



MATEMATIKAI ÉS INFORMATIKAI INTÉZET

Stratégiai társasjáték fejlesztése megerősítéses tanulást alkalmazó mesterséges intelligenciával

Készítette

Budai Roland

Programtervező informatikus BSc.

Témavezető

Dr. Kovásznai Gergely

Egyetemi docens

EGER, 2024

Tartalomjegyzék

Bevezetés	4
1. MI és technológia	5
1.1. A gépi tanulás elméleti alapjai és a megerősítéses tanulás	5
1.1.1. Neurális hálózatok	5
1.1.2. Az ügynök és a környezet fogalma	6
1.1.3. Megerősítéses tanulás	6
1.2. Felhasznált technológiák	7
2. Játékmechanika	8
2.1. A játék szabályrendszere és célja	8
2.2. Játékosok interakciói és fázisai	9
3. Implementáció	12
3.1. Fejlesztési környezet és eszközök	12
3.1.1. Backend	12
3.1.2. Frontend	13
3.2. Adatkezelés és játékállapot tárolása	14
4. Mesterséges intelligencia tervezése	20
4.1. A neurális hálózat és az ügynök	20
4.1.1. Folyamatos és diszkrét akciótér	21
4.1.2. DQN és PPO	21
4.2. Kommunikáció a szerverrel	22
4.3. Környezet tervezése	24
4.4. Tanítás és fejlesztés	26
5. Az elkészült alkalmazás	31
5.1. Tanítás eredménye, pontosság, hiba mértéke	31
5.2. Az alkalmazás felhasználói felülete	32
Összegzés	36

Bevezetés

A szakdolgozatom témája egy stratégiai társasjáték implementálása beépített Mesterséges intelligenciával, amely segít a támadási döntések meghozatalában. A projekt ötlete nem csupán egy tanulmányi kötelezettség teljesítéséből fakad, hanem egy régebb óta tervezett hobbi projekt része is. Már a tanulmányaim elején megfogalmazódott bennem a kérdés, hogy hogyan tudnék egy hasonló társasjátékot a számítógépes világban megvalósítani. Milyen adatbázist kell elkészíteni, milyen döntéseket hozhatnak a játékosok, hogyan lehet megoldani, hogy egyszerre többen is hozzáférjenek az adatokhoz, mégis konzisztens maradjon a játékmenet és a tárolt állapotai a játéknak? Hasonló kérdések fogalmazódtak meg bennem tanulmányaim során, melyekre a képzés ideje alatt egyre jobb és jobb ötleteket sikerült szerezni. Mindezek miatt nagy lelkesedéssel álltam neki a feladatnak a témaülasztást követően.

További érdekessége a szakdolgozatnak a mesterséges intelligencia (MI) fejlesztése, használata. A mesterséges intelligencia és a gépi tanulás területei egyre nagyobb szerepet játszanak a modern játékfejlesztésben, ezért ez a projekt nemcsak szakmai kihívást jelentett számomra, hanem egyben kiváló lehetőséget is arra, hogy mélyebben megismerkedjek az MI alkalmazásával, fejlesztésével a játékok világában. Az MI implementálásával lehetőségem nyílt egy olyan ellenfél létrehozására, amely képes tanulni és folyamatosan fejlődni a játékok során.

A mesterséges intelligencia fejlődése az elmúlt években jelentősen befolyásolta a játékfejlesztés folyamatát, lehetővé téve, hogy az MI-alapú ellenfelek egyre intelligensebb döntéseket hozzanak és alkalmazzák a játékosok stratégiájához. Stratégiai döntések meghozatalához különösen a projektemben is használt megerősítéses tanulás (reinforcement learning, RL) vált népszerűvé, amely az MI folyamatos fejlődését, tanulását teszi lehetővé.

A szakdolgozatomban szeretnék elsősorban a mesterséges intelligenciáról írni, bemutatni a projektet általánosságban, hogy milyen technológiák kerültek felhasználásra. Ezek után magáról a játékról, annak szabályairól lesz szó, majd ennek a konkrét implementációjáról online környezetben. Részletesen kifejtem a mesterséges intelligencia megtervezésének és fejlesztésének folyamatát, majd végezetül bemutatom az elkészült alkalmazást, összefoglalom a fejlesztés során szerzett tapasztalatokat. A projekt a következő GitHub linken érhető el: <https://github.com/Rolidroid0/Szakdolgozat>.

1. fejezet

MI és technológia

1.1. A gépi tanulás elméleti alapjai és a megerősítéses tanulás

A mesterséges intelligencia fejlődése során a gépi tanulás (machine learning, ML) kapott kiemelkedő szerepet az intelligens rendszerek fejlesztésében. A gépi tanulás olyan módszereket foglal magában, amelyek lehetővé teszik a számítógépes modellek számára, hogy tapasztalatok alapján javítsák a teljesítményüket, a programozásuk ember általi megváltoztatása nélkül. Az ML három fő típusa a felügyelt tanulás (supervised learning), a felügyelet nélküli tanulás (unsupervised learning) és a megerősítéses tanulás. [1]

A felügyelt tanulás esetében a rendszer egy előre meghatározott bemenet-kimenet párokból álló adathalmazon tanul, míg a felügyelet nélküli tanulás során a modell mintázatokat keres egy struktúra nélküli adathalmazon belül. Ezekkel szemben a megerősítéses tanulás egy olyan módszer, amelyben egy ügynök (agent) a környezetével (environment) interakcióba lépve tanul, és az akciókért visszacsatolást kap jutalmak (reward) vagy büntetések formájában.

1.1.1. Neurális hálózatok

A neurális hálózatok az ML egy legfontosabb eszközei, amelyek a biológiai neuron-hálózatok mintájára épülnek fel. [2] Egy mély neurális hálózat (deep neural network, DNN) több rétegű csomópontkból áll, ahol az egyes rétegek között súlyok segítségével történik az információátadás. Ezek az architektúrák jól alkalmazhatók képfelismerés, természetes nyelvfeldolgozás és komplex döntéshozatali folyamatok esetén. A megerősítéses tanulásban gyakran alkalmazznak mély neurális hálózatokat, például Deep Q-Network algoritmust, amely egy Q-tanulási (Q-learning) stratégiát kombinál mély tanulási modellekkel, lehetővé téve az ügynök számára a nagyobb állapottérrel rendelkező

környezetben való hatékony tanulást. [3]

1.1.2. Az ügynök és a környezet fogalma

Egy megerősítéses tanulást alkalmazó szoftver fejlesztése során két központi elemet kell meghatároznunk: az ügynököt és a környezetet. Az ügynök az a döntéshozó entitás, amely egy adott állapotban kiválaszt egy akciót a környezet befolyásolására. A környezet pedig az a rendszer, amelyben az ügynök működik. Folyamatosan visszacsatolást ad az ügynök döntéseire jutalmak vagy büntetések formájában. A tanulási folyamat során az ügynök folyamatosan próbálja maximalizálni az összegyűjtött jutalmat, miközben az ismeretlen környezetben kísérletezik és tanul a visszajelzések alapján.

1.1.3. Megerősítéses tanulás

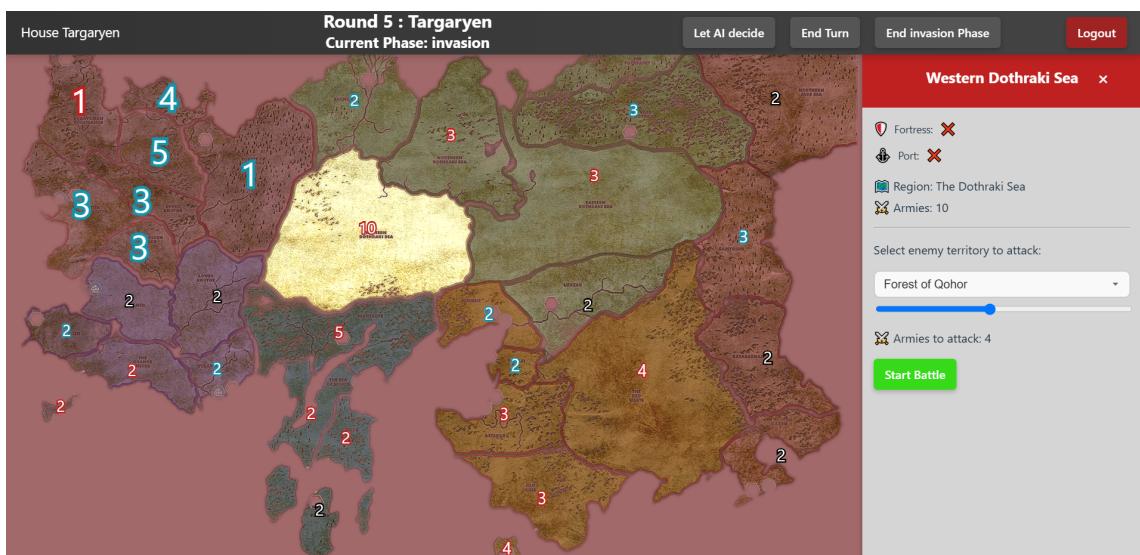
Mint említettem, a megerősítéses tanulás (Reinforcement Learning, RL) olyan gépi tanulási módszer, amelyben az ügynök egy környezetben interakciók során tanul optimális stratégiákat kialakítani. Az RL egyik legfőbb célja a döntéshozatal optimalizálása egy adott probléma keretein belül. [4] A megerősítéses tanulás egyik alapvető fogalma az állapot (state, S), amely a környezet aktuális reprezentációja, amely leírja az ügynök és a környezet közötti pillanatnyi helyzetet. A cselekvés (action, A) az ügynök által végrehajtható lépések halmaza. A jutalom (reward, R) a környezet visszacsatolása egy adott cselekvés végrehajtása után. Az ügynök célja ezen jutalom hosszú távon való maximalizálása. Az ügynök által használt politika (policy, π) egy szabályrendszer, amely meghatározza, hogy egy adott állapotban milyen cselekvést hajtson végre az ügynök. A hosszú távú jutalom várható értékének számításához egy adott állapot és cselekvés kombinációját használja, ez az úgynevezett Q-érték (Q-value, $Q(s, a)$).

Az RL során az ügynök kísérlet és hiba (trial and error) folyamat révén tanulja meg a várhatóan legjobb kimenetelű döntéseket. Számtalan algoritmust használhatunk a tanításhoz, ezek közül a legismertebbek közé tartozik a Q-learning és a Deep Q Network (DQN), amelyek az állapot-cselekvés értékek (Q-értékek) becslésén alapulnak.

A megerősítéses tanulás alkalmazásai számos területet lefednek, beleértve a robotikát, az önvezető járműveket és a stratégiai játékokat. Egy stratégiai játékban az RL lehetőséget ad arra, hogy a mesterséges intelligencia adaptálódjon a játékos döntéseihez, és folyamatosan fejlessze saját taktikáját. Az ilyen típusú modellek egyik legfontosabb előnye, hogy nem előre megírt szabályok szerint működnek, hanem a tapasztalatokból tanulnak és fejlődnek.

1.2. Felhasznált technológiák

A projekt megvalósítása során több különböző technológiát alkalmaztam, amelyek lehetővé tették a játék és a mesterséges intelligencia hatékony implementálását. Az MI fejlesztésének alapját a Python nyelv képezte, mivel számos hatékony könyvtár áll rendelkezésre a gépi tanulás számára. A neurális hálózat betanításához és optimalizálásához a PyTorch keretrendszer nyújtott megoldást. A játék backendjének fejlesztéséhez JavaScript-et alkalmaztam, amely lehetővé tette a szerveroldali logikák és az adatkezelés megvalósítását. Az adatokat a dokumentumorientált MongoDB adatbázisban tároltam. A React szolgált egy dinamikus és interaktív frontend fejlesztésére. A szerver és a különböző kliensek közötti kommunikációt kétféleképpen oldottam meg. Az alapvető adatok lekérdezésére REST API-n keresztsüli kommunikációt biztosítottam, a valós idejű játékélmény és gyors adatcsere érdekében pedig WebSocket kapcsolatot hoztam létre. További fejlesztési eszközöként szolgáltak a GitHub, a Visual Studio Code és az Inkscape amely a frontendhez szükséges svg kiterjesztésű fájlok létrehozásában segített (lásd: 1.1 ábra).



1.1. ábra. Pillanatokkal egy csata előtt.

2. fejezet

Játékmechanika

Ebben a fejezetben bemutatom a játék alapvető mechanikáit, szabályrendszerét, valamint a játékosok interakciót és a játékmenet fázisait. A játék egy körökre osztott stratégiai társasjáték digitális implementációja, amelyben a játékosok különböző területek felett próbálnak uralmat szerezni, harcolnak egymással, és erőforrásokat kezelnek. A játék szabályrendszere, mechanikái és karakterei a játék készítőinek¹ szellemi termékei, és ezen elemek kidolgozása nem az én saját alkotásom, csak tanulási célból alkalmazom őket a dolgozatom során.

2.1. A játék szabályrendszere és célja

A játékot többféleképpen lehet játszani, kettő játékosról egészen hézig, egy vagy két térkép bevonásával. A különböző játékmódok különböző bónuszokat tartalmaznak, mint a karakterkártyák, a pénz és erődítmények. A végső implementációban a legegyszerűbb kétszemélyes játékváltozat lett megvalósítva, ezért a továbbiakban ennek a szabályait fogom részletezni.

A játék célja, hogy a játékosok előre meghatározott győzelmi feltételt teljesítsenek, amely lehet bizonyos pontszám elérése, vagy az összes többi játékos legyőzése. A játékosok egy-egy házat képviselnek, és a térképen található területek felett próbálnak uralmat szerezni támadások és stratégiai döntések révén.

A játéktér egy rácsszerkezetként értelmezhető térkép, amely különböző régiókra oszlik. minden régió több területet tartalmaz, amelyek birtoklása kulcsszerepet játszik a játékosok stratégiájának kialakításában. A területek rendelkezhetnek erődökkel és kikötőkkel, amelyek befolyásolhatják a harci döntéseket. A játék legelején véletlenszerűen kiosztásra kerül 12-12 terület a két játékos között, majd a maradék területeket az úgynevezett semleges seregek birtokolják. minden területre a játékosok felhelyeznek két sereget, majd a területkártyák megkeverésre kerülnek úgy, hogy a játék vége kártya

¹ Hasbro

a pakli alsó felébe kerüljön véletlenszerű helyre.

Egy játékos körében négy különböző akciót lehet végrehajtani: erősítés, invázió, manőver és húzás. A játék szabályai biztosítják, hogy minden játékos azonos esélyel induljon, és a stratégiai döntéseik határozzák meg a végeredményt. A körökre osztott rendszer lehetővé teszi, hogy a játékosok átgondolják lépéseiket és reagáljanak az ellenfelek akcióira.

Amikor a pakliba rejtett játék vége kártya a felszínre kerül, minden játékos megszámolja a pontjait és a legtöbb ponttal rendelkező játékos nyer. Pont jár a területek, a várak és a kikötők birtoklásáért. Természetesen ha egy játékos minden területet megszerez, automatikusan megnyeri a játékot.

2.2. Játékosok interakciói és fázisai

A játékosok interakciói alapvetően három fő dolgon alapszanak: területfoglalás, csaták és diplomácia. Mint említettem a játék egy körökre osztott rendszerben zajlik, ahol a körök több fázisból állnak. minden kör végén a következő játékos veszi át az irányítást. A négy fázis/végrehajtható akció a következő:

- **Erősítés:** a játékos minden köre elején plusz seregeket kap, amelyekkel tetszőlegesen erősítheti a birtokában lévő területeket. A plusz seregek számát a következőképpen számolják:
 - A birtokolt területek számát össze kell adni a rajtuk lévő várak számával, elosztani hárommal és lefelé kerekíteni. Fontos kiegészítés, hogy legalább három sereget mindenkorban kap a játékos, még ha a számolás eredményeképp kevesebb járna is.
 - minden terület egy régióhoz tartozik és minden régióhoz van egy plusz sereget jelentő értéke. Ha egy játékos egy adott régió összes területével rendelkezik, akkor a meghatározott plusz seregeket hozzáadhatja az eddigi erősítésként járó seregeihez. Természetesen ha több régiót is teljesen birtokol egy játékos, akkor az összes régió bónuszt megkapja.
 - minden területkártyán van egy különleges egységet ábrázoló kép, amelyekből ha három összegyűlik további seregekre tehet szert a játékos. Fontos megjegyzés, hogy nem lehet öt kártyánál többet birtokolni, ha ebben a fázisban öt kártyája van egy játékosnak, azokból mindenkorban be kell váltania hármat. További bónusz seregekre tehet szert a játékos, ha a beváltott kártyákon lévő területeket is birtokolja. Ilyenkor két extra sereget helyezhet fel kizárálag ezekre a területekre.

Végezetül az összes meghatározott plusz sereget el kell helyezni a táblán a játékos által birtokolt területekre. Ezeket lehet tetszőlegesen elosztani vagy akár minden egy területre lehelyezni.

- **Inváziók és csaták:** Az inváziók és csaták a játék és a játékos körének a legfontosabb eseményei. Itt kell eldönteni, hogy kit és hol támadnak meg a játékosok, ezáltal közelebb kerülve a győzelemhez. Egy körben bármennyi inváziót kezdeményezhet a játékos, azonban egy területről csak egy támadás indítható adott körben. Olyan területeket lehet támadni, amelyek szomszédosak a kezdeményező terüettel és ellenség birtokolja őket. A kikötővel rendelkező területek szomszédsnak számítanak egymással, ezáltal ezekről a területekről akár messzebbre is el lehet kezdeni terjeszkedni. Természetesen a játékos bármennyiszer támadhat egy körben amennyiszer lehetséges, de akár dönthet úgy is, hogy egyszer sem támad, ekkor automatikusan folytatódik a kör a következő fázissal. Egy invázió megkezdésekor ki kell választani, hogy melyik területről és melyik területre szeregne megadni a támadást, továbbá meg kell határoznia a támadó seregek számát. Ez a szám nem haladhatja meg az adott területen lévő seregeinek a számát és legalább egy sereget hátul kell hagynia, hogy ne maradjon üresen a terület. Miután egy invázió sikeresen létrejön, megkezdődnek a csaták. A támadó fél maximum három sereggel csatázhat, még ha ennél többel is kezdte meg az inváziót, és a védekező pedig legfeljebb két sereggel védekezik. Mindkét játékos a csatázó seregeinek számával megegyező kockával dob, majd ezeket az értékeket párosítják a legmagasabbaktól a legalacsonyabbakig. Ha a támadó legnagyobb dobása a magasabb, akkor a védekező veszít egy sereget, ha a védekező dobása a nagyobb vagy egyenlő, akkor pedig a támadó vesz le egy sereget. Hasonló összehasonlítás történik a második legnagyobb dobásokkal is. Ha az egyik fél több kockával dobott, aminek nem jutott már párja, akkor azt figyelmen kívül kell hagyni, az nem okoz veszteséget egyik fél számára sem. Ez egészen addig ismétlődik, amíg az egyik fél seregei el nem fogynak az adott invázió során. Ekkor ha a védőnek nem marad serege a megtámadott területen, a támadó megszerzi az adott területet, és a megmaradó seregei felkerülnek erre a területre. A semleges seregek által birtokolt területekre hasonló szabályok érvényesek, viszont a dobásokat nem a másik játékos végzi, hanem azok automatikusan végrehajtódnak.
- **Manőver:** Amikor a játékos úgy dönt, hogy nem támad tovább ebben a körben, akkor egyszer megteheti, hogy a seregei mozgatásával megerősít egy területet. Az átcsoportosítás során tetszőleges számú sereget (egy seregnél minden képpen maradnia kell a területen) helyez át egy területről egy másik, azzal összekötöttében lévő területre. Két terület akkor áll összeköttetésben, ha a közük lévő összes terület ugyan azé a játékosé, tehát ellenséges vagy semleges területeken

nem mozoghat át sereg.

- **Területkártyák húzása:** Végezetül a játékos köre végén ha legalább egy ellenséges területet meghódított, húzhat egy területkártyát a pakli tetejéről. Egy körben csak egyet húzhatnak a játékosok, függetlenül a megszerzett területek számától. A megszerzett kártyákat az erősítés részben tárgyalt módon lehet plusz seregekre váltani.

3. fejezet

Implementáció

3.1. Fejlesztési környezet és eszközök

A projekt fejlesztéséhez Visual Studio Code-ot használok, mivel támogatja mind a Javascript, TypeScript és Python nyelveket, továbbá különböző bővítményekkel egy-szerűsíthető a fejlesztési folyamat. Egy ilyen bővítmény például a MongoDB, ami által könnyedén ellenőrizhetjük az adatbázisban történő változások sikerességét. A verziókötetésre a GitHub szolgál, amely lehetővé teszi a változások követését és a biztonságos tárolást, így felváltva tudom fejleszteni a projektet asztali számítógépről és laptopról.

3.1.1. Backend

A játék alapját egy Node.js alapú backend biztosítja, amely a MongoDB adatbázissal kommunikálva tárolja és frissíti a különböző játékállapotokat. A szerver Express.js keretrendszer használ a HTTP és WebSocket kapcsolatok kezelésére, lehetővé téve a valós idejű kommunikációt a játékosok között. A backend struktúrája a következő mappákra oszlik: models, routes, controllers, services, utils, handlers, config és db-seed.

A *models* mappában az adatbázisban tárolt dokumentumok sémái találhatók, mint például a kártyák vagy területek modellei. Ezen modellek felhasználásával történik meg az adatbázis feltöltése is a *db-seed* mappában elhelyezkedő fájlok segítségével.

A *routes* az API végpontok definiálására szolgál, amelyek a kliens kéréseit kezelik. Ezekhez a végpontokhoz tartoznak különböző műveletek, amelyek a *controllers* mapában helyezkednek el. Az üzleti logika nagy része itt történik, a kérések itt kerülnek feldolgozásra. A controller-ek különböző szolgáltatásokat használnak, akár többet is, amelyek a *services* mappában találhatók. Ezek az önállóan működő szolgáltatások a legfontosabbak, hiszen ezeken keresztül történik a kommunikáció az adatbázissal. A különböző játékbeli lépések itt vannak rendszerezve egy-egy fájlban, mint például a *cardsService.ts*, amely a kártyákhoz tartozó függvényeket tartalmazza (kártyák keveré-

se, kártya húzása, kártyák beváltása seregekre). Azok a függvények, amelyek különféle segédfüggvények, például adatellenőrzések és egyéb általános műveletek az *utils* mapában helyezkednek el.

A *config* mappában két lényeges része kerül beállításra a szervernek: az adatbázissal való kapcsolat és a WebSocket szerver. Az adatbázishoz való csatlakozás egy Singleton osztály, amely ha nincs még csatlakozva egy mongoose.Connection példányon keresztül, akkor kialakít egy kapcsolatot az adatbázis URI-ját használva. A WebSocket szerver pedig létrehozza a szervert, majd várja a kliensek fel- és lecsatlakozását, illetve az üzeneteiket. minden üzenetnek tartalmaznia kell egy kívánt akciót és opcionális adatokat. Az akciók a *handlers* mappában kerülnek feldolgozásra a neveik alapján. Mindhez tartozik egy függvény, amelyek a különböző szolgáltatásokat hívják meg az akciók végrehajtására, majd visszajelzést küldenek az üzenetet küldő kliensnek, vagy bizonyos esetekben az összes kliensnek (például egy terület megerősítéséről minden kliensnek tudnia kell, lásd: 3.1).

```
1 const wss = getWebSocketServer();
2 wss.clients.forEach(client => {
3     if (client.readyState === WebSocket
4         .OPEN) {
5         client.send(JSON.stringify
6             ({
7                 action: 'territory-
8                     updated',
9                 data: { territory }
10            }));
11    }
12 });
13 }
```

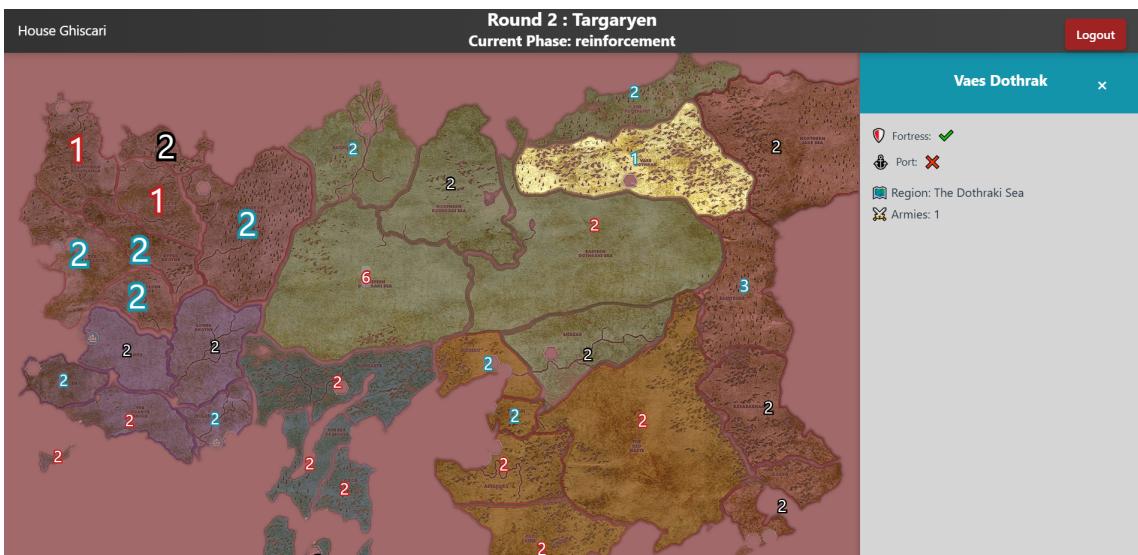
3.1. WSS kliensek értesítése.

3.1.2. Frontend

A React segítségével épített frontend biztosítja a játékosok számára az interaktív felületet egy webes alkalmazás formájában. A React könyvtár komponens alapú felhasználói felületek létrehozására alkalmas. Az egész alapját az App.tsx fájl adja, amely a főoldalnak felel meg. Ezen helyezkednek el a különböző komponensek, mint például a fejléc. Az egyes komponensek két részből állnak, egy tsx fájlból, amely tartalmazza a hozzá tartozó mezőket és függvényeket, illetve a végén egy HTML szerű leírást és egy css fájlból, ami pedig a formázásban és megjelenítésben segít. A legtöbb komponens rendelkezik egy UseEffect() függvénnyel, amely folyamatosan figyeli, hogy érkezik-e üzenet a szerver felől WebSocket-en keresztül, ha igen, akkor pedig a tartalma alapján eldönti,

hogy módosítania kell-e valamit a mezőin, vagy nem neki szólt ez az üzenet. A szerverhez hasonlóan a klienseken is fut egy szolgáltatás, amely egy Singleton osztálynak felel meg, és ezáltal csatlakozik a WebSocket szerverhez. Ebben a szolgáltatásban tudnak regisztrálni a komponensek úgynevezett handler-eket, amelyek figyelik a szervertől kapott üzenetek tartalmát. További szolgáltatások is megtalálhatók a kliensek kódjában, ezek azonban az API végpontok elérésében segédkeznek. Szintén a szerverhez hasonlóan itt is megjelennek az egyes adatbázis entitások modelljei.

A frontend egyik legkiemelkedőbb része a térkép. Ez az *assets* mappában helyezkedik el egy svg fájlként, amelyet az Inkscape program használatával került megalkotásra. minden terület egyedi azonosítóval rendelkezik, amely a játékbeli névvel egyezik meg, így a frontend felismerheti őket. A ReactSVG könyvtár biztosítja az interaktivitást, lehetővé téve az elemek manipulációját. Például egy területre való kattintásnál az a terület élesebb színűvé válik, illetve az azonosító alapján meg tud nyílni oldalt a hozzá tartozó fül a terület adataival (lásd: 3.1). Továbbá minden területen számszerűen megjelenik a rajta elhelyezkedő seregek száma a birtokos házának színével.



3.1. ábra. Vaes Dothrak terület adatai.

3.2. Adatkezelés és játékállapot tárolása

A játék egyik legfontosabb eleme a játékállapot pontos eltárolása, amely biztosítja a játékosok számára a legfrissebb helyzeteket. Ezáltal az adatbázis tervezése volt az egyik legfontosabb feladat az implementáció során.

A projektben MongoDB-t használok, mivel ez egy NoSQL-alapú, dokumentumorientált adatbázis. JSON-szerű dokumentumokat használ, amik könnyedén lekérdezhetők, kezelhetők és bővíthetők, ezáltal jól illeszkednek a játékstruktúrához. [5] A játékállapot folyamatos frissítése során fontos biztosítani, hogy ne történjenek ütközések,

illetve adatvesztések. Ennek biztosítására kínál lehetőséget a MongoDB az Atomic Update műveletekkel, mint például az updateOne() és findOneAndUpdate() beépített függvények (lásd a 3.2 kódrészletet).

```

1  const otherCard = await essosCards.findOne<
2      Card>({ sequence_number: newEndPosition,
3             game_id: ongoingGame._id });
4
5  ...
6
7  await essosCards.updateOne(
8      { _id: endCard._id, game_id:
9          ongoingGame._id },
10     { $set: { sequence_number:
11         newEndPosition } }
12 );
13
14
15 const shuffledCards = await essosCards.find
16     <Card>({ game_id: ongoingGame._id }).toArray();

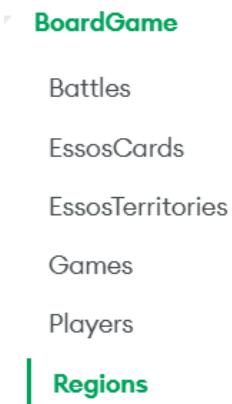
```

3.2. Példa az Atomic Update műveletekre a cardsService.ts-ból.

Hat adatbázis-kollekció lett létrehozva, a következőképpen (lásd: 3.2 ábra):

- **Players**: a játékosok adatait tartalmazza
- **Games**: az aktuális és lezárt játszmák állapotát tartalmazza
- **EssosTerritories**: a térkép területeit és kapcsolataikat írja le
- **EssosCards**: a játékban szereplő kártyák helyzetét rögzíti
- **Regions**: a területeket összekötő régiókat tartalmazza
- **Battles**: a folyamatban lévő és lezárt csaták részleteit kezeli

A játék adatbázisának célja, hogy hatékonyan kezelje a játékállapotot, a területek, a játékosokat, a kártyákat és a csatákat. Az alábbiakban részletesen bemutatom az



3.2. ábra. A MongoDB-ben tárolt kollekciók.

egyes táblák szerepét, a közöttük lévő kapcsolatokat, valamint az adatok kezelésének menetét. A régiókat tartalmazó kollekció a Regions.csv-ből került betöltésre, azonban mivel ezek az adatok nem változnak játék közben és két játék között sem, ezért ezzel a táblával adatmódosítás szempontjából nem is kell foglalkozni. Fő célja a kör elején kapott bónusz seregek számításánál a régióbónusz meghatározása. Erre a calculatePlusArmies(playerId: ObjectId) metódusban kerül sor (ahogya az a 3.3 kód részletben látható).

```

1 const regions = await regionsCollection.
2   find<Region>({}).toArray();
3 for (const region of regions) {
4   const ownedTerritoriesInRegion =
5     territories.filter(territory =>
6       territory.region === region.name
7     );
8   if (ownedTerritoriesInRegion.length
9     === region.territory_count) {
10     additionalArmies += region.
11       region_bonus;
12   }
13 }
  
```

3.3. Régióbónusz meghatározása.

Egy új játék elkezdésénél elsősorban a Games kollekció került módosításra. A state mezője felel a játék státuszának követésében, háromféle értéket vehet fel: ongoing, azaz ez a játék még éppen folyamatban van, terminated, miszerint a játék valamilyen oknál fogva befejeződött, le lett zárva a szerver által és X won, ahol az X a győztes játékos házának neve. Amikor valaki elindít egy új játékot, akkor a szerver megnézi, hogy van-e folyamatban lévő játék, és ha van, akkor azt lezárja, majd létrehoz egy új példányt

egyedi azonosítóval és a kört 1-re állítja.

Ezután következik a kártyák betöltése a seedEssosCards() függvényel és ezek megkeverése a shuffle() függvényel. Hasonlóan a régióhoz, a kártyák is egy csv-ből kerülnek az adatbázisba, azonban a kártyák minden játék elején újratöltődnek. minden kártyánál beállításra kerül a játékazonosító mező az előtte létrehozott új játék azonosítójával, illetve a tulajdonos mező "in deck" azaz a pakliban értéket vesz fel. A shuffle() biztosítja, hogy a kártyák véletlenszerűen legyenek elhelyezve a pakliban (a játék vége kártya is itt van biztosítva, hogy ne kerülhessen elő túl hamar, lásd 3.4 ábra).

```
1 const endCard = await essosCards.findOne<
2   Card>({ symbol: Symbol.End, game_id:
3     ongoingGame._id });
4
5 const minPosition = Math.floor(cardCount /
6   2);
7 const maxPosition = cardCount - 1;
8 const newEndPosition = Math.floor(Math.
9   random() * (maxPosition - minPosition +
10  1)) + minPosition;
11
12 if (!endCard || !otherCard) {
13   console.error('Cards not found');
14   return;
15 }
16
17 await essosCards.updateOne(
18   { _id: endCard._id, game_id:
19     ongoingGame._id },
20   { $set: { sequence_number:
21     newEndPosition } }
22 );
23
24 await essosCards.updateOne(
25   { _id: otherCard._id, game_id:
26     ongoingGame._id },
27   { $set: { sequence_number: endCard.
28     sequence_number } }
29 );
```

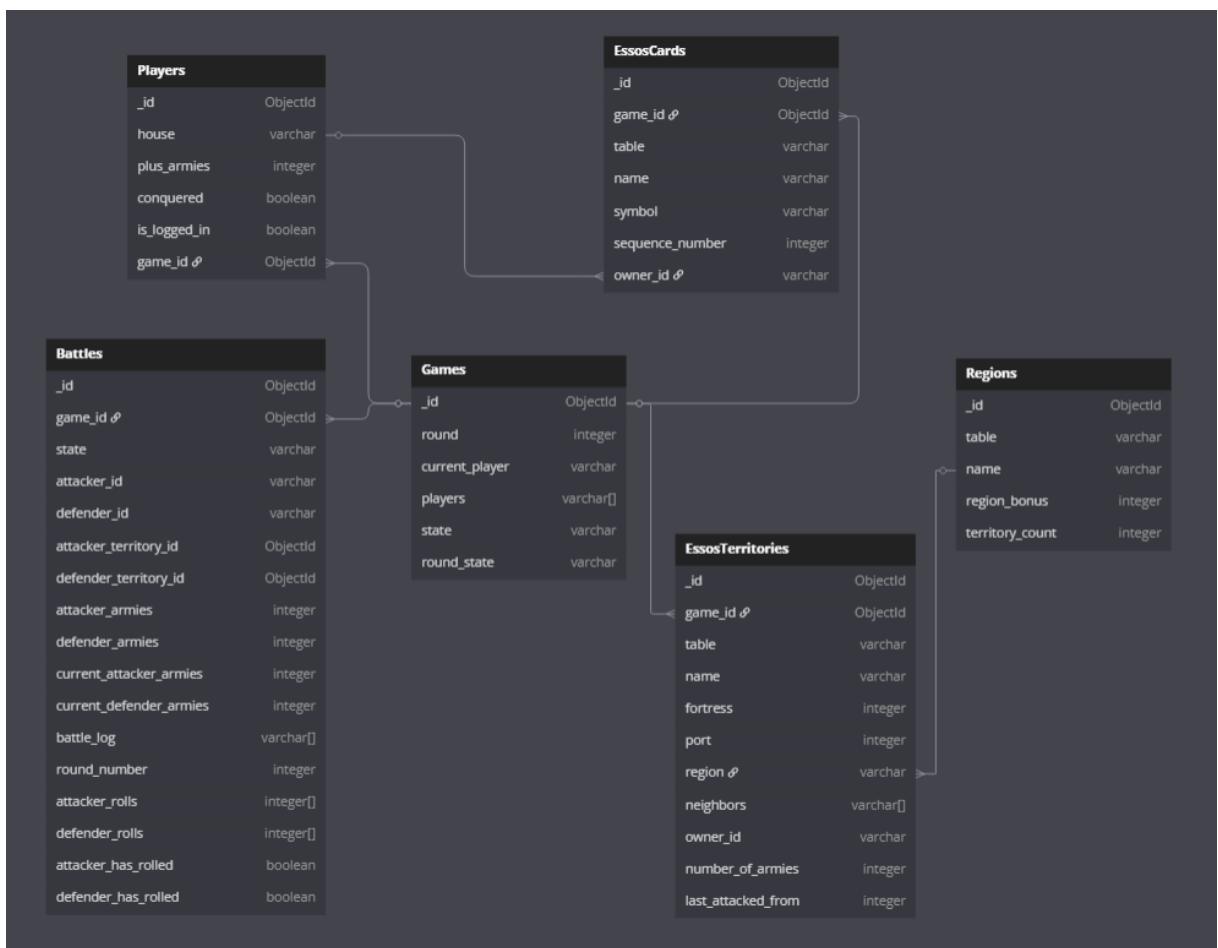
3.4. A játék vége kártya helyének biztosítása.

Miután minden kártya a helyére került, a generatePlayers(numberOfPlayers: number) függvény kerül meghívásra. A függvény létrehozza a kívánt mennyiségű játékost és betölti őket az adatbázisba, majd a Games kollekcióban módosításra kerül a players tömb, belekerülnek az újonnan létrehozott játékosok házainak nevei.

Végezetül a területek kerülnek az adatbázisba a seedEssosTerritories() függvénnyel. A függvény csak betölti a területekhez tartozó általános adatokat, az allocateTerritories() függvény felel a terület birtokosának meghatározásában. Ez a függvény két játékos között osztja szét a területeket a következőképpen: 12 véletlenszerűen választott területet kap az első játékos, 12 területet a második és a maradék terület birtokosa a semleges értéket kapja. Mindezek után az első játékos megkapja a neki járó plusz seregeket és kezdődhet is a játék.

Egyetlen kollekció van, amely egy új játék létrehozásánál nem játszik semmilyen szerepet, a csatákat tartalmazó tábla. Ez a kollekció felel a már lejátszott és a még zajló csaták nyomon követésében. Amikor egy csata létrejön, akkor beállításra kerülnek az alapvető mezők mint például a támadó azonosítója, vagy a védekező terület azonosítója a szerver által. A csata közben a kör száma mező módosul, a dobások értékeinek mezője, illetve a csatanapló. Ha egy csatának vége van, akkor az állapota módosul az egyik fél győzelmére, majd a csatanaplóban lehet visszanézni, hogy pontosan hogyan is ment végbe az adott csata.

A 3.3 ábrán látható a teljes adatbázis tervezete. A fejlesztés során eleinte nem minden tábla tartalmazta a game_id mezőt, mivel nem volt rá szükség, a mesterséges intelligencia tervezése során merült fel, hogy ezt a mezőt is be kell vezetni néhány táblán. Ezen változtatások miatt vannak mezők, amelyek magyarázatot igényelhetnek. Ilyen például az EssosCards tábla owner_id mezőjének típusa. Ez a mező hivatott összekötni a játékosok és a kártyák tábláit. Mivel egy játékos és egy kártya entitás is rendelkezik game_id mezővel, így biztosítva van, hogy csak az adott játékban lévő kártyák és játékosok kerülhetnek össze, ezért elég a játékos házának nevét megadni az azonosítója helyett, mivel a house mező egy játékon belül minden játékosnak egyedi. Hasonlóan a Games táblában a current_player és a players mezők a játékosok házainak neveit tartalmazzák mint azonosító. Az EssosTerritories táblánál pedig a region mező szintén egy régió nevét tartalmazza, mivel azok is egyediek, nem változnak a játékok között, ráadásul a csv-ból való betöltésnél is az egyes területeknél a régió neve van megadva.



3.3. ábra. Az adatbázis teljes tervezete.

4. fejezet

Mesterséges intelligencia tervezése

A mesterséges intelligencia integrálása a játékba egy összetett folyamat, amely több különböző lépést igényel. Az MI feladata a játékomban olyan stratégiai döntéseket biztosítani a játékos számára az invázió fázis folyamán, amelyek hosszútávon segítenek a játék megnyerésében. A fejlesztés során több szempontot is figyelembe kellett venni, mint például a szerverrel történő kommunikációt, a környezet megfelelő modellezését, illetve az ügynök és neurális hálózat fejlesztését.

4.1. A neurális hálózat és az ügynök

Az egyik legfontosabb lépés a megfelelő könyvtár/keretrendszer kiválasztása az ügynök és a neurális hálózat számára. Ezekre már többféle megoldás létezik, a kutatásom során ezek közül kettőt néztem meg mélyebben: a TensorFlow.js-t és a PyTorch-ot.

A TensorFlow.js egy olyan gépi tanulásos könyvtár, ami JavaScript nyelven alapul. A segítségével futtathatunk meglévő modellekkel, újra taníthatjuk őket, vagy akár írhatunk sajátokat is JavaScript használatával. A meglévő modellek futtathatjuk webes, illetve böngészőalapú alkalmazásainkban és Node.js környezetben különböző feladatok megoldásához. Ilyen feladatok például a képosztályozás, arc- és tárgyfelismerés vagy különböző nyelvi kérdések megválaszolása. Ezek által kiváló megoldást jelenthet webalapú játékok esetén is. Meglévő modell újratanítása a társasjátékom esetében nem volt opció, mivel a megerősítéses tanuláson alapuló modellek általában specifikus feladatokhoz készülnek, ezért egy meglévő modell alkalmazása a játékomban nem feltétlenül eredményes. A különböző játékokban eltérő környezetekkel és döntési struktúrákkal találkozunk, amelyek miatt egy előre betanított modell aligha illeszkedne jól a játékom szabályaihoz. A TensorFlow.js-en belüli Reinforcement Learning használata meglehetősen bonyolult lenne, mivel jelenleg nem érhető el kifejezetten az RL-nek dedikált könyvtár, mint például a Python-alapú TensorFlow tf-agents csomagja. A TensorFlow.js lehetővé teszi a gépi tanulási modellek JavaScript-ben való használatát, azonban RL specifikus implementációkat nem biztosít. [6]

A PyTorch egy nyílt forráskódú gépi tanulási keretrendszer, amelyet elsősorban a Facebook AI Research (FAIR) fejlesztett ki. A PyTorch népszerűsége főként a dinamikus számítási gráfoknak, a könnyen használható API-nak és a hatékony GPU-gyorsításnak köszönhető. Az egyszerű szintaxis és a rugalmas modelllépítési lehetőségek miatt különösen kedvelt a kutatók és fejlesztők körében. A PyTorch támogatja a mély neurális hálózatok különböző architektúráit, és lehetőséget biztosít a modell tanítására, validálására és kiértékelésére. A megerősítéses tanulás területén a PyTorch egyik legnagyobb előnye, hogy kompatibilis több RL-keretrendszerrel, például a Stable-Baselines3-mal és az RLLib-bel, amelyek már előre implementált algoritmusokat tartalmaznak. Emellett elérhető a TorchRL környvtár, amely kifejezetten megerősítéses tanulási modellek fejlesztésére készült, és támogatja a legnépszerűbb algoritmusokat, mint a DQN. [7]

Általánosan egy megerősítéses tanulást alkalmazó keretrendszer olyan ügynököt használ, amelyek célja egy virtuális környezetben való optimalizált döntéshozás. A döntéshozatal mellett tanulnak az elvégzett akciók és a velük járó jutalmak közötti kapcsolatokból, ezáltal fejlesztve a jövőbeli optimális döntéshozatalukat. Az ügynökök többféle módszert alkalmazhatnak, például: DQN, REINFORCE, DDPG, TD3, PPO, SAC.

4.1.1. Folyamatos és diszkrét akciótér

A megfelelő ügynök kiválasztása előtt meg kell vizsgálni egy fontos kérdést. A gépi tanulásban fontos fogalmak a diszkrét, illetve folytonos akcióterek. [8] Ezek nem más, mint a döntési lehetőségek típusát jelzik. Egy diszkrét akciótérben (discrete action space) az ügynök egy véges, meghatározott számú akció közül választhat, például mozoghat jobbra, balra, fel vagy le egy rácson. Jól alkalmazható olyan játékoknál, ahol a lépések száma korlátozott, mint a sakknál. A folytonos akciótérben (continuous action space) az ügynök tetszőleges, folytonos értékeket választhat egy adott tartományban, például gyorsulási fokot vagy kormányzási szöveget egy autós szimulációban. Az én játékomat tekintve az ügynök meghatározott lehetőséggel rendelkezik: kiválaszthatja, hogy melyik területre helyez seregeket, honnan támad és mennyi sereggel, stb. Míg a sereg mennyiségének kiválasztása közelíthet a folytonos akciótérhez, a lépések száma és jellege jól leírható diszkrét döntési lehetőségekkel, így feltehetőleg a diszkrét módszerek (például DQN vagy PPO) megfelelőek lesznek.

4.1.2. DQN és PPO

Mind a DQN (Deep Q-Network - Mély Q-Hálózat), és a Q-tanulás a megerősítéses tanulás módszerei közé tartozik. A Q-tanulás egy algoritmus, ahol az ügynök rendel egy úgynevezett Q-értéket minden lehetséges állapot-akció pároshoz. minden lépésnél

az algoritmus frissíti ezeket az értékeket annak alapján, hogy az adott akcióért milyen jutalmat kapott. A cél az, hogy az ügynök a lehető legjobb, azaz a legnagyobb Q-értékkel rendelkező akciót válassza, ezáltal optimalizálva a hosszútávú döntéseit. A Q-tanulás segítségével az ügynök lépésről lépésre, iteratívan fejleszti a meghozott döntéseit, amíg el nem éri a maximális értékű (optimális) stratégiájának kifejlesztését. Ezen gondolatot kiegészítve, a DQN továbbfejleszti a hagyományos Q-tanulást mély neurális hálózatok alkalmazásával. Míg egy Q-tanulást alkalmazó ügynök táblázatot használ az egyes állapot-akció párok Q-értékeinek tárolására, a DQN egy mély neurális hálózatot használ a Q-függvény megközelítésére. Ezáltal az ügynöknek lehetősége nyílik olyan komplexebb környezetekben is hatékonyan tanulni, ahol egyébként a lehetséges állapotok száma túl nagy lenne egy hagyományos Q-tanulás táblázatos megoldásának. A DQN tanulási folyamata során az ügynök folyamatosan frissíti a neurális hálózatának súlyait annak érdekében, hogy minél pontosabban megjósolja az egyes akciók hosszú távú jutalmát. Kijelenthetjük, hogy minden módszer célja az ügynöknek egy optimális döntéshozó képesség kifejlesztése, azonban a DQN ideálisabb megoldást nyújt egy komplex, sok állapottal rendelkező környezetben. Mivel a DQN mély neurális hálózatot használ a Q-értékek kiszámításához, azaz nem kell minden egyes állapot-akció párról egyedi adatot tárolni, ezért egy hatékonyabb, skálázhatóbb megoldást jelent.

A többi ügynököt vizsgálva még a PPO (Proximal Policy Optimization) eshet számításba, ugyanis a többi a folytonos akcióterekben nyújt megoldást. A PPO Policy-alapú tanulást alkalmaz, úgynevezett policy-kat (döntési szabályokat) fejleszt. Ezek a policy-k megmondják az ügynöknek, hogy milyen lépést válasszon az adott helyzetben. Célja, hogy egy stabil, könnyen beállítható módszert biztosítson a tanuláshoz, korlátozva a policy változásának mértékét, így megelőzve a gyakori, instabil lépésváltozásokat. Eme korlátozás eléréséhez az úgynevezett "clipping" mechanizmust alkalmazza, amely biztosítja, hogy a policy frissítése ne legyen túl nagy egy adott irányban. Ha a változás mértéke meghalad egy bizonyos küszöböt, a klip hatására a policy frissítés elutasítja az optimális szintet meghaladó változásokat így megakadályozva a túltanulást, instabilitást. [9]

4.2. Kommunikáció a szerverrel

Mivel a szerverem JavaScript nyelven íródott, illetve az MI Python nyelven kerül megvalósításra, valahogyan meg kell oldanom, hogy a kettő tudjon kommunikálni egymással. Az egyik megoldás a Python alkalmazáson belül egy REST API létrehozása (például Flask keretrendszerrel), ahol a Node.js szerver HTTP-kéréseket küldene, és a Python API pedig visszaadná az eredményeket. Ezzel a megoldással az a probléma, hogy nem valós idejű kommunikációra optimalizált. Ezzel szemben hatékonyabb megoldást nyújt a WebSocket kapcsolat. A Python és a Node.js között egy kétirányú, valós idejű kom-

munikációt biztosíthatunk a segítségével. Ez egy aszinkron adatcserét eredményez, ami hosszabb adatok küldésére és fogadására a legoptimálisabb. A Python oldalon hasonlóan a kliensekhez a WebSocket kapcsolat úgy van megoldva, hogy csatlakozik a szerverhez a Python kliens. A frontend kliensekkel ellentétben azonban a Python kliens minden kérésnél hozzáad egy random generált request_id-t, amelyet a szerver figyelembe vesz. Ez azért fontos, mivel a Python részen nem szükséges megkapni minden üzenetet, amit a szerver folyamatosan küldözik például egy új játék létrehozásánál. Ezzel a megoldással a Python kliens egy olyan üzenetre vár, ami tartalmazza a kérése azonosítóját, és csak akkor megy tovább, amikor a megfelelő választ megkapta. 4.1

```

1  while True:
2      response = await self.websocket
3          .recv()
4      response_data = json.loads(
5          response)
6
7      print("Received response:",
8          response_data.get("action"))
9
10     if response_data.get(
11         "request_id") == request_id:
12         print("Valid response")
13         return response_data.get(
14             "data")
15     else:
16         print("Received response"
17             "with mismatched"
18             "request_it, waiting for"
19             "correct one...")

```

4.1. Python kliens várakozása a megfelelő válaszra.

Végül a játék mesterséges intelligenciájának működéséhez egy FastAPI-alapú szerver is fut, amely fogadja a kliens kéréseit és visszaadja az AI által választott akciót. Ez a frontend kliensek miatt került bele a játékba, hogy tudják kérni az eddig betanított AI segítségét. A szerver a uvicorn segítségével futtatható, amely egy aszinkron webkit-szolgáló Pythonhoz, és különösen jól működik az async és await alapú alkalmazásokkal. A FastAPI segítségével egy REST API készült, amely egy /predict/ végpontot biztosít. Ez a végpont fogad egy JSON formátumú kérést, amely tartalmazza a játék állapotát és a lehetséges akciókat. Az API először a bemeneti adatokat egy megfelelő formátumba alakítja (_flatten_state függvény), majd meghívja az ügynököt, amely a tanult modell

alapján kiválasztja a legjobb akciót. A választott akciót az API visszaküldi a kliensnek, így a játék képes az MI döntései alapján továbbhaladni.

4.3. Környezet tervezése

A mesterséges intelligencia döntéshozatali folyamatainak tanításához elengedhetetlen egy jó környezet, amelyből tanulhat. Ennek a környezetnek pontosan kell szimulálnia a játék adott mechanizmusát, különben az MI rossz mintákat tanul meg, amelyek a valódi játékban irrelevánsnak számíthatnak. Ezért egy egyedi megvalósítást kellett megalkotni, amely pontosan követi a játék lépésein, szabályait. A Gym nevezetű könyvtárat használtam fel kiindulási alapnak, amely lehetővé teszi az egyedi megerősítéses tanulás környezetek kialakítását. A Gym-en belül is az Env osztályt írtam felül, amely alapjáraton a következő függvényeket tartalmazza: init, step, reset és render.

Az *init* függvényben a környezet legfontosabb mezői kapnak értéket. Az egyik ilyen a megfigyelési tér (Observation Space), amely a játék aktuális állapotát modellez. Az MI a döntéshozatalhoz öt információt használ. Az ownership, amely a területek birtoklását jelenti, értéke lehet 1, ha az AI birtokolja és 0, ha egy ellenfele. Az army_counts, amely az egyes területeken lévő seregeket reprezentálja. Értéke az adott területen lévő seregek száma osztva egy maximum értékkal. Az attackable mutatja, hogy mely területek azok, amelyekről az AI indíthat támadást jelen pillanatban. Az utolsó két értékek pedig az is_my_turn és a round_state, amelyekkel ellenőrizhető, hogy valóban az AI támadási köre van jelenleg. Másik fontos mező amely értéket kap az akciótér (Action Space), amely a végrehajtható akciókat tartalmazza dinamikusan. Ezek az akciók minden lépés után a get_valid_attacks() függvényrel (4.2) kerülnek meghatározásra. Ezeknek a maximuma a területek száma szorozva a területek száma mínusz eggyel, szorozva a maximális seregekkel, majd a végén plusz egy, a passzolási lehetőség miatt.

```
1 def _update_action_space(self):
2     self.actions = self.
3         get_valid_attacks() + [("pass",)
4     ]
5     self.action_space = spaces.Discrete
6         (len(self.actions))
7
8 def get_valid_attacks(self):
9     valid_attacks = []
10    for from_territory in self.
11        get_ai_owned_territories():
12            for to_territory in self.
13                get_attackable_neighbours(
14                    from_territory):
```

```

8             _max_armies = self.
9                 get_max_attack_armies(
10                from_territory)
11
12             for armies in range(1,
13                 _max_armies + 1):
14                 valid_attacks.append((
15                     from_territory,
16                     to_territory, armies
17                 ))
18
19             return valid_attacks

```

4.2. Akciótér beállítása.

A *reset* függvény felel a környezet újrakezdéséért, minden lejátszott játék után ez a függvény kerül meghívásra a tanítás során. A WebSocket-en keresztül küld a szervernek egy új játék kezdése kérést, majd a válasz alapján beállítja a kezdéshez szükséges adatokat. A legnagyobb különbség az egyes állapotok lekérése és a legelső állapot lekérése között az a statikus adatok beállítása. Ilyen adatok a kikötők és várak elhelyezkedése a területeken, vagy a szomszédsági mátrix (egy $n \times n$ -es mátrix, ahol az n a területek száma, a mátrix értékei pedig azt jelzik, hogy a két terület szomszédos-e), ugyanis ezek nem változnak a játék folyamán.

A *step* az egyetlen függvény, amely egy paramétert is vár. A paraméter a választott akció sorszáma, amelyet kikeret a környezet az elérhető akciókból. Miután ellenőrizte, hogy valóban a kör helyes szakaszában van és a lépés is létezik, megnézi, hogy a választott lépés a passzolás-e. Ha igen, akkor az automata-kör kérést küldi a szervernek, ha pedig nem, akkor egy csata-kezdése kérést küld, csatolva a támadó és védekező területek azonosítóját, a játékos azonosítóját és a seregek számát. Mindkét esetben egy jutalom értéket vissza a szervertől, amely ha meghalad egy bizonyos értéket, akkor befejezettnék tekinti az adott játékot. Ha ez az érték nem elég nagy, azaz még folyamatban van a játék, akkor pedig a jutalom értéke mellett a következő állapotot is visszaadja az MI-nek. Amikor egy automata-kör kérés érkezik a szerver számára, akkor a seregek lehelyezése egy algoritmus alapján megy végig, csak egyesével lepakolja őket a területekre sorban. Az ellenfél köre pedig szintén egy egyszerű algoritmus alapján szimulálódik le: megnézi a birtokolt területeket, azok közül is azokat, amelyek rendelkeznek ellenséges szomszéddal, majd megnézi hogy ezen a területen van-e elég sereg a biztonságos támadáshoz. Csak akkor támad, ha legalább három serege van, és ha a saját seregek száma legalább kétszer annyi, mint a védekezőé. A támadó seregek számát úgy választja meg, hogy a támadók száma a maximum sereg felének felel meg, de legalább kettő kell, hogy legyen, de nem támadhat annyi sereggel, hogy csak egy maradjon a területen. Ha az így kiválasztott seregek száma nagyobb, mint a védekező seregek száma, akkor támadást indít. Ha egy támadás elindul, vagy nem talál egy megfelelőt sem, akkor az algoritmus

leáll és visszatér, majd újra az MI következik. Magyarán csak akkor támad, ha biztos fölénye van, és igyekszik minimalizálni a kockázatot.

A *render* valójában a környezet vizuális megjelenítéséért lenne felelős, azonban a játékomhoz már létezik frontend a kliensek számára, ezért az a függvény nem került implementálásra a környezeten belül.

4.4. Tanítás és fejlesztés

Az inváziók eldöntésére létrehozott mesterséges intelligenciát tehát egy mély Q-hálózat alapú tanulóalgoritmus irányítja, amely megerősítéses tanulást tartalmaz. A rendszer egy neurális hálózatot használ, amely tapasztalatokból tanul, egy replay buffer segítségével. A fejlesztés során a tanulási folyamat több komponensre oszlik: a környezet (és annak interfészei), a tanuló ügynök, valamint a mély Q-hálózat. Mindezt pedig egy *train.py* fájlban összesítem egy DQNTrainer osztályban.

Környezet (Environment): A tanulási folyamat egy játékbeli szimulált környezetben történik, amelyet az *AttackEnvironment* osztály definiál (Bővebben lásd: 4.3. fejezet). Ez az osztály biztosítja a környezet interakcióit, az elérhető akciók listáját, valamint a környezet visszacsatolásait (jutalmakat).

DQN ügynök (DQNAgent): A tanulás kulcseleme a *DQNAgent* osztály, amely az alábbiakat kezeli:

- **Akció kiválasztása:** Ez a függvény felelős azért, hogy az AI kiválasszon egy akciót a jelenlegi állapot alapján. A *num_valid_actions* azt adja meg, hogy az adott pillanatban hány érvényes akció közül lehet választani. Az epsilon-greedy stratégiát alkalmazza, hogy az ügynök kezdetben véletlenszerű akciókat válasszon, majd idővel egyre inkább a tapasztalatokra alapozott döntéseket hozzon. Ha egy véletlen szám (0 és 1 között) kisebb, mint az epsilon, akkor véletlenszerű akciót választ. Máskülönben a neurális hálózat által becsült Q-értékek alapján választja ki a legjobb opciót. A state tömböt átalakítja az *unsqueeze* metódussal a neurális hálózat számára értelmezhető adattá, majd meghívja a Q-hálózatot (*self.q_network*) az állapot bemenettel, és visszaadja a hálózat által becsült Q-értékeket. Mivel a hálózat több Q-értéket is kiszámol, ezért a *valid_q_values*-ba levágjuk azokat az értékeket, amelyek nem tartoznak érvényes akciókhöz. Végezetül kiválasztja a legnagyobb Q-értékkel rendelkező akció indexét (ez a legjobbnak becsült akció), és a *.item()*-mel kinyerjük a Python-integer értéket (lásd: 4.3).

```
1  def choose_action(self, state,
2                      num_valid_actions):
3                  if np.random.rand() < self.
4                          epsilon:
```

```

3         return random.randint(0,
4                                         num_valid_actions - 1)
5     else:
6         state_tensor = torch.
7             FloatTensor(state).
8             unsqueeze(0)
9         with torch.no_grad():
10             q_values = self.
11                 q_network(
12                     state_tensor,
13                     num_valid_actions)
14             valid_q_values = q_values
15                 [:, :num_valid_actions]
16             return torch.argmax(
17                 valid_q_values).item()

```

4.3. Az ügynök akció választása.

- **Tapasztalatok tárolása:** A tanuláshoz szükséges múltbeli tapasztalatokat egy replay bufferben tárolja.
- **Tanulási folyamat:** A Q-értékeket folyamatosan frissíti a célhálózattal való kontraszt alapján.
- **Célhálózat frissítése:** Fenntart egy állandóan frissített célhálózatot, amely a tanulás stabilitását biztosítja.
- **Modell mentése és betöltése:** A betanított modellt egy pth formátumú fájlban tárolja, amelyet bármikor be lehet tölni, hogy az eddigi tapasztalatokat tovább lehessen tanítani, vagy csak a döntését kérni. Mint a 4.4 kód részletben is látható, nem kell az egész modellt elmenteni, elég a Q-Hálózat state_dict mezőjét, rugalmasságot biztosítva ezzel.

```

1 def save_model(self, filepath="dqn_model_checkpoint.pth"):
2     torch.save(self.q_network.
3                 state_dict(), filepath)
4     print(f"Model saved to {filepath}")

```

4.4. A tapasztalatok mentése.

Mély Q-Hálózat (DeepQNetwork): A DQN alapja egy mély neuronhálózat, amelyet a DeepQNetwork osztály valósít meg. A hálózat egy bemeneti rétegből, két

rejtett teljes összeköttetésű rétegből (és ReLU aktivációkból), valamint egy kimeneti rétegből áll, amely az egyes akciókhöz tartozó Q-értékeket szolgáltatja. A bemenetek dimenziója a játékbeli állapot adataira alapozott, a kimenetek száma pedig az elérhető akciók maximumát adja meg.

Tanulási folyamat (DQNTrainer): A tanulási folyamatot a *DQNTrainer* osztály valósítja meg, amely a *train.py* fájl fő komponense (bővebben lásd: 4.5 kódrészlet). Az osztály inicializálásakor megkapja a környezetet, a bemeneti dimenziót (állapothalmazt), a maximális akciók számát, valamint a WebSocket kapcsolatot, amelyet átad a környezetnek a szerverhez való csatlakozáshoz. A *train()* függvény aszinkron módon fut, először megpróbál kapcsolódni a szerverhez a WebSoketen keresztül, majd betölti a korábban mentett neurális hálózati modellt (ha van ilyen), vagy új hálózatot inicializál, ha nem talál mentést. Ezután a tanulás epizódokban történik: az EPISODES konstans határozza meg, hány játékmenetet (episode) kell lefuttatni. minden epizód elején a környezet visszaáll az alapállapotába az *env.reset()* segítségével, majd a ciklusban az ügynök addig hajt végre akciókat, amíg a játék véget nem ér. Az egyes lépések során lekérdezi az aktuálisan elérhető akciókat a környezettől, majd a *choose_action()* függvényel kiválaszt egyet közülük, epsilon-greedy módon. A kiválasztott akció hatását a környezet *step()* függvénye értékeli ki, és visszaadja a következő állapotot, a kapott jutalmat, valamint egy *done* jelzőt, ami mutatja, hogy vége van-e az epizódnak/játéknak. A tapasztalatot elmenti a replay bufferbe (*store_transition()*), majd az *agent.learn()* meghívásával frissíti a hálózat súlyait a minták alapján. Az epizód végén a teljes jutalom (*total_reward*) kiírásra kerül, és minden 10. epizód után a modell elmentésre kerül (*save_model()*). A főprogram rész (*__main__*) inicializálja a környezetet és a WebSocket kapcsolatot, majd létrehozza a DQNTrainer példányt, és meghívja a *train()* metódust egy aszinkron eseménykezelő segítségével (*asyncio.run*). Ez biztosítja, hogy a tanulás folyamata automatikusan lefusszon és a kapcsolat bezárásra kerüljön a végén.

```
1 EPISODES = 100
2 BATCH_SIZE = 64
3
4 class DQNTrainer:
5     def __init__(self, env, state_dim,
6                  action_dim, websocket):
7         self.agent = DQNAgent(state_dim,
8                               action_dim)
9
10        ...
11
12    @async def train(self):
13        try:
14            await self.websocket.connect()
15
16            ...
17
18        try:
```

```

13             self.agent.load_model()
14
15         except FileNotFoundError:
16             print("No checkpoint found,
17                   starting from scratch")
18
19         for episode in range(self.
20             max_episodes):
21             state = await self.env.
22                 reset()
23             total_reward = 0
24             done = False
25
26             while not done:
27                 valid_actions = self.
28                     env.actions
29                 action = self.agent.
30                     choose_action(self.
31                         _flatten_state(state
32                         ), len(valid_actions
33                         ))
34
35                 next_state, reward,
36                 done, _ = await self
37                     .env.step(action)
38
39                 self.agent.
40                     store_transition(
41                         self._flatten_state(
42                             state), action,
43                             reward, self.
44                                 _flatten_state(state
45                                 ), done)
46
47                 self.agent.learn(self.
48                     batch_size, len(
49                         valid_actions))
50
51                 state = next_state
52                 total_reward += reward
53
54             ...
55
56             print(f"Episode {episode+1}
57                   /{self.max_episodes},"
58                   "Total Reward:{"
59                   total_reward})
```

```
33             if (episode + 1) % 10 == 0:
34                 self.agent.save_model()
35
36         finally:
37             await self.websocket.close()
38     ...
39
40     if __name__ == "__main__":
41         env = AttackEnvironment()
42         ws_url = "ws://localhost:3000"
43         websocket = WebSocketClient(ws_url)
44         env.set_websocket(websocket)
45         ...
46         trainer = DQNTrainer(env, state_dim,
47                               env.max_actions, websocket)
48         asyncio.run(trainer.train())
49         print("Training completed!")
```

4.5. Train.py fájl.

5. fejezet

Az elkészült alkalmazás

Végezetül a játékhoz készült intelligens ügynök is betanítva bekerült az elkészült alkalmazásba egy Let AI decide gomb formájában, hogy segítsen a játékosoknak az optimális akcióválasztásban. Az alábbiakban bemutatom a tanítás során elért eredményeket, illetve az elkészült alkalmazás felületét.

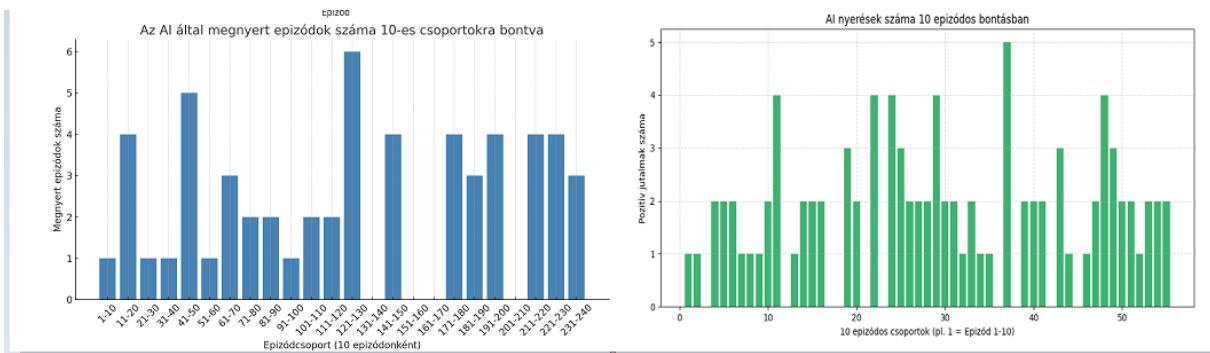
5.1. Tanítás eredménye, pontosság, hiba mértéke

A tanítás során különböző hosszúságú epizódokat futtattam le, amelyek eredményei txt fájlokban kerültek rögzítésre. A cél az volt, hogy megfigyeljem, hogyan alakul a tanulóügynök teljesítménye az epizódok előrehaladtával. A tanítás elején az ügynök pontszámai jelentős ingadozást mutattak. A korai epizódokban gyakran fordul elő -400 és -600 közötti pontszám, amely azt jelzi, hogy az ügynök még nem tudta hatékonyan optimalizálni a döntéseit. Néhány pozitív érték már a kezdeti szakaszban is előfordult, amely a szerencsének is köszönhető. Egy pozitív pontszám minden esetben azt jelenti, hogy az ügynök megnyerte az adott játékot és ezért jóval több jutalmat kapott.

Két főbb tanítási eredményt vehetünk figyelembe a végső elemzéshez: az **eredmények.txt** és a **rewards_until_episode_240.txt** (továbbiakban **rue240**) fájlok tartalmát. Az **eredmények** több kisebb epizóddal rendelkezdő tanítást foglal magában, van benne 30, 50, illetve 100 epizódos tanítás eredménye is. A **rue240** fájl ezzel szemben egy 240 epizódból álló tanítás eredményeit tartalmazza.

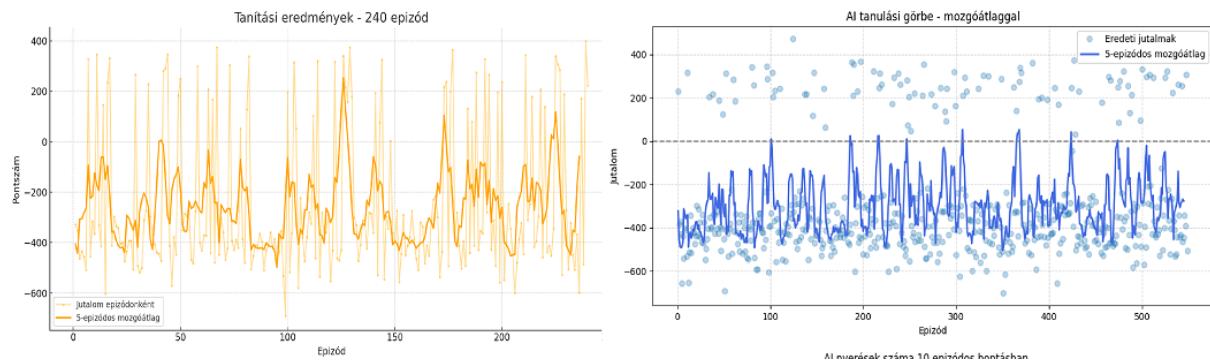
A **rue240** fájl adatai alapján megfigyelhető, hogy a tanítási szakasz végére az ügynök teljesítménye kissé stabilizálódott, és egyre gyakrabban fordult elő nyertes (pozitív jutalommal záruló) epizód. Ezzel szemben az **eredmenyek.txt** több különálló tanítási kísérlet eredményét tartalmazza, amelyek között nem látható határozott javuló tendencia.

Az 5.1 oszlopdiagram az epizódok során mért **nyertes játékok arányát mutatja 10 epizódonként**, külön a két tanítási eredményfájl alapján. Baloldalt a rue240, jobboldalt pedig az eredmenyek fájl adatai láthatók.



5.1. ábra. Pozitív pontszámú epizódok száma 10 epizódonként, összehasonlítva a két tanítással

Az 5.2 ábra egy **mozgóátlagos ábrázolást** mutat, amelyből jól látható, hogy a rue240 (bal) esetében a teljesítmény bár továbbra is szór, összességében valamelyest javul.



5.2. ábra. Epizódonkénti mozgóátlagos teljesítmény: rue240 és eredmények.txt

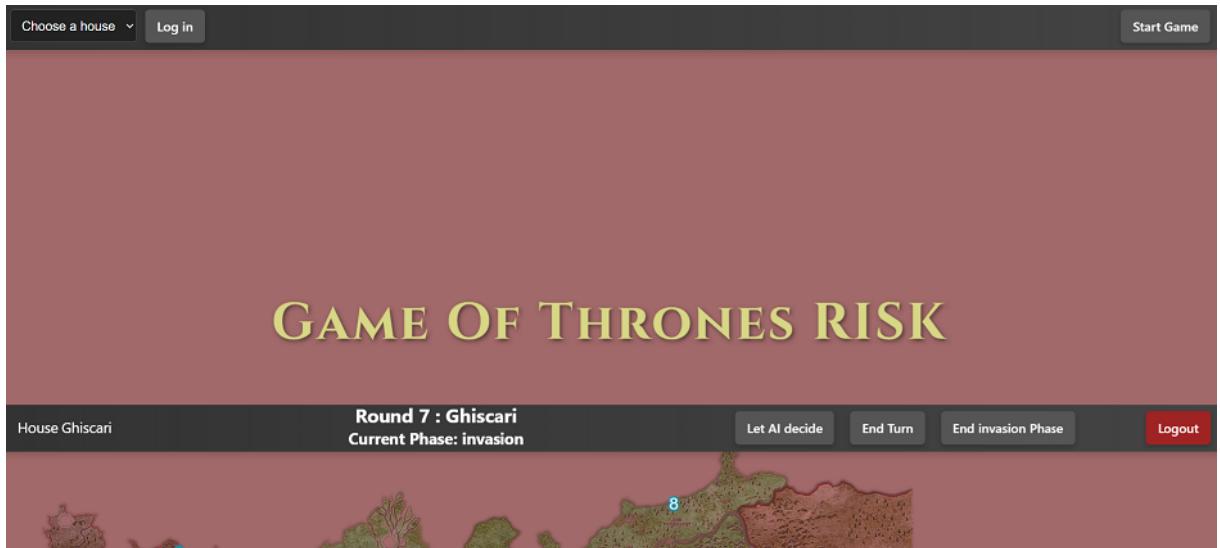
A tanulási folyamat során tehát nem beszélhetünk egyértelmű monoton növekedésről, de időszakosan megfigyelhető a teljesítmény javulása. Az epizódok során keletkező nagy szórást a környezet komplexitása, az ügynök döntési szabadsága és a szerencse eredményezheti. Jövőbeni pozitívabb teljesítmények eléréséhez érdemes lehet finomítani az MI tanulási rátáját vagy hosszabb tanítási időszakokat futtatni.

5.2. Az alkalmazás felhasználói felülete

Az elkészült játék webes felületen keresztül érhető el, és lehetőséget biztosít több játékos számára is, hogy egy közös játéktéren mérkőzzenek meg egymással. Az alábbi képernyőképeken bemutatom az alkalmazás fontosabb részeit.

Főképernyő és játéktér

A főképernyőn látható a játéktér, ahol a játékosok területei, seregei és az aktuális játékfázis jelenik meg (lásd: 5.3). Az egyes mezőkön lévő seregek színezése a birtokló



5.3. ábra. Ki- és bejelentkezett kezdőképernyők.

játékos házától függ. A fejlécen található a vezérlőpanel, ahol a játékos befejezheti a köröket/fázsokat, ki- és bejelentkezhet, illetve új játékot indíthat.

Területek adatai panel

Attribute	Value
Fortress:	X
Port:	X
Region:	The Dothraki Sea
Armies:	8

Select target territory for maneuver:

Vaes Dothrak

Armies to maneuver: 2

Maneuver

5.4. ábra. Manőverezés fázis.

Egy területre kattintva megnyílik az adatok panel, ahol a játékos ellenőrizheti, hogy az adott területet ki birtokolja és mennyi sereggel, továbbá leolvashatja, hogy a terület

milyen bónuszokkal rendelkezik (vár, kikötő). Az adott fázistól függően további elemek jelennek meg itt az éppen soron lévő játékos számára. Megerősítésnél ha saját területre kattintott a játékos, akkor kiválaszthatja, hogy mennyi sereget kíván lehelyezni az adott területre. Inváziót kiválaszthatja azt a szomszédos területet, ahová támadni akar és a támadó seregek számát. Manőver esetében pedig azokat a területeket választhatja, amelyek valamelyen módon összeköttetésben állnak a választott terüettel (lásd: 5.4).

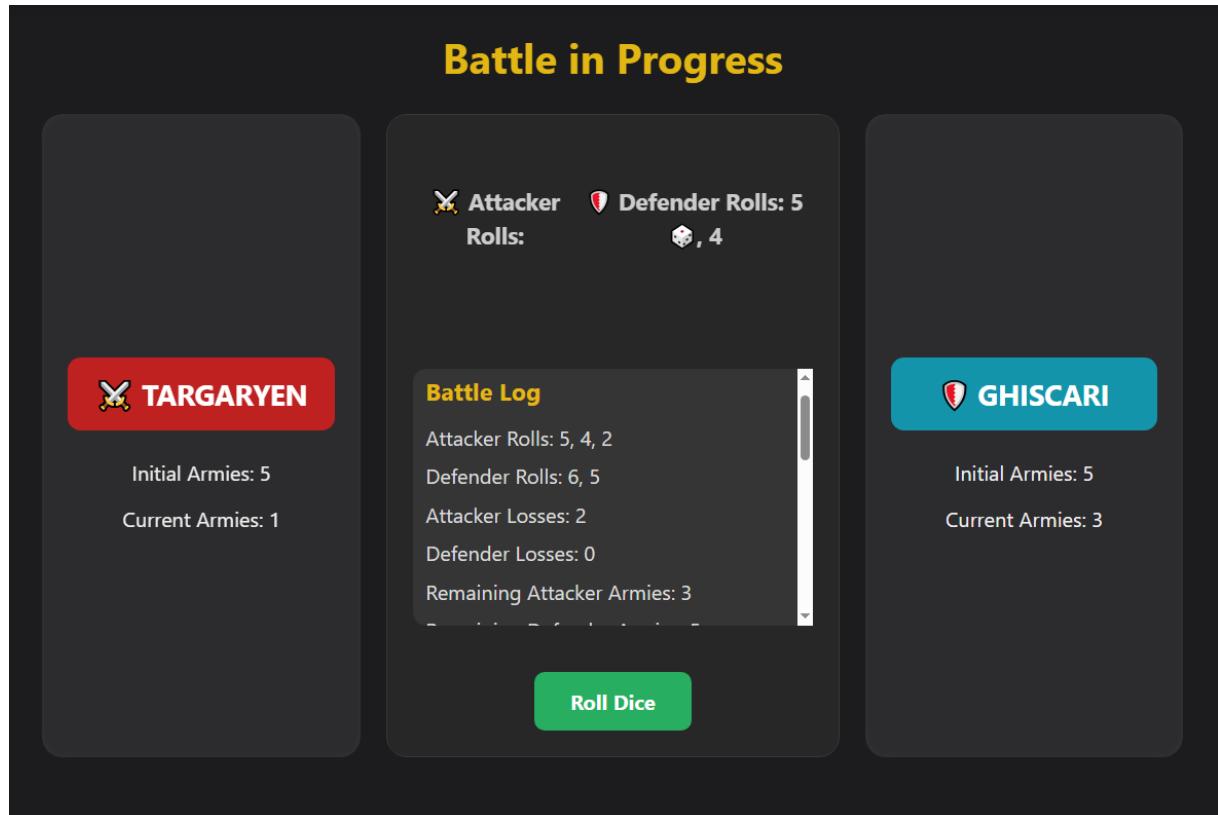
Invázió (Invasion) fázis és AI működés



5.5. ábra. Let AI decide gomb.

Az invázió fázis során a játékosnak megjelenik a Let AI decide gomb (lásd: 5.5), amely az AI tanult modellje alapján választja ki, hogy melyik területről melyik ellenes területre támadjon, és hány sereggel. Az AI döntése real-time történik, az aktuális játékállapotot veszi figyelembe, és a többi játékos figyelemmel kísérheti a támadást.

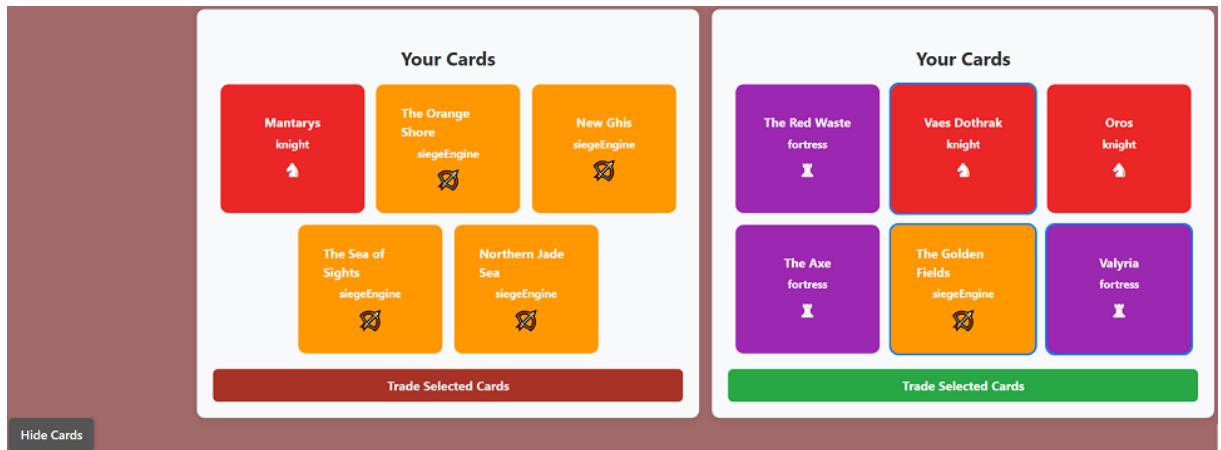
Dobókockák és harc lefolyása



5.6. ábra. Egy folyamatban lévő csata.

Egy invázió megkezdésénél minden játékos számára felugrik egy csata ablak. A csata során mindenki két fél (támadó és véző) dobókockákkal határozza meg a harc kimenetelét. Az ablakban megjelennek a dobott értékek, a seregek változása, valamint a csata logja is (lásd: 5.6). Ez a funkció lehetővé teszi, hogy a játék dinamikus és átlátható maradjon.

Kártyák



5.7. ábra. Beváltható kártyák.

A Show Cards gomb megnyomásával tekintheti meg a játékos a saját kártyáit (lásd: 5.7). A kártyák szimbólumtól függően más-más színűek. Ha összegyűlt három azonos vagy teljesen különböző kártya, akkor azokat a játékos kijelölheti és becserélheti plusz seregekre a megerősítés fázisában. Ekkor ezek a kártyák eltűnnek, majd a Hide Cards gombbal bezárhatja ezt az ablakot.

Összegzés

Az elkészült dolgozatomban egy olyan webes alapú, valós idejű stratégiai játéket valósítottam meg, amely rendelkezik saját mesterséges intelligenciával. A fejlesztés során kifejezetted célon volt egy gördülékeny, vizuálisan és játékélmény szempontjából is élvezhető játék megalkotása, amelyben az emberi játékosok mellett egy AI is szerepet kaphat.

A fejlesztés során számos kihívással szembesülvtem, többek között a több kliens és szerver közötti valós idejű kommunikáció megoldásának problémájával, a komplex játékszabályok formalizálásával, az AI, a környezet és a szerver közötti kommunikáció biztosításával, valamint a megfelelő technológiák hatékony kiválasztásával és felhasználásával. Ezek megoldása során mélyebb ismereteket szereztem mind a gépi tanulás gyakorlati alkalmazásáról, mind pedig a webes játékfejlesztésről.

A tanítási szakasz során különböző epizódszámú tréningeket hajtottam végre, amelyek eredményeiből vizuális ábrázolás segítségével következtetéseket vontam le a modell teljesítményéről. Bár a tanulási görbe nem volt tökéletesen monoton, az ügynök idővel képes volt egyre hatékonyabban reagálni a környezeti ingerekre és bizonyos esetekben sikeres támadási döntéseket hozni, ezáltal játékokat is nyerni.

Úgy vélem, hogy a jövőben a rendszer továbbfejleszthető többféle úton is. A társasjáték része kibővíthető újabb játékmechanizmusok fejlesztésével mint például a pénz és mesterkártyák implementálása, de jelentős változást adhat a térkép kibővítése és a játékosok számának emelése, akár 3-7 játékosra is. A kód nagy része úgy van implementálva, hogy az könnyen módosítható legyen több játékos befogadására is. Egy másik fejlesztési lehetőség újabb mesterséges intelligenciák hozzáadása, például az erősítés vagy a manőverezés fázisához. Ezek által akár az egyik játékos szerepét teljesen át is vehetnék a különböző MI-k, ezzel lehetővé téve, hogy a játéket egyedül is lehessen játszani a gép ellen.

A projekt megvalósítása számomra inspiráló és érvezetes volt. Nemcsak elméleti, hanem gyakorlati tapasztalatokat is szereztem az MI-alapú játékfejlesztés világában, amelyek a jövőbeli tanulmányaim és pályám során is hasznosnak bizonyulhatnak.

Irodalomjegyzék

- [1] ALPAYDIN, ETHEM: Introduction to Machine Learning, MIT Press, 2020.
- [2] LARRY HARDESTY: Explained: Neural networks, MIT News Office, 2017.
- [3] NEURAL NETWORK WIKIPEDIA: [https://en.wikipedia.org/wiki/Neural_network_\(machine_learning\)](https://en.wikipedia.org/wiki/Neural_network_(machine_learning))
- [4] REINFORCEMENT LEARNING WIKIPEDIA: https://en.wikipedia.org/wiki/Reinforcement_learning
- [5] MONGODB WIKIPEDIA: <https://en.wikipedia.org/wiki/MongoDB>
- [6] TENSORFLOW.JS: <https://www.tensorflow.org/js>
- [7] PYTORCH: <https://pytorch.org/>
- [8] ACTION SPACE: <https://medium.com/@corinacataraug/hybrid-action-spaces-in-rl-a-short-overview-5314ac0d62d6>
- [9] PROXIMAL POLICY OPTIMIZATION: <https://www.mathworks.com/help/reinforcement-learning/ug/proximal-policy-optimization-agents.html>

Nyilatkozat

Alulírott *Budai Roland*, büntetőjogi felelősségem tudatában kijelentem, hogy az általam benyújtott, *Stratégiai társasjáték fejlesztése megerősítéses tanulást alkalmazó mesterséges intelligenciával* című szakdolgozat önálló szellemi termékem. Amennyiben mások munkáját felhasználtam, azokra megfelelően hivatkozom, beleértve a nyomtatott és az internetes forrásokat is.

Aláírásommal igazolom, hogy az elektronikusan feltöltött és a papíralapú szakdolgozatom formai és tartalmi szempontból mindenben megegyezik.

Eger, 2025. április 10.



aláírás