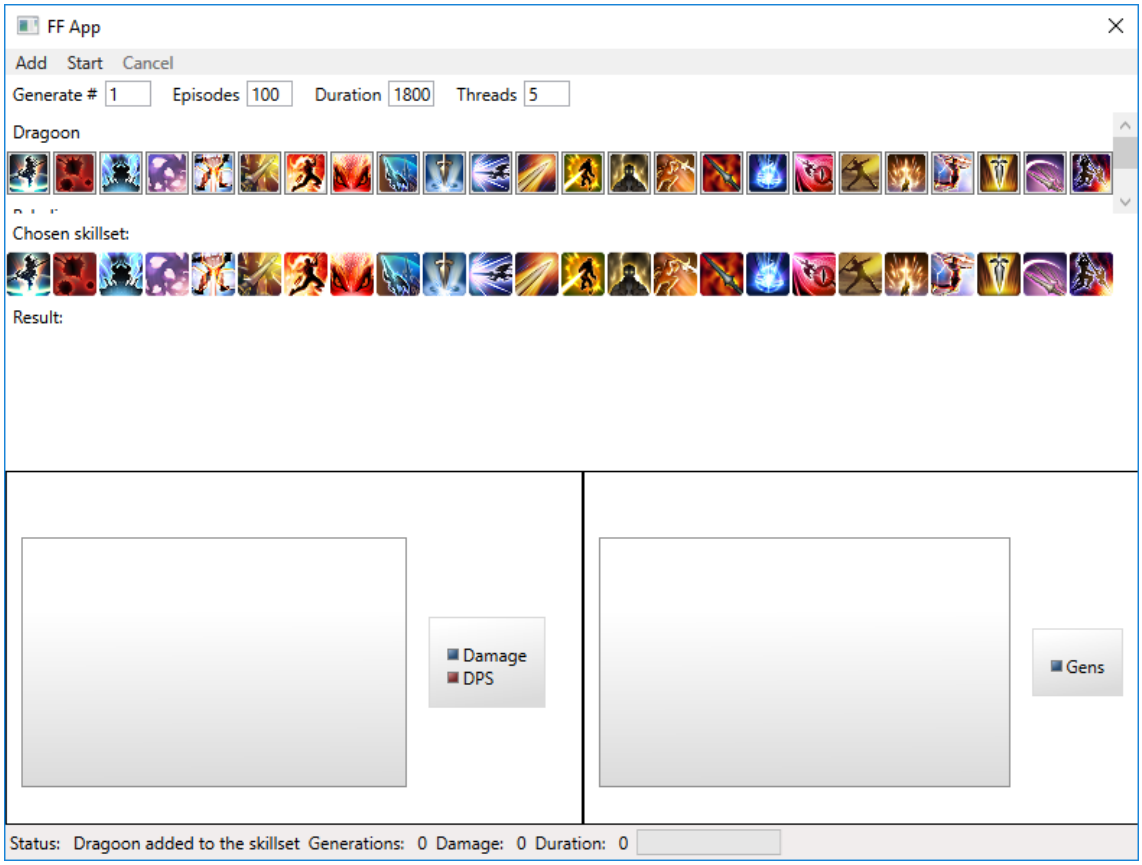
2. ábra. Lehetséges választható listák

Ha megtaláltuk a megfelelő listát, akkor az Add menüpontban (3) választhatjuk ki a szükséges listát név alapján. Fontos, hogy ilyenkor még szabadon cserélhetjük a karaktert, ez a menüpont Start-ra kattintás után nem lesz előrhető. (Kivéve a New gomb)



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:

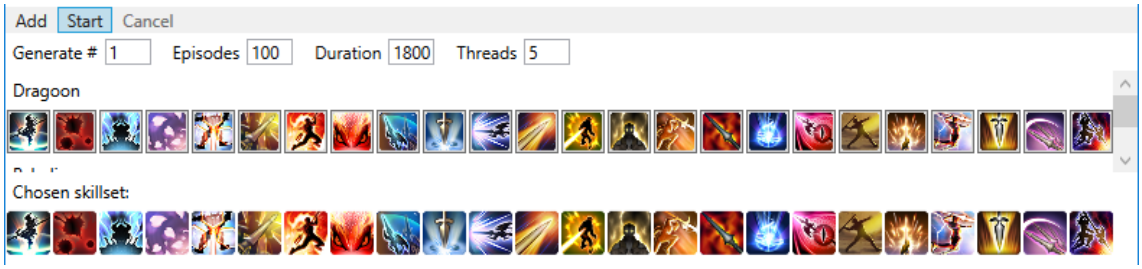
Generate A kívánt generációk számát adhatjuk meg

Episodes Generációnként létrehozott szimulációk száma

Duration Szimulációk hossza tizedmásodpercben

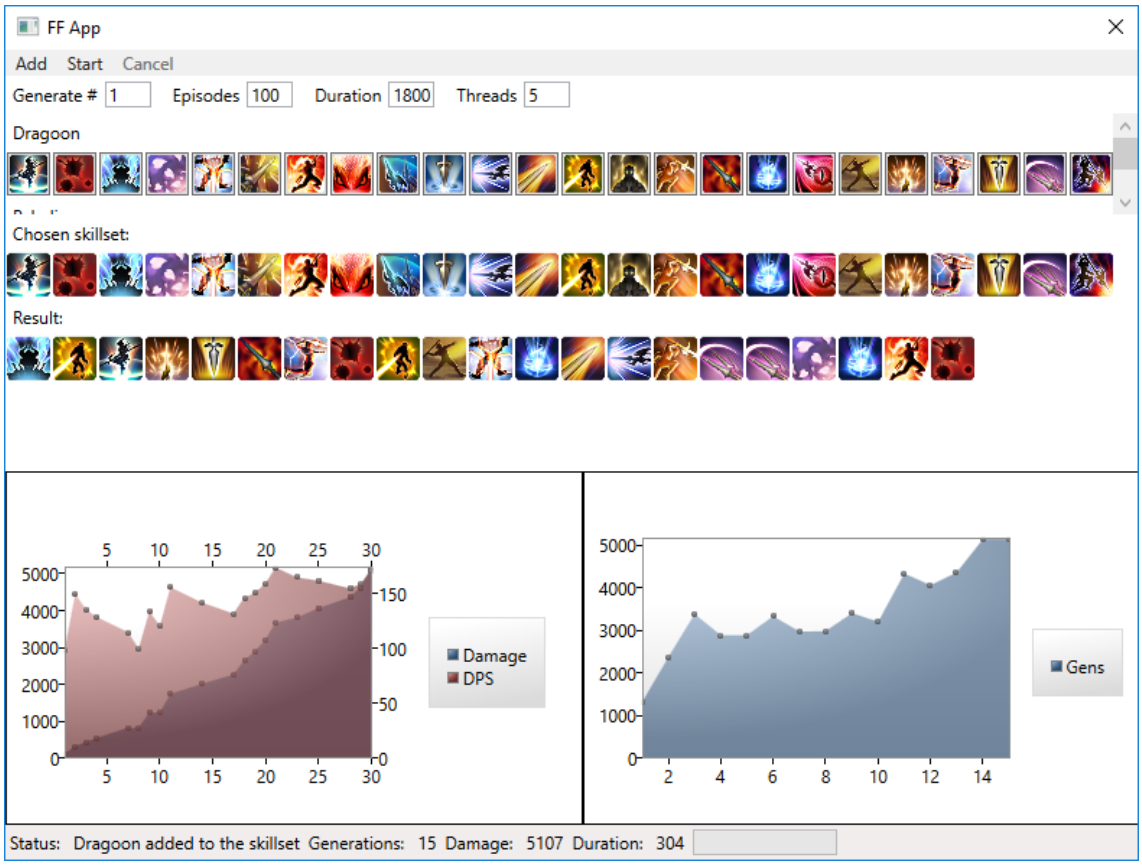
Threads Párhuzamosítás során használható magok száma

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

Amennyiben újra szeretnénk kezdeni a tanulás folyamatát, az Add menüpont alatt (3) található egy New gomb, amivel alaphelyzetre állíthatjuk a programot. Ilyenkor a kiválasztott képesség lista nem változik, ennek változtatásához a programot újra kell indítani.

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:

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

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 álltal 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 model

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 alabbi 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
   NotFirst       : bool
}
```

Job A Job típus valósítja meg a karaktert. Ennek jellemzése hasonló módon történik. Tulajdonságai jól megfeleltethetőek 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 orignalSkill = 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 álltal 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 megszámolni 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 álltal 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 tényezőből már egy számszerű sebzés érték rendelhető minden Tickhez. 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 a következő:

Kezdetben $\forall s \in \mathcal{S} : \forall a \in \mathcal{A}(s)$:
  $Q(s,a) \in \mathbb{R}$ (tetszőleges)
  $C(s,a) \leftarrow 0$
  $\pi(s) \leftarrow argmax_a Q(s,a)$
Végtelen ciklus (minden szimulációra) :
  $b \leftarrow$ tetszőleges gyenge stratégia
  Szimuláció léterehozása b-t használva: $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  $W \leftarrow 1$
  Ciklus a szimuláció minden lépésére, $t = T-1, T-2, ..., 0$:
    $G \leftarrow \gamma G + R_{t+1}$
    $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
    $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G - Q(S_t, A_t)]$
    $\pi(S_t) \leftarrow argmax_a Q(S_t, a)$
    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:

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

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

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

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

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

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

Episode : $\mathcal{E}^*$

GenerateEpisode : $\pi \to Episode$

CreateRandomPolicy : π

CreateGreedyPolicy : $\mathcal{Q} \to \pi$

CreateEpsilonGreedyPolicy : $\mathcal{Q} \times \varepsilon \to \pi$

MonteCarlo : $Q \times C \times \pi \to Q \times C \times \pi$

Ezek alapján az algoritmus megfeleltethető a következő függvénynek:

```
MonteCarlo : 𝒬 × 𝒞 × π → 𝒬 × 𝒞 × π
MonteCarlo Q C b =
  episodeFolder : ℝ × 𝒬 × 𝒞 × ℝ → ℰ → ℝ × 𝒬 × 𝒞 × ℝ
  episodeFolder (G, Q, C, W) E =
    (S, A, R) = E
    g = γG + R
    c (S, A) = C(S, A) + W
    q (S, A) = q(S, A) + W/C(S,A) [G − Q(S, A)]
    w = W 1/b(S)|A
    (g, q, c, w)
  episodes : Episode*
  episodes = [GenerateEpisode b|x ← [1..]]
  episodesFolder : 𝒬 × 𝒞 → Episode → 𝒬 × 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 : ℝ* → ℕ
ToSkill : ℕ → Skill
Add : Skill × Rotation → Rotation
FromRotation : Rotation → 𝒮
ToDamage : Rotation → ℕ
Time : Rotation → ℕ
EmptyRotation : Rotation
Duration : ℕ
```

Ezek használatával a szimuláció már egyszerűen létrehozható az alábbi módon:

```
GenerateEpisode : π → Episode
GenerateEpisode b =
  Generate : Rotation × S × L × Episode
  Generate Rot S l Ep =
    Ha l: Ep
    A : A
    A = Random (b S)
    (s, r, l, rot) : S × R × L × 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: ToDamage(rotation) / Time(rotation)
      finished = (Time Rot) = (Time rotation) ∨ (Time rotation) > Duration
      (state, reward, finished, rotation)
    Generate rot s finished (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 π értéke egy ε-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 π-t ez alapján a tudás alapján hozzuk létre egy már kisebb ε 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 használata segítségével:

```
GeneticMonteCarlo : Q × C × π → Q × C × π
GeneticMonteCarlo Q C b =
  Folder : Q × C × π → ℕ → Q × C × π
  Folder (Q C b) i =
    epsilon = 1 / (5 × 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

3.3. Szerkezet

3.3.1. Külső függőségek

3.3.2. Adatszerkezetek

3.3.3. Modulok

3.3.4. Képességek

3.4. Tesztelés