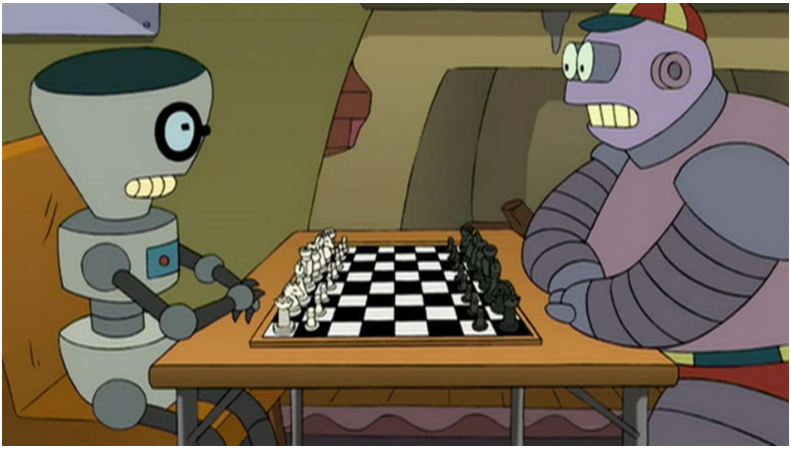
Eötvös Loránd Tudományegyetem
Informatikai Kar
Programozáselmélet és
Szoftvertechnológiai Tanszék

Sakk-bot Verseny

dr. Gregorics Tibor
Tanszékvezető egyetemi docens
Virágh Tibor
Programtervező Informatikus BSc

Budapest, 2019



1. ábra. Robotok sakkoznak a Futurama ©című sorozat egyik epizódjában.

I. rész

Bevezető

1. fejezet

A dolgozatról

1.1. A dolgozat célja

A dolgozat célja a mesterséges intelligencia kurzuson bemutatott **„kétszemélyes, teljes információjú, diszkrét, véges és determinisztikus, zéró-összegű, játékok”** algoritmusainak bemutatása a gyakorlatban, illetve az érdeklődőbb hallgatók bevonása egy olyan játékba, melyen keresztül elmélyíthetik tudásukat ezen algoritmusokból.

E célból a (normál) sakkot vesszük alapul, mely elég bonyolult ahhoz, hogy látványos eredményeket láthassunk, a legtöbben ismerik a szabályait, illetve egy klasszikus példája a fent nevezett játékok halmazának.

A dolgozat "melléktermé" egy olyan *API interface*, melyhez kapcsolódva nem csak mesterséges intelligenciát megvalósító programok,

hanem "sima", emberi játékosok által is használható kliens is írható, így a téma iránt nem érdeklődők is kipróbálhatják magukat például egy mobilalkalmazás vagy egy asztali Windows / Linux alapú kliens megírásában, illetve tapasztalatra tehetnek szert az elosztott rendszerek működésében és az aszinkron programozás feletteb érdekes és néha nagy fejfájást okozó működésében is.

1.2. A dolgozat felépítése

A dolgozat három fő részből áll: jelen bevezető, a fejlesztői és a felhasználói dokumentáció. A fejlesztői dokumentáció négy fő részre tagozódik: **"elemzés"**, **"tervezés"**, **"megvalósítás"** és **"tesztelés"** - míg a felhasználói dokumentáció a programcsomag alkalmazásai mentén van szétvágva, és mindegyiknél szót ejt a **telepítésről**, **használatról** illetve a **karbantartásról**. A dolgozat végén kiegészítésként a függelék tartalmaz néhány a megértést segítő információt a dolgozattal kapcsolatban, illetve néhány érdekességet, tapasztalatokat.

II. rész

Felhasználói Dokumentáció

2. fejezet

Szerver

2.1. Telepítés

A kiszolgáló telepítéséhez Windows környezetben a következő előfeltételek szükségesek:

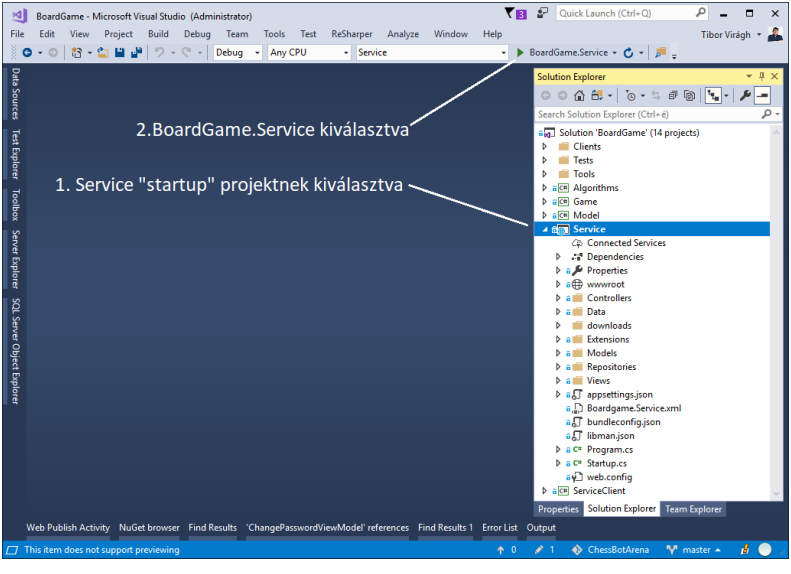
- Telepített IIS,
- Telepített .NET Core 2.1 runtime,
- Telepített .NET Core Hosting Bundle,
- MSSQL adatbázis,
- A klienseknek pedig .NET framework 4.7.2, ami be van építve a telepítőjükbe.

Ezen előfeltételek telepítéséről további információk a Microsoft oldalán található.

2.1.1. Fejlesztéshez, lokálisan

Lokális telepítés lépései Visual Studio-ban:

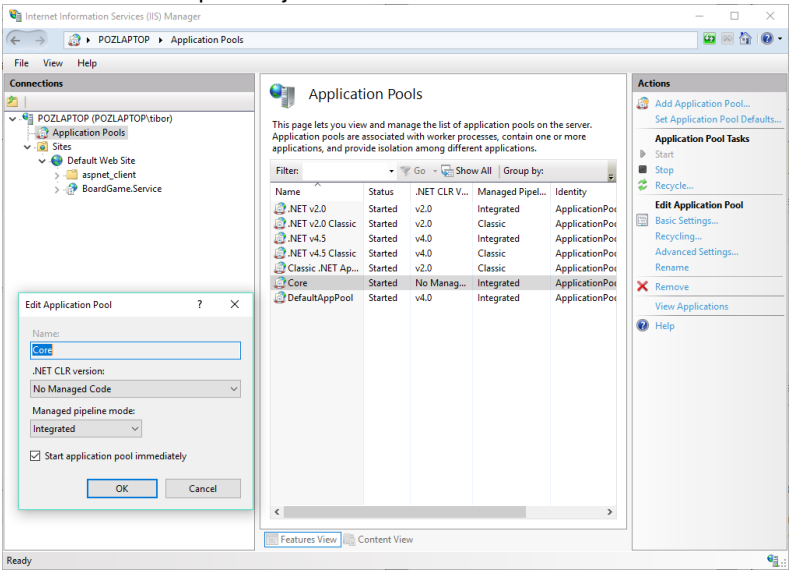
1. Solution megnyitása Visual Studio-ban majd "Service" projekt startup projektként való beállítása:



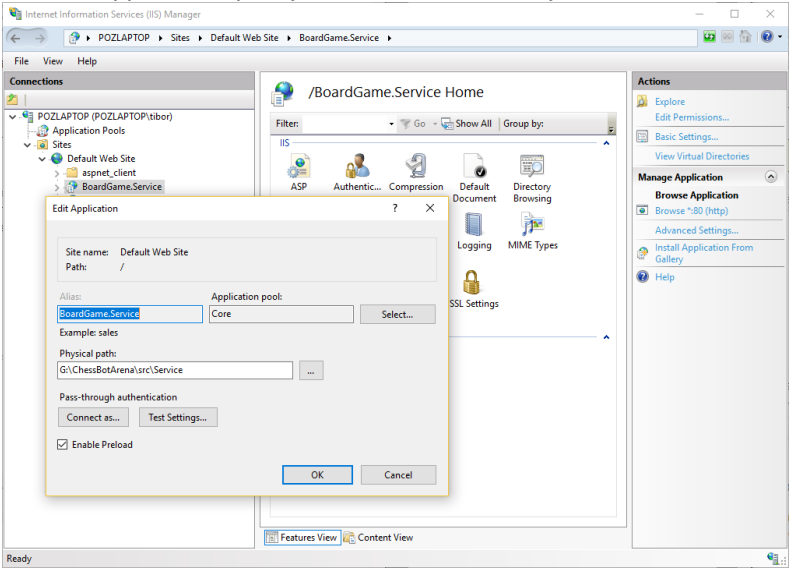
2. Build.

Ekkor a Visual Studio létrehozza a site-os az IIS-ben.

3. Application pool létrehozása ("Core") a .NET core-os projekthez. A beállításokat a kép mutatja:



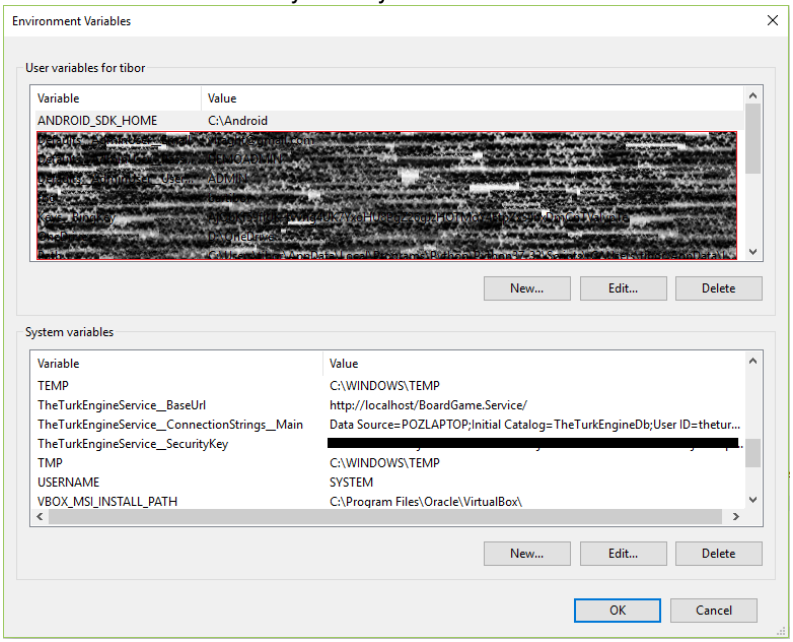
4. Site application pool-jának beállítása az újonnan létrehozottra.



5. Környezeti változók beállítása

A projektben találhatók példa konfigurációs file-ok, mint például a `appsettings.json` file. Ezekbe behelyettesíthető például az adatbázis connection string, de ez esetben fennáll a veszélye annak, hogy véletlen bekerülnek a jelszavaink egy GIT repository-

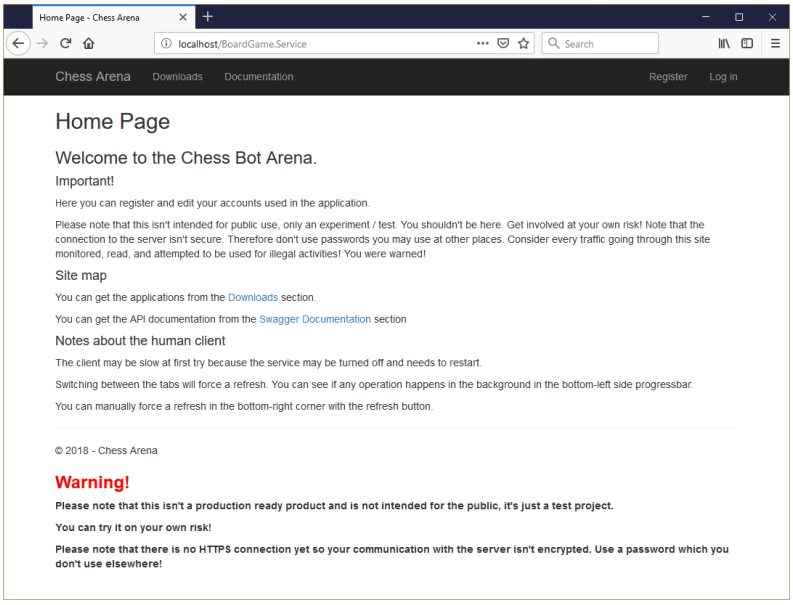
ba. Ezért tanácsos helyi környezeti változókba menteni. Példa:



- **TheTurkEngineService__BaseUrl** Ide kerül a szerver címe.
- **TheTurkEngineService__ConnectionStrings__Main** Ide kerül a connection string.
- **TheTurkEngineService__SecurityKey** Ide kerül a privát kulcs, amivel aláírja a szervíz a JWT tokeneket.

6. Build és futtatás

Ha minden jól megy, akkor a böngészőben meg kell jelennie a frontend-nek:



7. Hiba esetén ellenőrizendő:

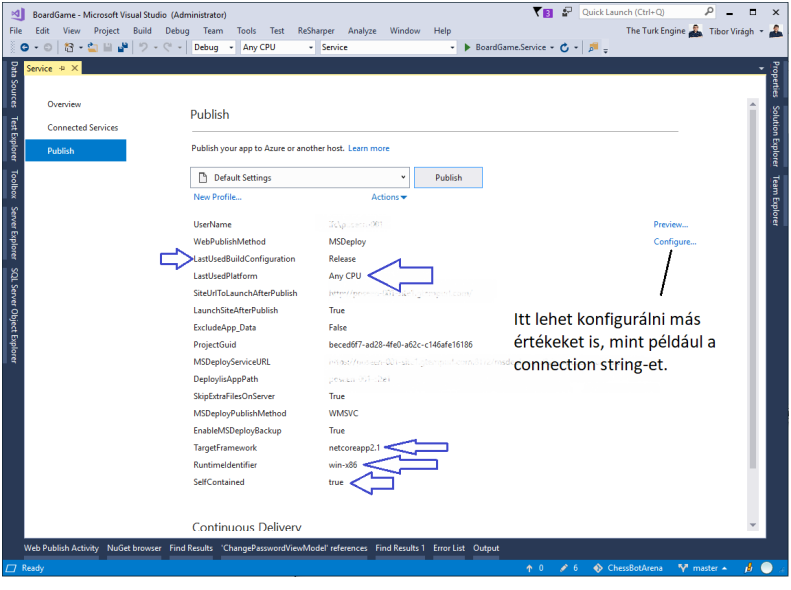
- IIS-ben megfelelő application pool van-e beállítva?
- IISRESET futtatása adminisztátori parancssorban
- Windows Event Viewer tartalmazhat információt a hiba forrásáról

2.1.2. Távoli szerverre

1. Solution megnyitása Visual Studio-ban

2. Jobb klikk a Service projekten, majd "Publish" kiválasztása.

3. Publish ablakban szervernek megfelelő beállítások kiválasztása. Példa:



4. Publish futtatása

5. Belépés szerverre, web.config és appsettings.json ellenőrzése. Ezek módosítása ha szükséges, vagy környezeti változók használata. (Ekkor appsettings.json file törlése.)

6. Ha szükséges, site újraindítása.

7. Hiba esetén ellenőrizendő:

- Távoli IIS-ben megfelelő application pool van-e beállítva?
- Távoli gépen IISRESET futtatása adminisztátori parancssorban
- Távoli Windows Event Viewer tartalmazhat információt a hiba forrásáról

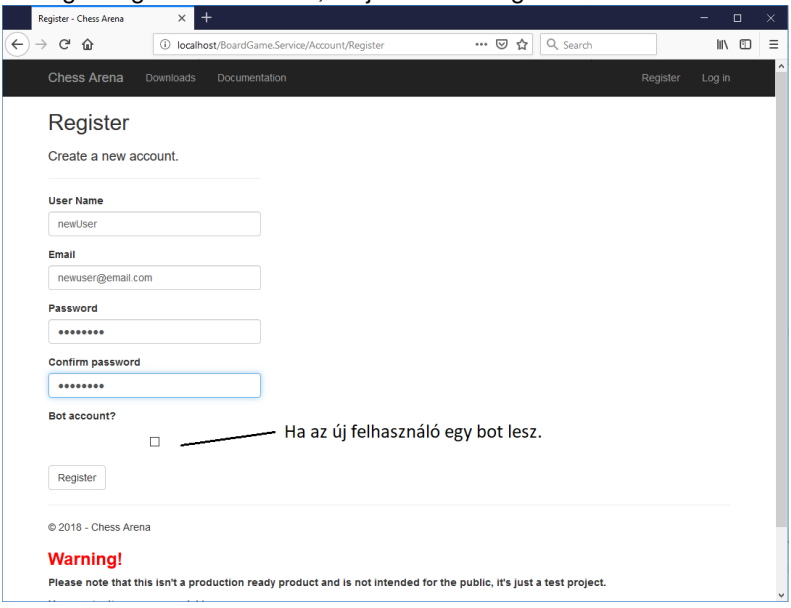
2.2. Használat

A használati útmutató már nem csak rendszergazdáknak és a fejlesztőknek, hanem az átlagfelhasználóknak is szól.

2.2.1. Regisztráció

Az alkalmazást nem lehet regisztráció nélkül használni. Regisztráció lépései:

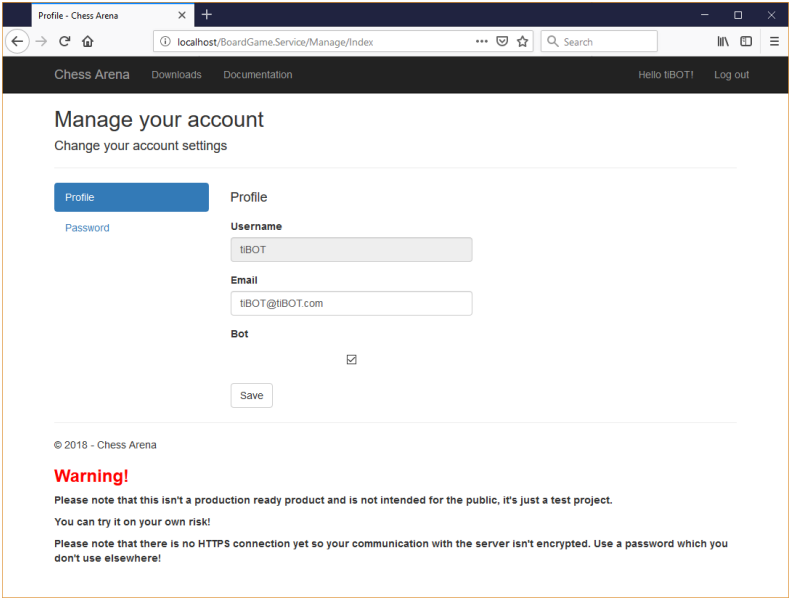
1. Register gombra kattintás, majd adatok megadása



A jelszónak tartalmaznia kell kis és nagy betűt, egy számot és valamilyen jelet.

A "Bot account" alatti checkbox akkor használandó, ha egy olyan felhasználót akarunk regisztrálni, amely botként lesz nyílvántartva.

Ha minden rendben volt, akkor a rendszer bejelentkezteti az új felhasználót és a saját adatainak oldalára visz:



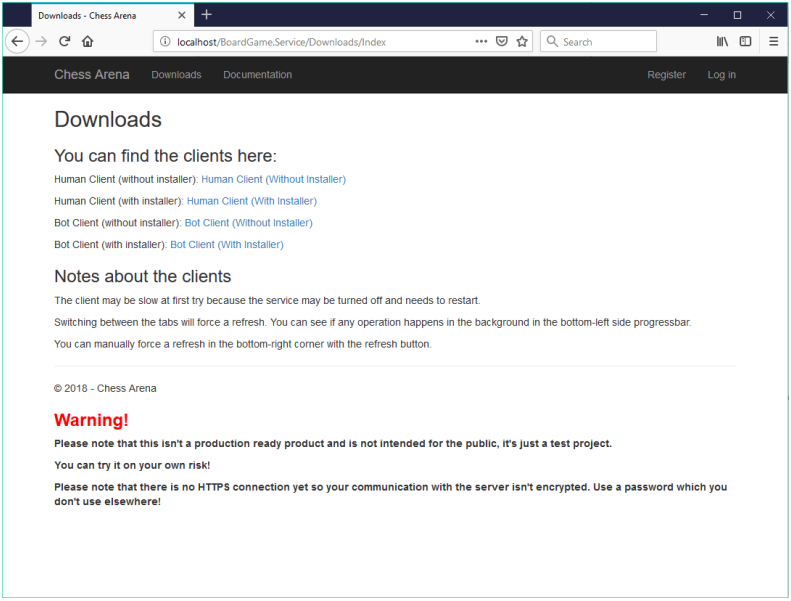
Ezen a képernyőn lehet módosítani a jelszót, e-mail címet, illetve azt, hogy a felhasználó robot-e.

3. fejezet

Játékkliens

3.1. Telepítés

1. A weboldalon kattintás "Downloads" gombra, majd a megfelelő verzió letöltése

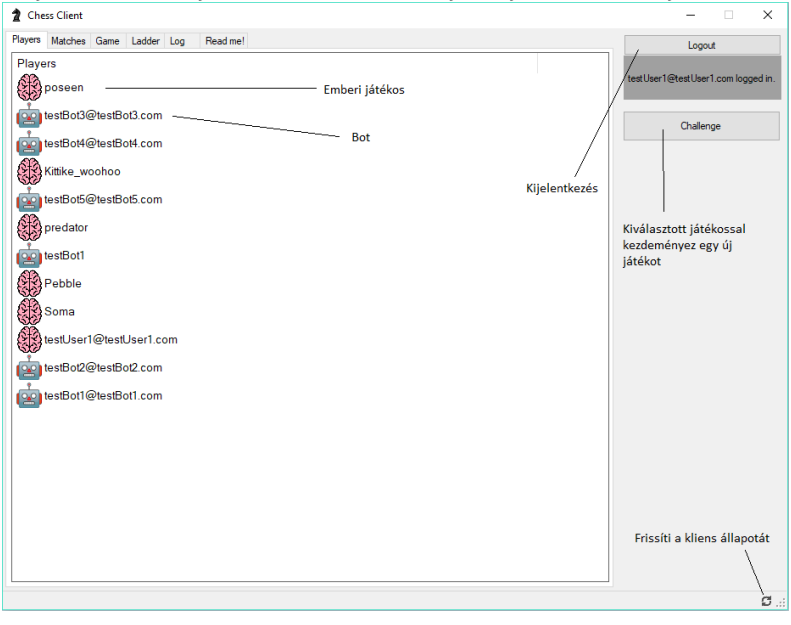


2. A "Without Installer" verzióban csak a futáshoz szükséges file-ok vannak betömörítve. Ezt a ZIP file-t letöltve és kitömörítve el lehet indítani a klienst.

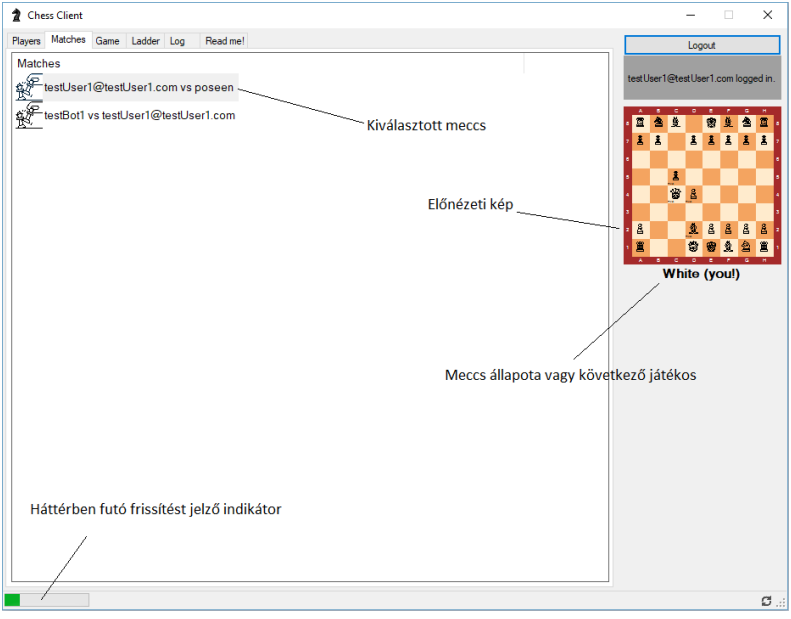
A "With Installer" verzió tartalmazza a telepítőt, amely installálja a .NET Framework 4.7.2-es verzióját, ha az nincs még telepítve. Ehhez csak ki kell tömöríteni a file-t és a Setup.exe-t elindítani és követni az utasításokat.

3.2. Használat

1. Játékot elindítva a fő ablakba érkezünk, ahol be tudunk jelentkezni, majd a sikeres bejelentkezés után láthatjuk a játékosok listáját:

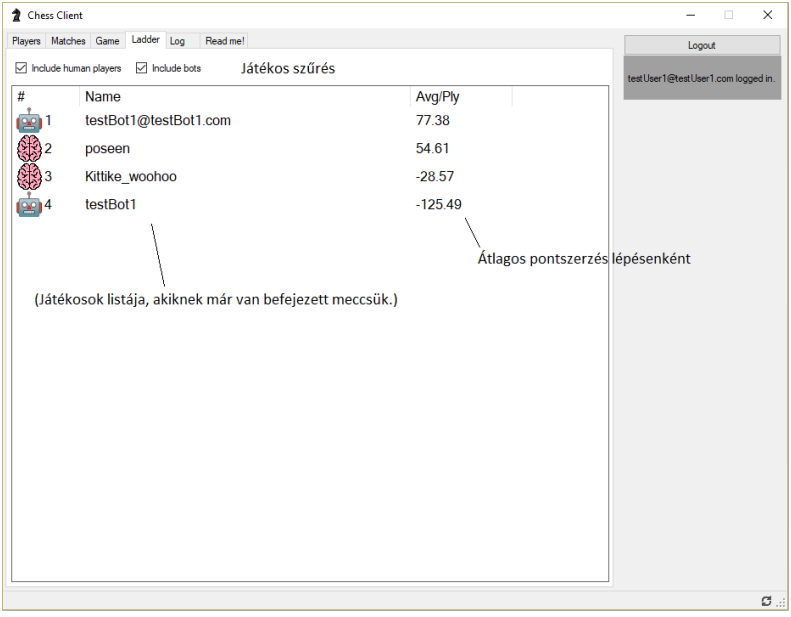


2. A második tabra átkattintva láthatjuk a meccseink listáját. Ezekre rákattintva a jobb oldalon egy kis előnézeti ablakban láthatjuk az ottani állást, illetve ezzel aktiválunk egy meccset arra, hogy a következő tabon megnézzük vagy esetleg lépjünk:



3. A harmadik oldal maga a játéktér. Először kiválasztjuk azt a bábut, amivel lépni szeretnénk, majd a megfelelő mezőre kattintva adjuk meg a lépést. A képernyő jobb oldalán látható lista tartalmazza az elmúlt lépéseket, illetve felette a speciális parancsokat.:

4.A negyedik oldalon látható az eddigi toplista. Csak azok látszódnak, akiknek már van befejezett játékuk. Ez a képernyő elérhető bejelentkezés nélkül is:



3.3. Hibaelhárítás

- Log fül tartalmazza minden kivétel szövegét, így innen esetlegesen meg lehet állapítani, mi romlott el.
- Windows Event Viewer is tartalmazhaz információkat

- Néha a szerver leáll ha sokáig nem érkezett kérés. Ekkor az első kérés eltarthat egy ideig. A kliensnek le kell kezelnie ezt a problémát. Ha mégsem sikerül, akkor később újra probálkozik a frissítéssel.

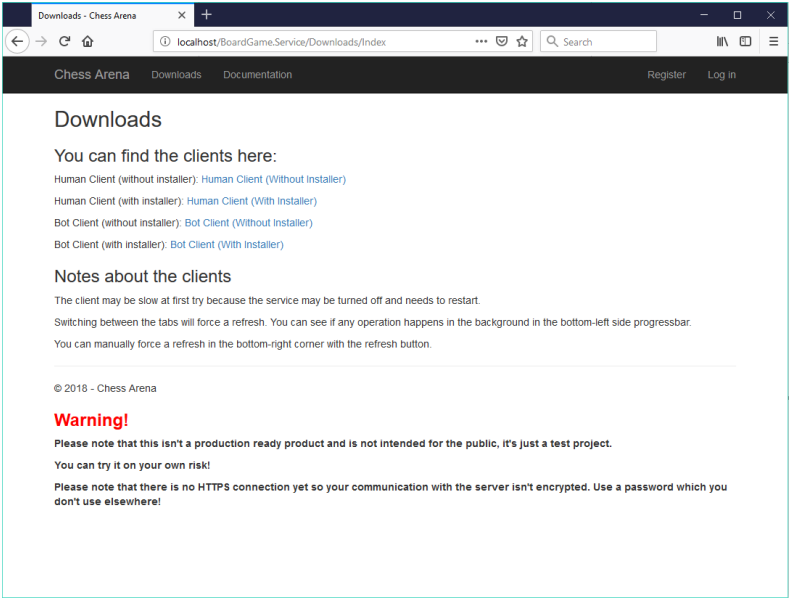
- Ha végképp nem érkezik válasz, akkor érdemes megnyitni a szerver weboldalát. Ha az nem megy, akkor maga a szolgáltatás is áll.

4. fejezet

Botkliens

4.1. Telepítés

1. A weboldalon kattintás "Downloads" gombra, majd a megfelelő verzió letöltése

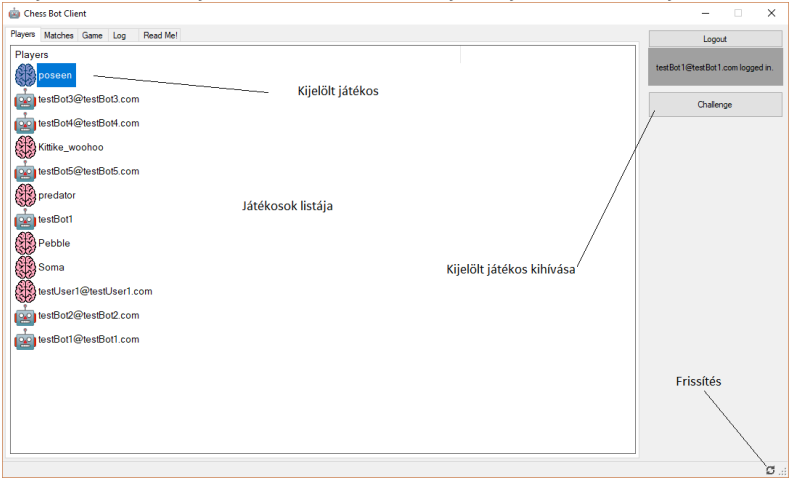


2. A "Without Installer" verzióban csak a futáshoz szükséges file-ok vannak betömörítve. Ezt a ZIP file-t letöltve és kitömörítve el lehet indítani a klienst.

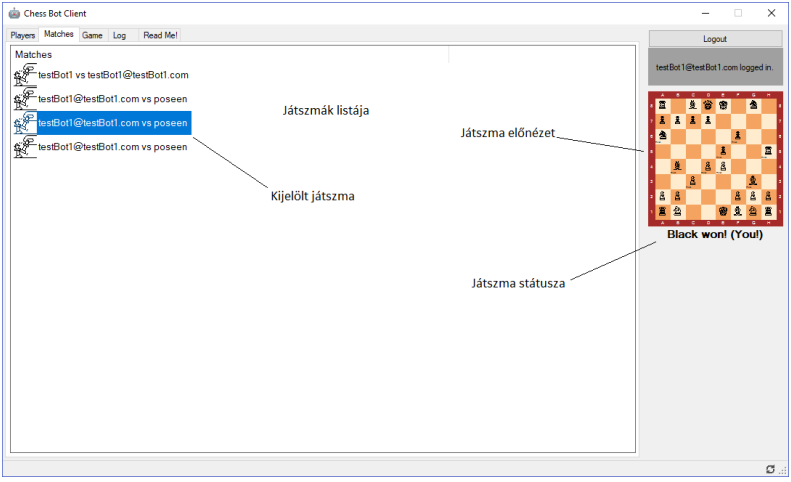
A "With Installer" verzió tartalmazza a telepítőt, amely installálja a .NET Framework 4.7.2-es verzióját, ha az nincs még telepítve. Ehhez csak ki kell tömöríteni a file-t és a Setup.exe-t elindítani és követni az utasításokat.

4.2. Használat

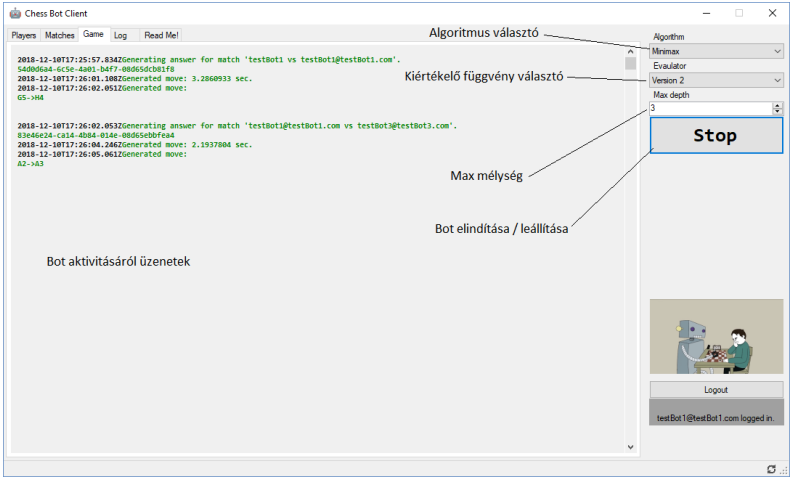
1. Játékot elindítva a fő ablakba érkezünk, ahol be tudunk jelentkezni, majd a sikeres bejelentkezés után láthatjuk a játékosok listáját:



2. A második tabra átkattintva láthatjuk a bot meccseinek listáját. Ezekre rákattintva a jobb oldalon egy kis előnézeti ablakban láthatjuk az ottani állást



3. A harmadik oldal maga a bot vezérlőpultja. Itt állítható be, hogy milyen algoritmus, mely kiértékelő függvénnyel és milyen maximum mélységgel fusson. A tér nagy részét a bot üzeneteinek fenntartott szövegmező foglalja el. Itt látható, hogy épp mit csinál a bot. 15 másodpercenként lekéri a legfrisebb állást, majd azon meccseken, ahol ő következik és léphet, kigenerál egy lépést és elküldi. A Start/Stop gombbal indíthatjuk el / állíthatjuk le a botot.



4.3. Hibaelhárítás

- A Log és a Bot fül tartalmaz minden információt, amelyre szükség lehet a hiba felderítésekor.
- Windows Event Viewer is tartalmazhaz információkat

- Néha a szerver leáll ha sokáig nem érkezett kérés. Ekkor az első kérés eltarthat egy ideig. A kliensnek le kell kezelnie ezt a problémát. Ha mégsem sikerül, akkor később újra probálkozik a frissítéssel.

- Ha végképp nem érkezik válasz, akkor érdemes megnyitni a szerver weboldalát. Ha az nem megy, akkor maga a szolgáltatás is áll.

III. rész

Fejlesztői Dokumentáció

5. fejezet

Elemzés

A megoldandó feladat egy interneten keresztül játszható sakkprogram elkészítése, melyen keresztül a játékosok egymás, illetve olyan gépi ellenfelek ("botok") ellen játszhatnak, amelyeket más játékosok készítettek. Emiatt fontos elvárás a programmal szemben, hogy emberek által könnyen használható legyen és a bot-készítő játékosok számára minden lehetséges segítség és dokumentáció a rendelkezésre álljon a feladathoz.

5.1. A program felhasználási módjai

A programcsomag felhasználási módja alapján két csoportba osztható:

- **Sakk**. Ekkor a játékosok csak játszani szeretnének egymás ellen.

- **Bot-készítés.** Amikor a játékosok egy botot szeretnének kifejleszteni és azt megmérettetni más emberi és gépi ellenfelek ellen is.

Ezen két felhasználási-módnak különböző követelményei vannak.

5.2. Felhasználói esetek

5.2.1. A sakk-szolgáltatás működésével kapcsolatos elvárások

Ebben a felhasználási helyzetben (vagy máshogy fogalmazva: a sima sakkjátékos szemszögéből) a következők a követelmények:

Kliens funkcionalitása

- **Levelezési sakk.** A program működésének hasonlítania kell a levelezési sakkhoz. Nem kell teljesen valósidejűnek lennie, de bármikor le tudja tölteni az épp aktuális állást, tudjon rá reagálni és elküldeni.
- **Felhasználói fiókok.** Minden felhansználónak legyen saját fiókja.
- **Elérhető játékosok listázása.** A játékosok szeretnék látni, hogy kiket hívhatnak ki egy partira attól függetlenül, hogy az épp kiszemelt játékos be van-e jelentkezve, vagy sem.
- **Partira kihívás.** A játékosoknak lehetőséget kell adni, hogy sakkpartit tudjanak kezdeményezni egymással.
- **Partik listázása.** A játékosoknak lehetőséget kell biztosítani, hogy láthassák az eddigi játszmáikat, azok állapotát, és ki tudják választani, hogy melyikre akarnak válaszolni.

- **Sakkjátszma.** A játékosoknak az előbbiekben kijelölt sakkjátszmához lehetőséget kell biztosítani, hogy kiválaszthassák a kívánt lépést és elküldhessék. A programnak le kell ellenőriznie, hogy a lépés megfelel-e a szabályoknak.

- **Toplista.** A játékosok szeretnék látni az eddigi játszmák alapján előálló toplistát, lehetőség szerint csoportosítva: emberi játékosok, botok, vagy a kettő együtt.

Kliens indítása	
1	Alkalmazás indítása
GIVEN	A kliens telepítve van.
WHEN	A felhasználó jelzi a játék indításának szándékát.
THEN	A kilens elindul és: • Megjelenik a kliens a hitelesítési alfelülettel.

5.1. táblázat. Kliens alapfunkciói - indítás

Kliens - Hitelesítési (al)felület	
1	Kilépés
GIVEN	Hitelesítési (al)felület aktív.
WHEN	Kilépési szándék jelzése (majd annak pozitív megerősítése).
THEN	Alkalmazás bezárása.

5.2 folytatása az előző oldalról...

Kliens - Hitelesítési (al)felület	
2	Hitelesítés
GIVEN	Hitelesítési (al)felület aktív.
WHEN	• Felhasználói név és jelszó megadása. • Belépési szándék jelzése.
THEN	Adatok elküldése hitelesítésre a szerverhez. • Ha az adatok megfelelőek: – Felhasználó beléptetése. – Hitelesítési felület elrejtése. – Lobbi megjelenítése – Hitelesítési (al)felület elrejtése – Kliens szinkronizálása a legfrissebb adatokkal. • Ha az adatok nem megfelelőek: – Felhasználói tájékoztatása problémáról.

5.2. táblázat. Kliens alapfunkciói - hitelesítési (al)felület

Kliens - Lobbi	
1	Kilépés
GIVEN	A felület aktív.
WHEN	Kilépési szándék (majd annak megerősítése).
THEN	Alkalmazás bezárása.
2	Új játék kezdeményezése
GIVEN	A lobbi-felület aktív.
WHEN	Bejelentkezett játékosok közül leendő ellenfél kiválasztva, új játék kezdésének szándéka.
THEN	Új játék kezdeményezésére vonatkozó kérelem elküldése szervernek.
3	Partik listájának megtekintése
GIVEN	A felület aktív.
WHEN	Partik megtekintésének szándéka.
THEN	Partik felület mutatása.

5.3. táblázat. Kliens alapfunkciói - lobbi

Kliens - Partik listája	
1	Kilépés
GIVEN	A felület aktív.
WHEN	Kilépési szándék (majd annak megerősítése).
THEN	Alkalmazás bezárása.

5.4 folytatása az előző oldalról...

Kliens - Partik listája	
3	Kiválasztott játék megjelenítése
GIVEN	A felület aktív, partik listájában egy játék ki van választva.
WHEN	Parti megtekintésének kezdeményezése.
THEN	Játékfelület mutatása.

5.4. táblázat. Kliens alapfunkciói - partik listája felület

Kliens - játékfelület	
1	Kilépés
GIVEN	A felület aktív.
WHEN	Kilépési szándék (majd annak megerősítése).
THEN	Alkalmazás bezárása.
2	Lépés
GIVEN	A felület aktív.
WHEN	Lépési szándék vagy feladás/döntetlen felajánlása/döntetlen elfogadása-elutasítása ellenfélnek a sakk szabályai szerint.
THEN	Kérés elküldése szervernek, majd a játékállapot frissítése.

5.5. táblázat. Kliens alapfunkciói - játékfelület

Kliens - toplista	
1	Kilépés
GIVEN	A felület aktív.
WHEN	Kilépési szándék (majd annak megerősítése).
THEN	Alkalmazás bezárása.
2	Toplista megnyitása
GIVEN	A felület aktív.
WHEN	Bármely felület aktív, toplista megnyitásának szándéka.
THEN	Kérés elküldése szervernek, majd a toplista frissítése és megmutatása.

5.6 folytatása az előző oldalról...

Kliens - toplista

5.6. táblázat. Kliens alapfunkciói - toplista

Szolgáltatás-oldal funkcionalitása

Szolgáltatás funkciói	
1	Bejelentkezés
GIVEN	A szolgáltatás elérhető.
WHEN	Beérkezik egy bejelentkezési kérés egy felhasználói névvel és jelszóval.
THEN	A szolgáltatás a bejelentkezési felhasználói név és jelszó alapján visszajelez, ha sikertelen. Siker esetén küld egy titkosított azonosítót, amellyel a kliens tudja igazolni magát. Az azonosítónak egy idő után le kell járnia. Bármilyen probléma esetén jelzi a hibát.

Table 5.7 folytatása az előző oldalról...

Szolgáltatás funkciói	
2	Titkosított azonosító meghosszabbítása
GIVEN	A szolgáltatás elérhető.
WHEN	Beérkezik egy kérés egy azonosítóval azzal a szándékkal, hogy az érvényessége meg legyen hosszabbítva.
THEN	A szerver leellenőrzi az azonosítót, és ha még érvényes, akkor egy meghosszabbított token-t ad vissza. Bármilyen probléma esetén jelzi a hibát.
3	Szolgáltatás állapotának lekérdezése ("Health-check")
GIVEN	-
WHEN	Beérkezik egy kérés a szolgáltatáshoz, amelyben a kliens érdeklődik szolgáltatás státusza felől.
THEN	Ha a szolgáltatás elérhető, akkor visszajelez. (Egyébként nem.)

Table 5.7 folytatása az előző oldalról...

Szolgáltatás funkciói	
4	Játékosok lekérdezése
GIVEN	A szolgáltatás fut, a kliens rendelkezik érvényes token-nel.
WHEN	Beérkezik egy kérés a szolgáltatáshoz, amelyben a kliens lekéri a játékosok listáját.
THEN	A szolgáltatás visszaküldi a játékosok listáját (ha érvényes volt a token). Bármilyen probléma esetén jelzi a hibát.
5	Partik lekérdezése
GIVEN	A szolgáltatás fut, a kliens rendelkezik érvényes token-nel.
WHEN	Beérkezik egy kérés a szolgáltatáshoz, amelyben a kliens lekéri az aktuális játékos partijainak listáját.
THEN	A szolgáltatás visszaküldi a partik listáját ha érvényes a token. Bármilyen probléma esetén jelzi a hibát.

Table 5.7 folytatása az előző oldalról...

Szolgáltatás funkciói	
6	Lépés
GIVEN	A szolgáltatás fut.
WHEN	Beérkezik egy kérés a szolgáltatáshoz, amelyben a kliens közli, hogy melyik partin milyen lépést szeretne kezdeményezni.
THEN	A szolgáltatás ellenőrzi, majd végrehajtja és lementi a lépést, erről visszaigazolást küld a legfrisebb állással. Bármilyen probléma esetén jelzi a hibát.
7	Kihívás
GIVEN	A szolgáltatás fut.
WHEN	Beérkezik egy kérés a szolgáltatáshoz, amelyben a kliens közli, hogy melyik felhasználót szeretné kihívni.
THEN	A szolgáltatás ellenőrzi a kérést, majd ha rendben találja, kisorsolja hogy melyik játékos lesz a fehér, majd létrehozza a játékot és visszaküldi a meccs alapinformációit a kliensnek. Bármilyen probléma esetén jelzi a hibát.

Table 5.7 folytatása az előző oldalról...

Szolgáltatás funkciói	
8	Eredmények
GIVEN	A szolgáltatás fut.
WHEN	Beérkezik egy kérés a szolgáltatáshoz, amelyben a kliens le akarja tölteni a felhasználók aktuális pontjait. (LADDER végpont, POST metódus)
THEN	A szolgáltatás a befejezett játékok alapján kiszámolja és visszaküldi a felhasználók toplistáját. Bármilyen probléma esetén jelzi a hibát.

5.7. táblázat. Szolgáltatás alapfunkciói

Bot-készítés

A bot elkészítésének segítéséhez magas fokú **dokumentáltság** és a lehető legtöbb segítség megadása szükséges a programozók számára könnyen felhasznáéható könyvtárak illetve példák formájában:

- **Dokumentáció** a szolgáltatás és a könyvtárak használatáról.
- **Könyvtárak** melyek segíthetik a programozó munkáját:
  - Játék-modellezését segítő könyvtár,
  - Mesterséges Intelligencia készítését segítő könyvtár(ak).

5.3. A sakktábla információhiánya

A sakk szabályait nem részleteznénk, ismertnek tételezzük fel, illetve az interneten könnyen hozzáférhető, mint például a Sakk.hu [1] oldalon. Néhány fontos dolog azért megjegyzendő, amely kissé megbonyolítja a sakk szabályait.

Egy volt munkahelyemen dekoráció gyanánt ki volt rakva egy sakktábla bábukkal. Ezt az emberek elkezdték használni: kinyomtattak egy papírt, egyik oldalára azt, hogy "fehér jön" a másik oldalára pedig azt, hogy "fekete jön". Ha valakinek volt kedve és ideje, odamehetett, elgondolkodhatott a lépésen, majd miután lépett, megfordította a papírt és várták, hogy valaki lépjen.

Felmerülhet a kérdés: a meccs "előtörténetének" ismerete nélkül vajon csak a tábla ismeretével kiszámítható-e az összes lehetséges lépés? Ha valaki odalép ehhez a sakktáblához, meg tudja-e mondani az összes lehetséges lépést? A rövid válasz: **nem**. A hosszú válasz: **igen, de az már egy egyszerűsített sakk**. Vannak különleges lépések, amelyekhez kell ismerni a parti addigi alakulását is (mikor lehet feladni a játékot, mikor áll be automatikusan döntetlen), illetve ha az előző lépések nem is, de valamilyen cimke a bábukon is kell vagy kellhet, például a sáncolás, vagy az *en passant* lépés felismeréséhez.

Első példaként vegyük szemügyre a sáncolást. Ekkor láthatjuk, hogy a benne résztvevő bábuknak nem elég a helyükön lenni, de az is elvárás tőlük, hogy nem is mozoghattak előtte. Ezt meg lehet állapítani az előző lépések vizsgálatával, vagy ha gyorsítani akarunk ezen, akkor egy cimke ráragaszásával arra bábura, amely elmozdult a játszma során.

Második ilyen példa a kevésbé ismert "*en passant*" ütés. Ez akkor következik be, amikor az ellenfél az egyik gyalogjával kettőt lép előre pont a mi gyalogunk mellé. Ebben az esetben a következő (és csakis a következő) körben leüthető úgy, mintha nem kettőt, csak egyet lépett volna eredetileg előre. Viszont fontos, hogy ha nem ütjük le, akkor a következő körben ezt már nem tehetjük meg. Ez esetben is használható a "cimkézés", amelyet a lépés után rárakunk, de ha a következő körben az ellenfél nem üti le, akkor levesszük róla.

Harmadik érdekes szabály az, hogy ha többször fordul elő ugyanaz az állapot, akkor a játékosok felajánlhatnak döntetlent a másiknak. Ehhez viszont ismerni kell az előző lépéseket, a táblára "simán csak ránézve" ezt a szabályt sem lehet alkalmazni. Ebben az esetben a cimkézés már nem elég, ehhez a szabályhoz már kell az előtörténet ismerete is.

6. fejezet

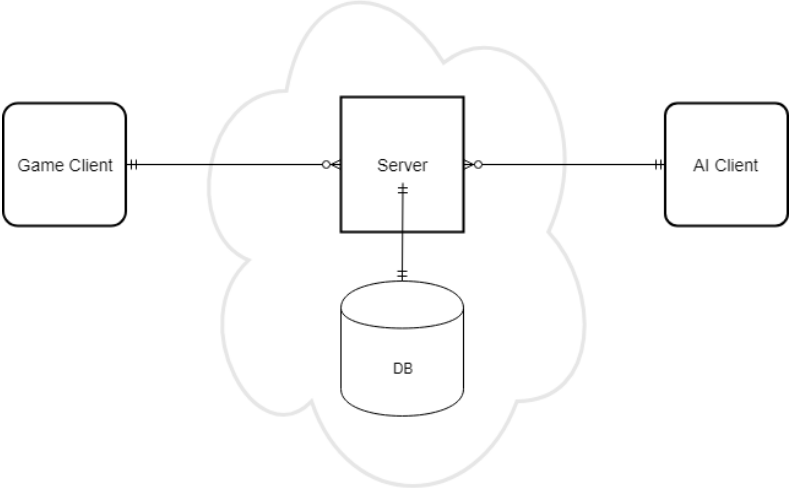
Tervezés

Az előzőekből látható, hogy két nagy részre bontható programcsomagról van szó. Az egyik a **kliens** oldal a másik a **szolgáltatást nyújtó** oldal. Erre egy sima szerver-kliens architektúra alkalmazása megfelelőnek tünik: (6.1 ábra.)

6.1. A szerver

A szerver a következő funkciókat megvalósító végpontokkal fog rendelkezni:

- **Felhasználó regisztrációja**
  Felhasználói fiókok létrehozása, kezelése. Különbséget kell tenni az emberi- és a botjátékosok között. Ez kell ahhoz, hogy később a toplista csoportosítható legyen.



6.1. ábra. Magas-szintű architektúra

- **Felhasználó nyomonkövetése.**
  A felhasználók belépését, kilépését, tokenek érvényességének vizsgálatát is a szerver végzi és csak akkor nyújtja a szolgáltatásokat, ha ezek rendben voltak.

- **Játékosok listája**
  Csak bejelentkezett felhasználók láthatják és így csak ők hívhatnak ki egy játékost.

- **Meccsek listája**
  Csak bejelentkezett felhasználók láthatják a saját meccseiket és csak akkor léphetnek, ha a szabályok szerint ez lehetséges.

- **Lépések ellenőrzése**
  Bejelentkezett felhasználó ha van kijelölt meccs, akkor a lépést a szerver ellenőrzi, hogy a sakk szabályai szerint érvényes-e. Ha igen, akkor elvégzi a parti módosítását és elmenti az új állapotot, majd a változást közvetíti a kliens felé.

- **Toplista**
  Bejelentkezés nélkül is elérhető, kilistázza a regisztrált játékosok eddigi eredményeit, feltéve ha vannak mások ellen befejezett meccsei.

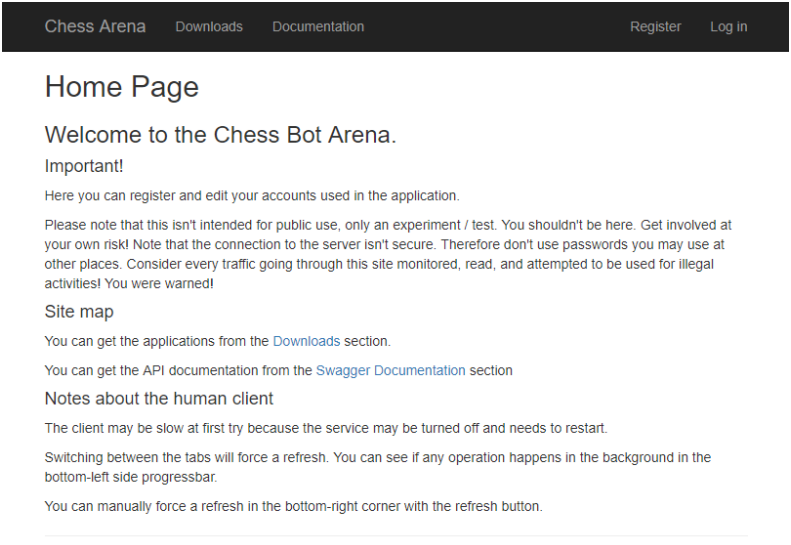
- **Státusz**
  Bejelentkezés nélkül is elérhető, akkor válaszol, ha a szerver fut.

6.1.1. A szerver felhasználói felülete

A szervernek rendelkeznie kell felhasználói felülettel, ahol a regisztráció és felhasználói adatmódosítás elérhető, illetve innen lehet elérni különböző a játékkal kapcsolatos anyagokat is, mint például a dokumentáció, letölthető kliensek.

Szerver főfelülete

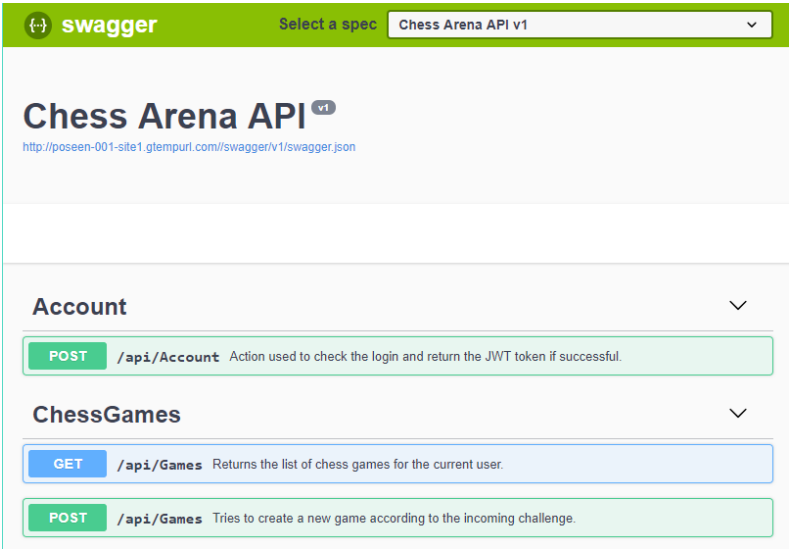
Minimalista, csak dokumentációt, letöltéseket, felhasználó regisztráció és menedzsment elérhető róla. (6.2. ábra)



6.2. ábra. Szerver fő felület

Dokumentáció

Az automatikusan generált, a fejlesztést segítő dokumentáció tervét mutatja a(z) 6.3. ábra.



6.3. ábra. Szerver - generált dokumentáció

6.2. Játék- és botkliensek

A klienseket ketté lehet bontani az alapján, hogy ki hozza a döntéseket. Ezek szerint lesznek **játékkliensek**, ahol emberi játékosok tud-

nak játszani, illetve lesznek a **botkliensek**, melyek minimális vagy semmilyen emberi beavatkozással tudnak válaszolni a bejövő kérésekre.

A két fajta kliens nem tér el túlságosan, mindkettőben kell támogatni a beléptetést, nyomonkövetést, játékinformációk szinkronizációját.

Amiben viszont eltérnek az, hogy mekkora önállóságuk van. A játékkliensnél a lépéseket az emberi felhasználó határozza meg, míg a botkliensnél egy MI algoritmus. Adódhat az ötlet, hogy a más felhasználók kihívását is automatizáljuk, de nem egyértelmű olyan megoldás implementálása, ami nem árasztja később el a játékosokat kihívásokkal, ezért ez a funkció maradhat az emberi felhasználónál.

A botkliensekből legyen lehetőség kézzel kihívni másokat, de ne tegye automatikusan. Természetesen emberi játékosok kihívhatják a botokat, azok onnantól kezdve automatikusan kell, hogy válaszoljanak a felkérésekre. Fontos kitétel a klienseknél, hogy nem szabad leállniuk egy-egy kérésnél, mindig tudniuk kell válaszolni.

A kliensek sokfélék lehetnek, nem kívánatos egy technológiához kötni őket, ezért a szerver-kliens kapcsolatot úgy kell megtervezni, hogy bármely platformon könnyen elérhető és megvalósítható legyen.

6.2.1. A kliensek működése

A különböző tabok között szabadon válthat a felhasználó. Minden tabváltáskor történik egy frissítés, amely hatására az összes oldalon levő UI elem frissül. Természetesen a frissítés a háttérben kell, hogy történjen, a UI nem "fagyhat le".

A terveken nincs kiemelve, de mindkét kliensben van egy **Log** és egy **Read me** tab. Előbbi hibakereséskor jön majd jól, ugyanis a klien-

seknek nem szabad semmilyen *exception*-t dobni, viszont mindent le kell logolnia. A *Read me* tab tartalmazza a klienshez a hasznos tudnivalókat, figyelmeztetéseket.

A következőkben lássuk a két megvalósítandó kliens felhasználói felületének terveit. Természetesen ezek csak példák, mivel a szolgáltatás nyílt API-val kell, hogy rendelkezzen, ezért a jövőbeli fejlesztőknek bármilyen más elképzelésük is lehet, például egy BASH kliens megírása is lehetséges.

A képeken látszik, hogy a két kliens nagyon hasonlít egymáshoz hiszen funkcionalitásban is elég közel állnak egymáshoz.

6.2.2. A játékkliens felhasználói felülete

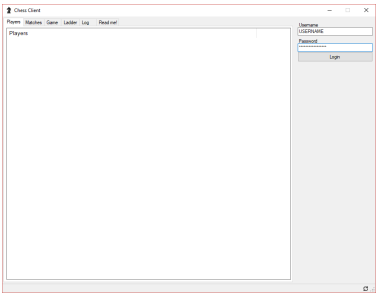
(Lásd: 6.4.ábra)

A különböző funkciók között tabokkal legyen lehetőség lépkedni, bármely felületről bármelyikre át lehessen így kattintani. A "Game" tabon lehessen a kiválasztott játékra válaszolni. Először a mozgatni kívánt bábura, majd az új pozíciójára kattintva kell, hogy elküldje a kliens a kérést a szerver felé-

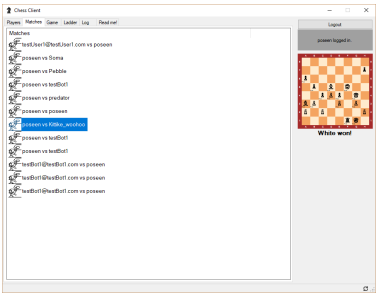
6.2.3. A botkliens felhasználóit felülete

(Lásd: 6.5)

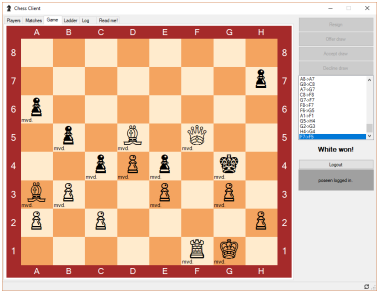
A botkliens "Game" tabján lehet vezérelni a botot. A jobb oldalon lehessen kiválasztani az algoritmust, a kiértékelő függvényt, az algoritmus maximális mélységét, illetve ott lehet elindítani és leállítani. A



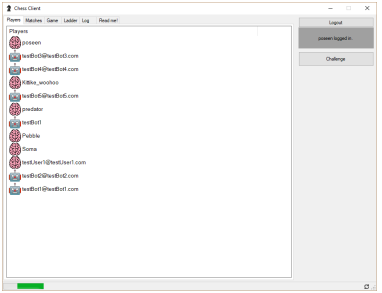
(a) Belépő képernyő



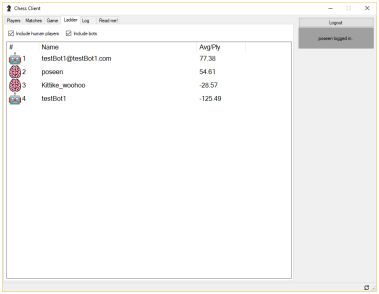
(b) Partik



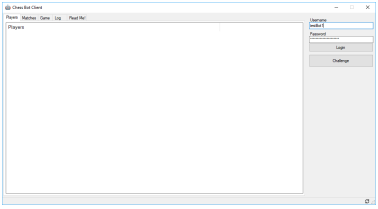
(c) Bot vezérlése



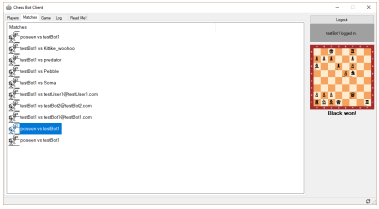
(d) Játékosok listája



(e) Toplista

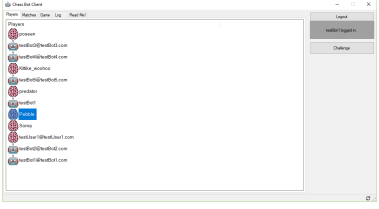


(a) Belépő képernyő



(b) Partik

(c) Bot vezérlése



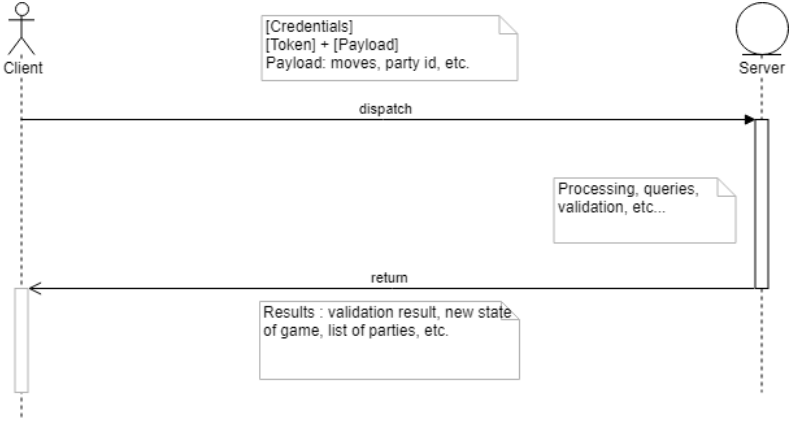
(d) Játékosok listája

6.5. ábra. Botkliens UI terv

tér nagy részét egy szövegmező foglalja el, ott látszódik, hogy épp mit csinál a bot.

6.3. Kommunikáció a szerver és a kliensek között

A kommunikációról egy sematikus rajz a 6.6 ábrán látható.



6.6. ábra. Kliens - szerver közötti kommunikáció sémája

Kérés-válasz séma: a szerver nem küld magától semmilyen információt, nem tart fent a klienssel kapcsolatot, és a szervernek nem is kell semmilyen információt tárolnia az aktuális munkamenetről. Minden kérés önmagában értelmezhetőnek kell lennie és a szervernek minden kérésnél minden információnak rendelkezésre kell állnia ahhoz,

hogy válaszolni tudjon. Megfordítva: a kliensnek minden kérést úgy kell összeállítania, hogy a szervernek elegendő információja legyen a válaszadáshoz.

6.4. Fejlesztése segítő segédkönyvtárak

6.4.1. A szolgáltatás igénybevételét segítő könyvtár

A kérések összeállítására tanácsos egy könyvtár implementálása, amire a botokat vagy klienseket író fejlesztők támaszkodhatnak. Ez azért is lehet segítség, hiszen ha a szerverből kijön egy újabb verzió új vagy módosított funkciókkal, akkor egy központi helyről beszerezve ezen könyvtárakat könnyen át tudnak állni az új verzióra.

Kéréseket összállító könyvtár

Kell egy olyan könyvtár, amely viszonylag közel áll a szerverhez. Összeállítja a kéréseket, elküldi, fogadja a válaszokat és átalakítja a megfelelő modellre, amellyel a klienst fejlesztő programozó már könnyen tud dolgozni.

Munkamenetet kezelő könyvtár

A további könnyítés érdekében a kéréseket összeállító könyvtárra építve érdemes egy munkamenetet kezelő könyvtár létrehozása is. Ez nem azonos az előzőekben emlegetett munkamenettel. E könyvtár feladata az lenne, hogy a kliens fejlesztője elől elrejtse az elosztott

rendszerek minden problémáját (szinkronizációs problémák például) és adjon egy olyan felületet, amely kívülről statikusnak néz ki, ám mindig friss információval rendelkezik az aktuális játékról és játékos "munkamenetéről".

6.5. A játék-modell és mechanika

6.5.1. Egy körökreosztott általános táblás játék

Egy körökreosztott általános táblás játék a következő elemekből áll:

- **Modell**
  A játékállás ábrázolása sokrétű lehet. Tárolni kell a játékosok listáját, az aktuális játékost és a játéknak megfelelő állást: táblán lévő bábuk vagy asztalon levő kártyák.

- **Szabályok** (avagy játékmechanika)
  A szabályok határozzák meg, hogy milyen helyzetben mik a lehetséges lépések. Az ezt kiszolgáló osztály legyen a játékmester, aki hozzányúlhat a reprezentációhoz, így biztosítva, hogy a játék mindig a szabályoknak megfelelően menjen.

Egy játékot meghatározza a **kezdőállapota** és az állapotok közötti lehetséges **állapotátmenetek**. Egy állapot akkor érvényes, ha a kezdőállapotból bármilyen úton eljuthatunk a megadott állapotba. Ezután már adódik az ötlet: egy játékmenetet reprezentálhatunk egy ***címkézett*, irányított gráffal**, ahol a csomópontok a lehetséges állapotok, az élek pedig az átmenetek.

6.5.2. A sakk-modell

A sakk modellezésénél két fontos dolgot kell figyelembe venni:

- **A sakktábla problémája**
  A sakktábla információhiánya (5.3) résznél már értekeztünk arról, hogy nem elég "ránézni" a sakktáblára, további adatok is szükségesek a lehetséges lépések meghatározásához.

- **Továbbfejleszthetőség**
  Sakktáblán sokféle játékot lehet játszani, de még akkor is sok opciónk van, ha ragaszkodunk a táblához és a sakkbábukhoz. Például ilyenkor lehet beszélni *francia sakkról* vagy a *véletlensakkról*, ami ugyanúgy működik, mint a normál sakk, azzal a különbséggel, hogy a kezdőpozíciói a bábuknak véletlenszerű.

Ebből két dolog látszik: az egyik az, hogy a játszma aktuális állapotának modelljét el kell választani a szabályoktól, hiszen ugyanazokkal a bábukkal más játékot is lehet játszani.

A másik dolog pedig az, hogy nem elég a táblán lévő bábukra hagyatkozni, be kell vezetni később jelzéseket a bábukra, illetve az eddigi lépéseket is tárolni kell. Így látszik, hogy a játék-modell két komponensből kell, hogy álljon: a **modellből** és a modell konzisztenciáját fenntartó **játékmechanikára**, amely kizárólagos módon módosíthatja a modellt a sakk szabályai alapján.

Ebből kell kialakulnia majd egy könyvtárnak, amely a sakkot modellezi és a sakkjátszmákat vezeti le.

6.6. Mesterséges intelligencia algoritmusok

6.6.1. A probléma komplexitása

Fogalmazzuk meg a problémát: szeretnénk egy olyan programot, amely legyőzi a másik játékost (vagy legalábbis nem veszít), ami ekvivalens azzal, hogy minden egyes lépésnél a legoptimálisabb lépést választja.

Tekintsük egy sakkjátszmát: egy átlagos meccs 35-40 fordulós [2] [3] [4], az átlagos lehetséges lépések száma egy adott állapotból 20 [5]. Shannon adott egy alsó-korlátot a lehetséges játékok számára, amely 10^{120}*.* [1] *Ebből látható, hogy brute-force technikával, a jelenlegi technikai háttérrel nem érdemes hozzákezdeni.*

Így a probléma elég bonyolult és ezért egzakt megoldása nem létezik egyelőre, csak közelíteni tudjuk. A továbbiakban az algoritmust általánosan fogalmazzuk meg és formalizáljuk, de a sakkon keresztüli példákon mutatjuk be.

6.6.2. A probléma, mint útkeresési algoritmus

Láthatjuk, hogy így egy játékmenet az irányított gráf egy útja lesz. A probléma pedig amit az algoritmussal meg szeretnénk oldani (vagy közelíteni) pedig az, hogy találjunk egy olyan útvonalat ebben a gráfban, amelyben a nyereségünket maximalizálni, vagy legalább az ellenfél nyereségét minimalizálni próbáljuk.

[1] Összehasonlításképp: a világegyetemben levő atomok számát 10^{80} darabra teszik, a világegyetem pedig körülbelül 4.32×10^{17} másodperce keletkezett, ami kevesebb, mint 10^{18}-on, ami még így is sok nagyságrenddel kevesebb, mint a lehetséges játékok száma.

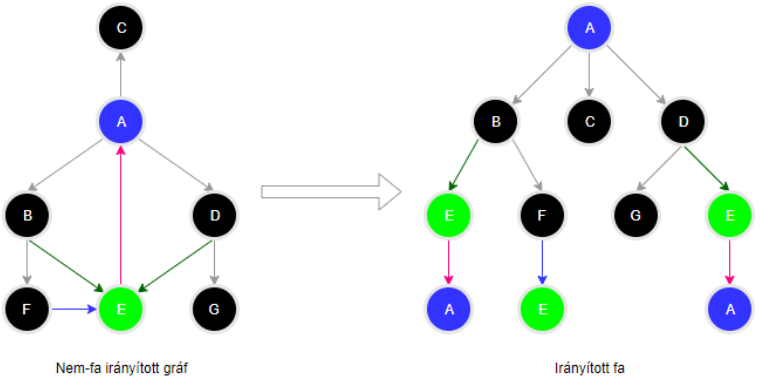
A sakk nem más, mint egy körökre osztott két személyes táblás játék és a fenti analógia alkalmazható rá: Kezdetben van egy kezdőállapot. (Felállított bábuk a táblán, világos játékos kijelölve mint kezdőjátékos). Ezek után az állapotátmeneti függvény határozza meg a lehetséges lépések halmazát, amelyből egy irányt kiválasztva és követve egy új állapotba kerülünk, és így tovább. Ez lesz a játék **játékgráfja**.

6.6.3. Egy általános gráf. Fa-e vagy sem?

Erről a gráfról semmi érdemlegeset nem állíthatunk, nem speciális, hiszen előfordulhat, hogy más utakon is eljuthatunk ugyanazon csúcsba, tehát nagy valószínűség szerint nem fa. Ezt a problémát megszüntethetjük többféle módon, például:

- **Csomópont duplázással**
  Ilyenkor az történik, hogy ha egy állapotba több úton is eljuthatunk, akkor ezeket a csúcsokat többszörözzük.
  (Lásd.: 6.7 ábra.)

- **Ügyes reprezentációval**
  Ilyenkor azt tesszük, hogy elgondolkodunk: attól, hogy ugyanabba a csomópontba jutunk több úton, az nem jelenti-e esetleg azt, hogy a más úton elért csúcs már inkább egy másik csúcs. Sakkos példánál maradva: ezt úgy érhetjük el, hogy egy állapot reprezentációjába bevesszük az odavezető utakat. Például előállhat egy hasonló állapot, ha mi is és az ellenfél is előrelép majd vissza ugyanazzal a bábuval.



6.7. ábra. Általános irányított gráfból irányított fa konstruálása

6.6.4. Nem is kell feltétlenül fa

Tegyük fel, hogy van egy algoritmusunk, amely kiszámítja a saját és az ellenfél legmegfelelőbb lépéseit a játék végéig. Így megkapjuk az utat ami a mi győzelmünkhöz vagy legalább az éppen legjobb elérhető eredményhez vezet. Ez nem feltétlen célravezető, hiszen mi történik akkor, ha az ellenfél végül nem azt a lépést lépi valamilyen okból? (Lehet, hogy az ő algoritmusa jobb vagy rosszabb, mint a mienk. Lehet, hogy más kiértékelőfüggvényt használ, stb.) Ekkor azt látjuk, hogy a sok munka, amellyel felépítettük az utat, kárba vész. Ezért nem érdemes az egész utat felépíteni, elég csak annyit elvárni tőle, hogy adjon egy javaslatot a következő lépésre. Természetesen a "hogyan" kérdésre a válasz lehet elég bonyolult is (minimax-, alfa-béta-, genetikus- és

tanuló-algoritmusok trükkös heurisztikákkal), de a vége ugyanaz: egy lehetőleg legoptimálisabb lépés kiszámítása. Mivel nem fogjuk bejárni az egész fát csak egy adott mélységig, ezért a lépések száma véges lesz, így már nem probléma ha a gráf nem fa (azaz van benne kör), így a probléma nem is biztos, hogy létezik.

6.6.5. Mesterséges Intelligencia Algoritmusok könyvtára

Érdemes definiálni egy mesterséges intelligencia általános algoritmusokat megvalósító könyvtárat is, ezzel nyújtva további segítséget a jövőbeli fejlesztőknek. Ez azért is hasznos, mert az algoritmusok nem változnak, a nagy különbség a játékállapotot kiértékelő implementációban van, így elég, ha a programozó arra koncentrál.

6.7. Összefoglalás

Ezek alapján összegezzük a leendő programcsomag szerkezetét és különböző elemeinek feladatát:

- **Játék-modell** (könyvtár)
  - Megvalósítja a játéktér modelljét
  - Ezen játéktérhez megvalósítja a sakk szabályait
- **Algoritmusok** (könyvtár)
  - Definiál egy interface-t

- Tartalmazza néhány algoritmus általános megvalósítását:
  - Minimax, átlagoló minimax
  - Alfa-béta
  - Egyéb egyszerűbb példák teszteléshez (random, mohó)
- Könnyen használható és kiterjeszthető más alkalmazásokban.

- **Architektúra**
  - **Szerver**
    - Felhasználó regisztrációja / hitelesítése / nyomonkövetése.
    - Ellenőrzi a kliensektől jövő lépések érvényességét.
    - Közvetíti a kliensek felé a partik alakulását.
  - **Kliens (humán vagy MI kliens)**
    - Felhasználó beléptetése, nyomonkövetése.
    - Közvetíti a felhasználó felé a partik alakulását.
    - A játékos döntéseit közvetíti a szerver felé.
    - Példakliensek implementálása:
      - Játékkliens
      - Botkliens
  - **Kliens-közeli könyvtárak**
    - **Szerverközeli kliens könyvtár**
      - Könnyen kezelhető interfészt ad

- Összeállítja a kéréseket
- Visszajövő válaszokat a kliensfejlesztő számára könnyen feldolgozható modellé alakítja

* **Játék munkamenet-könyvtár**
  - Egy egyszerű interface-t ad
  - A szerverközeli klienskönyvtárra épül
  - Szinkronizációs feladata van
  - "Business object"-ként funkcionál

7. fejezet

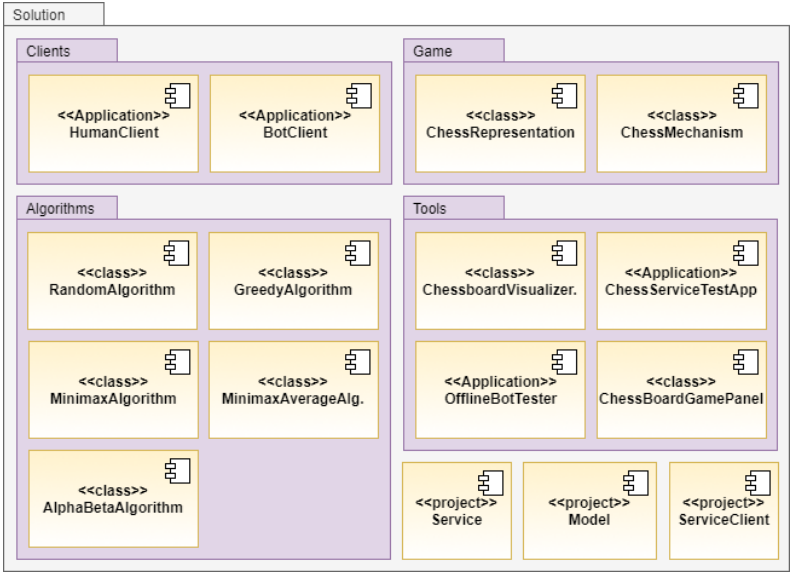
Megvalósítás

A projekt fontosabb részeinek felépítését lásd: 7.1. ábra.

7.1. A sakk megvalósítása (`Game` projekt)

7.1.1. Egy általános körökreosztott táblás játék modellje

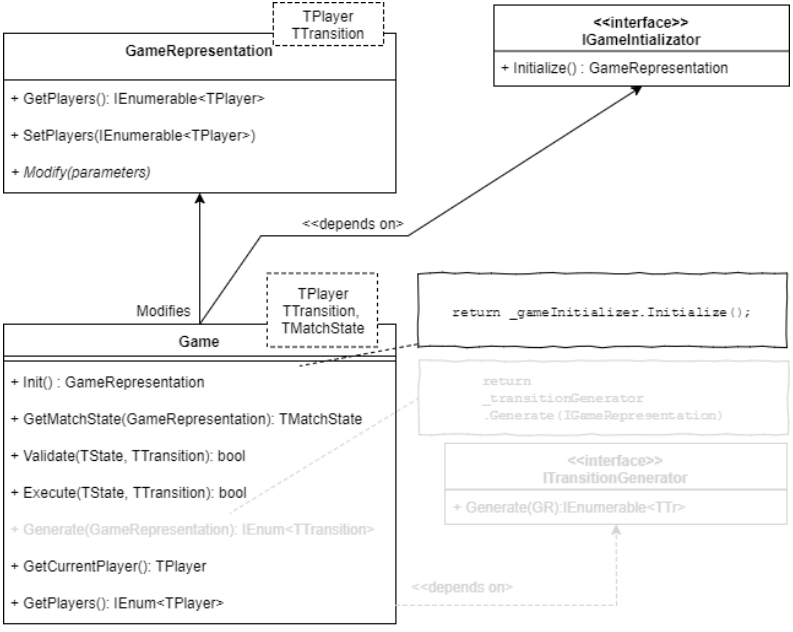
Kezdjük megint egy általános megközelítéssel: Legyen `GameModel` osztály, ami reprezentálja egy játék egy állapotát. Csak tárolja a tábla vagy asztal aktuális állapotát és lehetőséget biztosít a módosítására. Nem szabad semmilyen játék alapelveit beleépíteni, hiszen például egy sakktáblán is lehet többfajta sakkot játszani, amelyek szabályai különbözőek.



7.1. ábra. AlphaBetaAlgorithm osztály

A `GameMechanism` osztály a "játékmester", ő az, aki a szabályok szerint módosítja a reprezentációt. Rajta keresztül kell validálni és végrehajtatni a lépéseket, illetve tőle lehet lekérdezni az aktuális meccs állását illetve egyéb fontosabb tulajdonságait.

Egy a fentieknek megfelelő magasszintű terv látható a 7.2. ábrán.



7.2. ábra. Egy általános táblás, körökre osztott játék absztrakt reprezentációja

7.1.2. A modell (`ChessRepresentation` osztály)

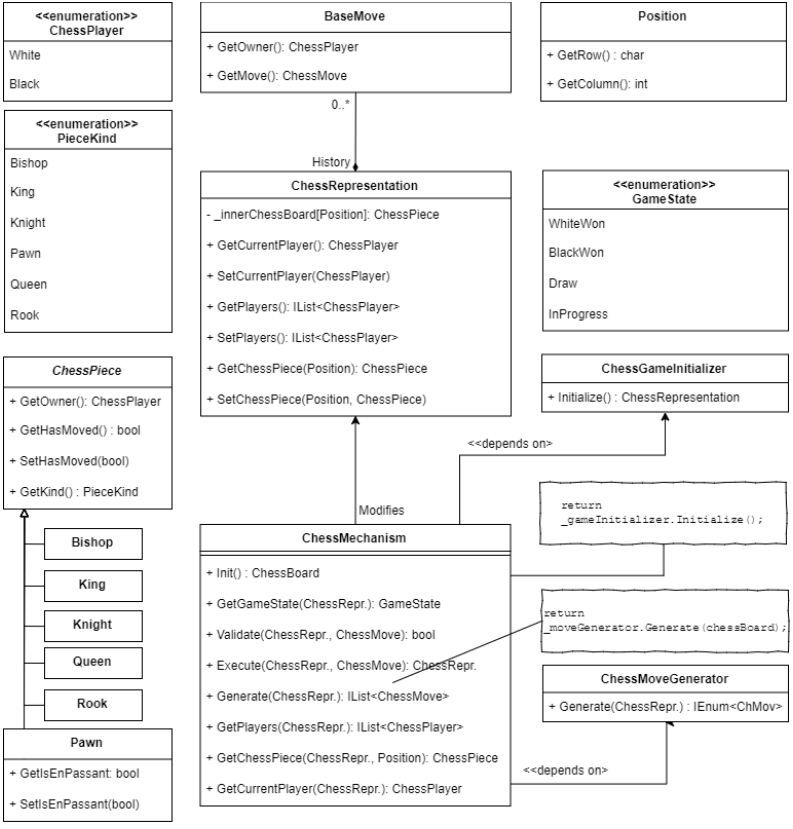
Az előző fejezetekben végiggondoltuk, hogy mi a minimális információmennyiség, amely ahhoz kell, hogy egy sakkjátszma egy állapotát leírjuk úgy, hogy abból egyértelműen kiszámítható legyen a lehetséges lépések halmaza. Ezek alapján sejthető a lehetséges reprezentáció: egy sakk-állás leírásához a következő információk kellenek:

- Tábla állapota
- Következő játékos
- Bábuk jelölése (elmozdult? en passant?)
- Eddig megtett lépések listája

Illetve az előzőekben megtárgyaltuk, hogy hogyan is lehet érdemes modellezni és implementálni a sakkot, ha a későbbiekben továbbfejlesztési terveink vannak vele: A sakk is két fő részből áll: modell és játékmechanika. Ezeket a `ChessBoard` és a `ChessGame` osztályok valósítják meg. (Lásd: 7.3. ábra.)

A `Generate()` függvény azért van kiszürkítve az ábrán, mert egy játék lehetséges lépései nem feltétlen diszkrét értékek egy véges halmaza. Szigorúan véve csak azt kell megkövetelni a `GameMechanism` osztálytól, hogy el tudja dönteni egy bejövő lépésről, hogy az érvényes-e. A lépések kigenerálása inkább csak a mesterséges intelligencia algoritmusának "jöhetne jól".

A sakktáblát reprezentáló `ChessBoard` osztály az előzőekben részletezett alapelveknek felel meg. Legfontosabb művelete a



7.3. ábra. Sakkot reprezentáló osztálydiagram

`Move(from, to)` függvény, amely elvégzi a módosításokat a táblán, ám a `History`-ba nem írja be, az majd a "játékmester" feladata lesz. Továbbá a `Move()` függvény visszatérési értéke a "leütött" bábu, vagy `null` ha üres volt.

A sakktáblát egy egydimenziós tömb tárolja, amin a tájékozódás kissé nehézkés, ezért a sakkhoz jobban illő `Position` tipuson keresztül lesz ez megkönnyítve, amely az *algebrai lejegyzést* használja.

A későbbi fejlesztések segítéséhez az osztály tartalmaz konvertereket (array index és pozíció között oda-vissza) és egyenlőségvizsgálatot is, hogy írhassuk: `p1 == p2` vagy `p1.Equals(p2)`.

Ezeken kívül - kissé lazábban kapcsolódva a szigorúan vett feladathoz - implementálva lett a `Position` osztályhoz néhány kiterjesztés is, amely a sokszor használt pozícióval kapcsolatos műveleteket implementálja. Ilyen például az "északi irányba lépés" vagy a "lólépés", illetve a diagonális, horizontális és vertikális lépések listájának kigenerálása. Ezt valósítja meg a `PositionExtensions` osztály (Lásd 7.1. kódrészlet).

A kiterjesztési függvények segítségével szép láncolt kifejezéseket lehet szerkeszteni, példa erre a `KnightNorthEast` függvény, ahol könnyen leolvasható miből is áll egy észak-keleti irányba lépő huszár: "kettőt északra, egyet keletre".

7.1.3. A sakk-mechanizmus (`ChessMechanism` osztály)

A sakk implementációja sokkal bonyolultabb és felvet néhány érdekes problémát. Minden függvény első bemenő paramétere

7.1. Kódrészlet. Pozíciók segédfüggvényes - PositionExtensions osztály

```
public static class PositionExtensions
{
  public static Position ParsePostal(this string postalNotation) { /*
       ... */ }

  public static Position North(this Position position, int number =
      1) { /* ... */ }
  public static Position South(this Position position, int number =
      1) { /* ... */ }
  public static Position West(this Position position, int number = 1)
       { /* ... */ }
  public static Position East(this Position position, int number = 1)
       { /* ... */ }
  public static Position NorthEast(this Position position, int number
       = 1) { /* ... */ }

  /* ... */

  public static Position KnightNorthEast(this Position position)
  {
    var newPosition = position.North(2).East(1);
    return newPosition;
  }

  public static Position KnightNorthWest(this Position position) { /*
       ... */ }

  /* ... */

  public static IEnumerable<Position> KnightMoves(this Position
      position)
  {
    var possibleMoves = new []
    {
      position.KnightSouthWest(),
      position.KnightSouthEast(),
      position.KnightWestSouth(),
      position.KnightWestNorth(),
      position.KnightEastSouth(),
      position.KnightEastNorth(),
      position.KnightNorthWest(),
      position.KnightNorthEast()
    }.Where(x => x != null);

    return possibleMoves;
  }
```

egy modell. Ebből állapítja meg, hogy mik a lehetséges lépések (`GenerateMoves`), egy megadott lépés helyes-e az adott helyzetben (`ValidateMove`) és alkalmazza a megadott lépést ha helyes (`ApplyMove`). Ezeket külön-külön megvizsgáljuk:

GenerateMoves()

A `GenerateMoves` függvény az egész osztály legfontosabb függvénye. Erre építkezik több másik függvény is. Mivel minden állapotban a lehetséges lépések halmaza véges és diszkrét, ezért implementálható és felhasználható más függvényekben is. *(Ha nem lenne implementálható, akkor csak a `ValidateMove` és az `ApplyMove` függvényeket lehetne implementálni.)*

A lépések kigenerálásának két lépése van: a naív kigenerálás és az illegális lépések kiszűrése.

1. **Lehetséges lépések naív kigenerálása**

   - **Normál lépések kigenerálása**
     A legtöbb báb ahová léphet, ott üthet is. Így ezen lépések kigenerálása egyszerű, csupán annyi megszorítással például egy futónál, hogy habár bármennyit léphet diagonálisan, bábut nem ugorhat át, így a saját bábu esetében előtte meg kell állnia, vagy ellenfél bábuja esetében azt leütve a helyére kell lépnie és tovább nem mehet.
   - **A gyalog lépései**
     A gyalog lépéseinek kigenerálása az egyik bonyolult függvény. Eleve ketté kell osztani a lépéseit, ugyanis neki van-

nak olyan lépései, amelyek nem ütnek. A gyalog csak "srégen előre" tud ütni, viszont lépni (ha van helye) előre is tud. Ezért ezt a két esetet szét kell választani, amely majd jól jön később, amikor a sakkba lépést vizsgáljuk.

- **Sáncolás**
  A sáncolásnak több előfeltétele is van: a benne résztvevő bástya és a király nem mozdulhatott még el a parti alatt, a király nem lehet támadva ("sakkban"), illetve a király által lépdelt mezők sem lehetnek támadva. Ez így elég sok feltétel, ráadásul rendhagyó is. Az előzőekben részletezett lépéseknél könnyen használható a későbbi szűrés, de mivel itt szigorúan véve nem csak egy lépés, hanem egy lépéssorozat történik és minden lépésben meg kell vizsgálni, hogy támadva van-e a sáncolás útvonala, ezért ezt "helyben" kell elvégezni. Ehhez ki kell számolni azokat a mezőket, amelyek támadás alatt vannak. (Ez egy érdekes rekurzív problémához vezet.)

2. **Szabályokkal ellenkező lépések kiszűrése**
   Meg kell vizsgálni, hogy az adott lépés "felfedi-e" a királyt egy támadó bábunak. Ez esetben a lépés nem lehetséges. Ezt úgy tudjuk megvizsgálni, hogy kiszámoljuk a saját lépésünk utáni támadott mezőket és megnézzük, hogy ezek között szerepel-e a király pozíciója. Ugyanez az eljárás sakk esetén is, hiszen ez esetben pedig csak olyan lépések jöhetnek számításba, amelyek megszüntetik a sakkot. Ha ilyen lépés nem létezik, akkor megtörtént a sakk-matt.

A szűrésnél a másik dolog amire figyelni kell, hogy csak a támadó lépéseket kell figyelembe venni. Ez a gyalog lépéseinél jöhet szóba, hiszen egy gyalog léphet előre, ám ott nem üthet, ezért a király például léphet az ellenfél gyalogja elé. Ezért is kell szétválasztani a gyalog esetében a normál- és ütőlépések kigenerálását.

Egy érdekes rekurziós probléma

A sáncolás kiszámításánál futhatunk bele egy trükkös rekurzióba. A feltételek ellenőrzésének része, hogy megállapítsuk: a megfelelő mezők és bábuk támadva vannak-e. Ehhez meg kell állapítani az éppen az ellenfél által veszélyeztetett mezőket. Ehhez ki kell generálnunk az aktuális helyzettől függő ellenfél lépéseit, amelynek viszont része a sáncolás is. Így egy végtelen rekurzióhoz jutunk. Ezt úgy lehet megszüntetni, hogy a veszélyeztetett mezők kiszámításánál nem vesszük figyelembe az ellenfél sáncolását, hiszen az semmilyen esetben nem lehet hatással a mi sáncolásunkra.

ValidateMove()

Ennél a játéknál szerencsénk van: ha a `GenerateMoves()` függvény implementálható (azaz a következő lehetséges lépések halmaza véges és diszkrét elemekből áll) akkor nincs más dolgunk, mint megállapítani: a beérkezett lépés benne van-e a kigenerált lehetséges lépések között.

GetGameState()

A `GetGameState` függvény hivatott kiszámolni, hogy mi a játék aktuális állapota? Ha van még lehetséges normál lépés, akkor jelzi, hogy lehet még lépni, ha bármilyen módon vége van (valaki feladta, mattot adott vagy patt-helyzet alakult ki) akkor azt jelzi. Ehhez az utolsó lépést vizsgálva és a `GenerateMoves` függvényt meghívva számolja ki a kimenetet:

- Ha az utolsó lépés egy döntetlen elfogadása $\rightarrow$ döntetlen.
- Ha az utolsó lépés egy feladás $\rightarrow$ ellenfél játékosa nyert.
- Ha van még lépés $\rightarrow$ A játék folyamatban van.
- Ha nincs már lépés, akkor...
  - Ha az aktuális játékos sakkban áll $\rightarrow$ MATT
  - Ha az aktuális játékos nem áll sakkban $\rightarrow$ PATT (döntetlen)

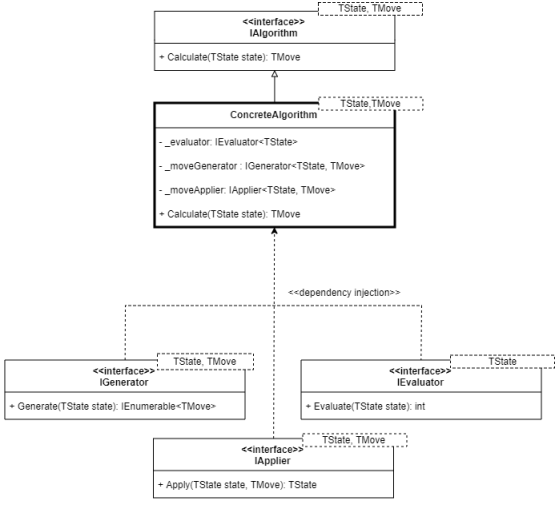
ApplyMove()

Az `ApplyMove` függvény végrehajtja a megfelelő módosításokat a táblán és a lépést "lejegyzi". Kimenete a módosítás utáni reprezentáció. Ha a lépés nem megengedett `null`-lal tér vissza.

7.2. MI algoritmusok (`Algorithms` projekt)

7.2.1. Algoritmusok absztakt interface-e

Az algoritmusok interfésze nem bonyolult: egyetlen függvénye van, a `TMove Calculate(TState state)`, amely paraméterként kap egy aktuális állapotot (játék-modellt) és visszatér egy lépéssel. (Lásd: 7.4. ábra.)



7.4. ábra. Absztrakt algoritmus interface és függőségei

A konkrét megvalósításhoz kellhet egy kis segítség. Az algoritmusnak sok lehetséges lépésből kell kiválasztania a legmegfelelőbbet. Ehhez három dolog kell:

- **Lépés-generátor**
  Segédkezik a lehetséges lépések kigenerálásában.

- **Kiértékelő**
  A kiértékelő osztály mondja meg egy állapotról, hogy az "mennyire jó".

- **Alkalmazó**
  Ez az osztály hivatott módosítani a reprezentációt, valójában egy "wrapper" osztály a valódi "játékmester osztályhoz".

A generátor interface

Az algoritmus nem tud semmit a sakkról és nem is kell neki. Azt, hogy milyen lépések lehetségesek, kívülről megadott függőségek fogják megadni. Így a megvalósítás lehet a játékhoz készült generátor osztály köré rakott wrapper vagy proxy-osztály, de lehet több felelőssége is. Egyetlen függvénnyel rendelkezik, egy megadott állapotra alkalmazva visszaadja a lehetséges lépések (egy) halmazát. Lehet egyszerű és elég bonyolult is. A következő lépéseket tartalmazhatja:

Első lépésben diszkretizálhat: például ha a játék lehetséges lépéseinek száma egy adott állapotban végtelen, ekkor véges számú lépéseket gyárt és azt adja vissza. Ha - mint például a sakkban - minden állapotból csak véges sok lépés lehetséges, akkor azokat adja vissza.

Második lépésben szűrhet. Például egy heurisztika alapján az első lépésben megkapott listát leszűrheti az alapján, hogy mit van értelme megvizsgálni és mit nem.

Harmadik lépésben sorrendiséget határozhat meg. Ez például jól jöhet az alfa-béta algoritmusnál, ahol a bejövő lehetséges lépések sorrendjének az eredményre bár nincs hatása, de a futási időre már annál inkább lehet. Ha a lépések pont "rossz sorrendben" jönnek be, akkor az alfa-béte lefutása nem lesz jobb a minimax lefutásánál. Ha viszont "jó sorrendben" jönnek a lépések vizsgálatra, akkor drasztikus sebességnövekedést lehet elérni az alfa-béta algoritmussal. (Egy kedvező és egy kedvezőtlen esetet láthatunk a 7.9 ábrán.)

Az alkalmazó interface

Egyetlen függvénnyel rendelkezik, egy megadott állapotra alkalmazza a megadott lépést és visszaadja az új állapotot. Legtöbbször egy wrapper a játékot kezelő osztály körül.

A kiértékelő interface

Kell egy kiértékelő-osztály is, amely eldönti az újonnan előállt állapotról annak értékét az aktuális játékos szemszögéből. Erre jelen esetben az `integer` megfelelő választás lesz, maximum és minimum értékét (`int.MaxValue` és `int.MinValue`) extremális elemként használva. A minimum érték jelentse a $-\infty$-t, a maximum érték pedig jelentse a $+\infty$-t.[1] Egyetlen függvénnyel rendelkezik, a megadott állapotra ad egy pontértéket az aktuális játékos szemszögéből.

[1] Megjegyzendő, hogy a visszaadott érték nem feltétlen kell, hogy egy sima természetes vagy valós szám legyen, de fontos, hogy az `TResult Evaulate(TState state)` függvénynek a `TResult` típusérték-halmazának legalább részben-rendezett halmaznak kell lennie.

7.2.2. Véletlen-algoritmus (`RandomAlgorithm` osztály)

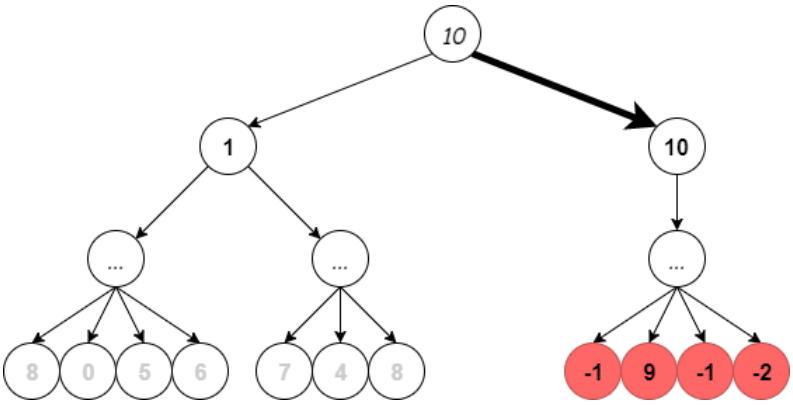
A lehetséges lépések közül véletlenszerűen választ egyet. Működése nagyon egyszerű: nem végez kiértékelést, csak a lehetséges lépések közül választ egyet. Néha szerencséje van. Használata akkor javasolt, amikor egy gyors algoritmusra van szükségünk egy-egy új koncepció teszteléséhez, vagy a szerver teherbírásának kipróbálásakor sok adat generálásánál.

7.2.3. A mohó algoritmus (`GreedyAlgorithm` osztály)

A mohó algoritmusban habár már van logika, az nem feltétlen célravezető. A lehetséges lépések közül azt választja, amelyik a legnagyobb veszteséget okozza a következő körben az ellenfél játékosának. A mohó algoritmus nagy hibája, hogy meg fog találni egy lokális maximumot, de nem feltétlen találja meg a globális maximumot. Ezt egyébként onnan is lehet látni, hogy meglepően agresszíven játszik. Ha lehetséges, üt. Ha lehetséges, sakkot ad, és nem érdekli, hogy utána esetleg veszít egy tisztet. (Példát lásd 7.5. ábra.)

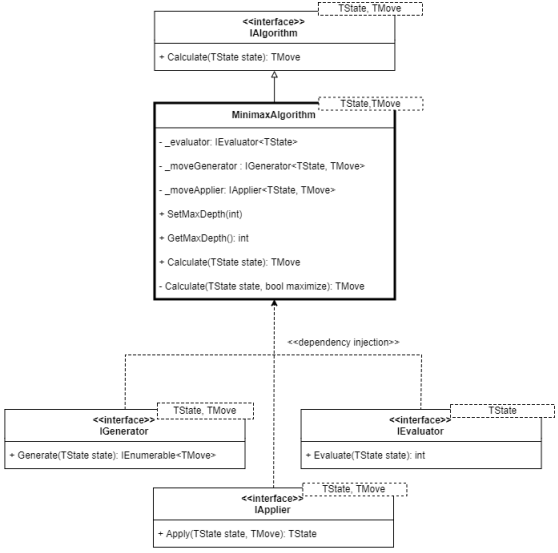
7.2.4. A minimax-algoritmus(`MinimaxAlgorithm` osztály)

Most nézzük a Minimax algoritmust, mely absztrakt UML diagramja 7.6. ábrán látható.



7.5. ábra. Mohó algoritmus működése és ahogy tévútra vezet a lokális maximum.

A minimax algoritmus lényege, hogy a maximum-mélységben kiértékeli az állásokat, és ezeknek a maximumát illetve minimumát viszi tovább attól függően, hogy melyik szinten van. Első szint a MAX szint, ahol a következő szinten levő csomópontok közül abba az irányba indulunk el, amelyik a legkisebb, ezzel próbálva minimalizálni az ellenfél nyereségét. Következő szint a MIN szint lesz, ahonnan a legnagyobb értékű csomópont felé indulunk el, hogy maximalizáljuk a nyereséget. (Példát lásd 7.7. ábra.) [6]



7.6. ábra. MinimaxAlgorithm osztály

Hatékonyság

Tegyük fel, hogy minden csomópontból x a lehetséges lépések száma és d a maximum mélység amit vizsgálunk. Ekkor az algoritmus műveletigénye: $\mathcal{O}(x^d)$. [7]

7.2.5. Az átlagoló eljárás (`MinimaxAverageAlgorithm` osztály)

A minimax algoritmus egy általánosítása. *Avagy a minimax algoritmus az átlagoló eljárás egy speciális esete.* A probléma a statikus kiértékelő-függvényekkel az, hogy lehetetlen olyat megadni, amely megbízhatóan értékeli ki az állásokat. Lehet, hogy egy nagyon jónak ígérkező állásról kiderül, hogy csak egy "csapda" volt és onnan nehezebb vagy lehetetlen megnyerni a meccset. Erre a problémára adhat egy jobb megoldást az átlagoló eljárás. Az ötlet mögötte az, hogy míg az eredeti Minimax algoritmus mindig az aktuális csúcsból elérhető csúcsok minimumát vagy maximumát veszi, addig az átlagoló eljárás a kiterjesztett csúcsok értékeinek MAX szinten az m legnagyobb átlagát, MIN szintjén az n legkisebb átlagát veszi. (Példát lásd 7.7. ábra.) [6]

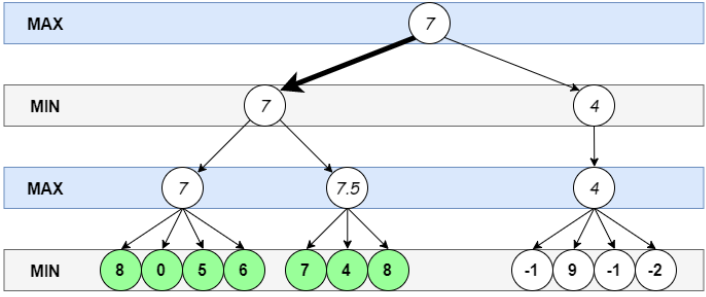
Tehát ha úgy vesszük, akkor a minimax algoritmus nem más, mint az átlagoló-minimax $m = 1$ és $n = 1$ esetén.

Hatékonyság.

Ugyanaz, mint a minimax algoritmus esetében. Tegyük fel, hogy minden csomópontból x a lehetséges lépések száma és d a maxi-

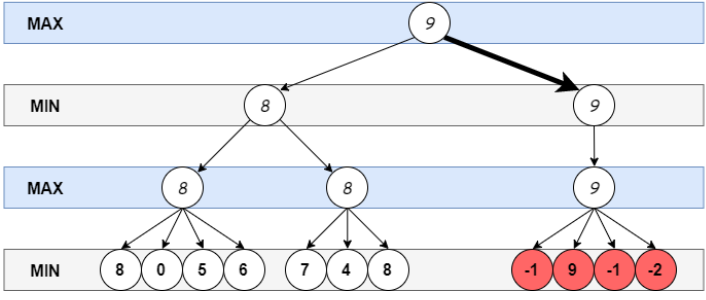
Minimax-átlagoló példa

m=2, n=2



Minimax példa

Minimax-átlagoló == Minimax, ha m=1, n=1

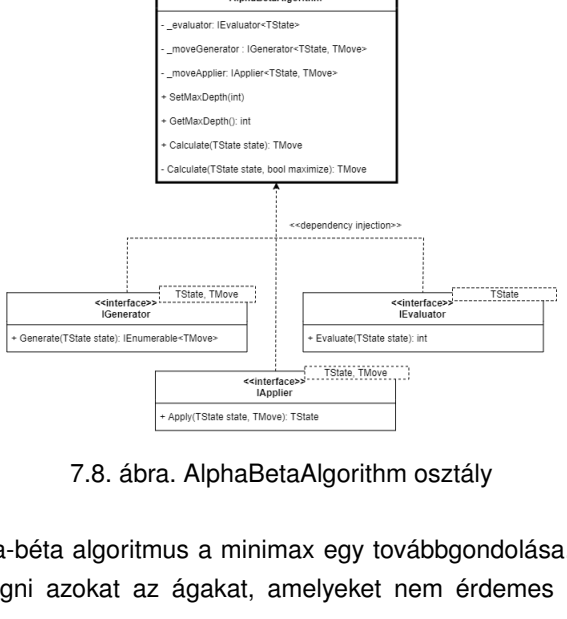


7.7. ábra. Minimax és Átlagoló eljárás összehasonlítása [6]

mum mélység amit vizsgálunk. Ekkor az algoritmus műveletigénye: $\mathcal{O}(x^d)$. [7]

7.2.6. Az alfa-béta algoritmus (`AlphaBetaAlgorithm` osztály)

Az alfa-béta algoritmus felépítése nagyon hasonló a minimax-ok felépítéséhez. Absztrakt UML diagramja 7.8. ábrán látható.



7.8. ábra. AlphaBetaAlgorithm osztály

Az alfa-béta algoritmus a minimax egy továbbgondolása. Megpróbálja levágni azokat az ágakat, amelyeket nem érdemes már vizs-

gálni. Ez jól jön a számítási-igény csökkentésénél. Az esetek túlnyomó többségében ugyanazt az eredményt kell adnia, mint a minimax-algoritmusnak.

Hatékonyság.

Tegyük fel, hogy minden csomópontból x a lehetséges lépések száma és d a maximum mélység amit vizsgálunk.Ekkor a legrosszabb esetben, amikor nem tudunk vágni és ezért minden terminális csúcsot ki kell értékelni, az algoritmus műveletigénye megegyezik a minimax algoritmuséval: $\mathcal{O}(x^d)$. Azonban legjobb esetben, ha minden egyes esetben a számítás szemszögéből legjobb csúccsal kezdjük a kiértékelést, ez a műveletigény lecsökken: $\mathcal{O}(\sqrt{x^d})$. [7]

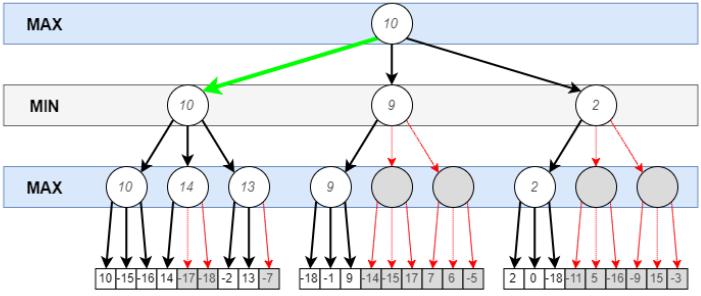
A tapasztalat azt mutatja, hogy az alfa-béta algoritmus legtöbbször ugyanazt az eredményt adja rögzített mélység esetén, mint a minimax algoritmus. Viszont sokkal hatékonyabb, így ha a csúcsok a "megfelelő sorrendben érkeznek kiértékelésre", akkor a keresőfa mélysége akár meg is duplázható.
(Összehasonlító ábrát lásd: 7.9. ábra.)

7.2.7. Egy érdekes végtelen ciklus

Vigyázni kell, mert az azonos kezdeti feltételek kombinálva azonos algoritmussal, max-mélységgel és kiértékelő függvénnyel egy végtelen játszmába vezethet. Előfordulhat, hogy a botok elkezdenek oda-vissza lépni és onnantól csak ezt a négy lépést ismételgetni. Ez a probléma a determenisztikus algoritmusoknál jöhet elő, mint például egy minimax-

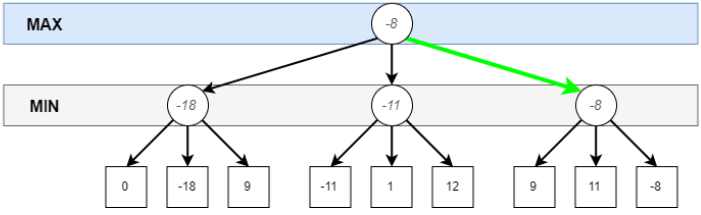
Alfa-Béta vágás

Egy kedvező eset



Alfa-Béta vágás

Egy kedvezőtlen eset



7.9. ábra. Alfa-béta algoritmus számítási-igény változása a bejövő adatok sorrendjének függvényében.

nál, ahol ugyanarra a bemenetre ugyanazzal a kiértékelő függvénnyel és mélységgel ugyanaz az eredmény fog kijönni a végén. Probléma lehet még például ha az egyik oldalon egy minimax, a másikon egy alfa-béta algoritmus fut, hiszen az esetek nagy többségében ezek az algoritmusok ugyanazt az eredményt adják, az alfa-béta csak a minimax egy gyorsítása.

Tehát adott 3 paraméter: algoritmus, mélység, kiértékelő függvény. Ekkor hogy elkerüljük, a végtelen-csapdát a következőket tehetjük:

- **Különböző kiértékelő függvény**
  Azonos algoritmus és mélység-beállítás esetén a két botnak különböző kiértékelő függvénye legyen

- **Különböző algoritmusok**
  Azonos mélység-beállítás és kiértékelő függvény esetén a két botnak ne ekvivalensen determinisztikus algoritmusa legyen. (Például ne legyen minimax és alfa-béta párosítás, de például megy minimax-minimax átlagoló már működhet.)

- **Maximális mélység-beállítás legyen különböző**
  Azonos algoritmus és kiértékelő függvény esetén a maximális mélység-beállítás legyen különböző

- **Nem-determinisztikussá tétel**
  Ha mindhárom azonos, akkor pedig lehet egy kis véletlenszerűséget becsempészni akár az algoritmusba, akár a kiértékelő-függvénybe.

7.3. A kiszolgáló (`Service` projekt)

A bevezetőben már volt róla szó: mivel kérés-válasz párokra épül maga az eredeti játék is, ezért egy RESTful szolgáltatás építése tünik a legmegfelelőbbnek amely JSON-ban kommunikál. Ez azért is lesz jó, mivel így bármilyen platformon viszonylag könnyű lesz hozzá klienst írni illetve mivel iparági-sztenderd, így könnyű hozzá könyvtárakat letölteni és használni. Például a *Visual Studio*-ban pár kattintással létrehozhatunk egy teljesértékű szolgáltatást egy példa végponttal, illetve felhasználó-regisztrációval, beléptetéssel.

Framework szempontjából a **.NET Core 2.1**-et használjuk, mert lehetővé teszi a cross-platform fejlesztést (lásd. .NET Standard) és így a könyvtárak más platformokon is használhatók lesznek.

Adatbázisnak **MSSQL**-t fogunk hanszálni, mert egy már fentiekben definiált *Microsoft*-os fejlesztői környezetbe jól integrálódik és így könnyű vele dolgozni.

A kliens-programokhoz **WinForms** keretrendszert fogunk használni, mivel talán az egyik legegyszerűbb (habár nem a legjobb) rendszer.

A szolgáltatás weboldalát Razor engine fogja generálni, az API referencia-dokumentációjának automatikus elkészítéséhez pedig a **Swagger**-t fogjuk használni, amely segítségével még le is generálhatunk egy kliens-vázat a szolgáltatáshoz szinte bármely jelentősebb nyelvre, így könnyítve meg a jövőbeli API-felhasználó fejlesztők dolgát, akik például nem .NET-ben hanem JAVA-ban vagy akár go-ban szeretnének klienst vagy botot fejleszteni a projekthez.

RESTful API

A szerver **RESTful** szolgáltatást nyújt a kliensek felé, amely többek között azt jelenti, hogy a szerver nem tárol információt a klienssel fennálló kapcsolatról, a kliens elegendő információt kell hogy küldjön a szervernek ahhoz, hogy az válaszolni tudjon.

A kommunikáció alapjaként a HTTP protokollt fogjuk használni, amely rendelkezik a szükséges státuszjelzésekkel, így egy teljesértékű API-t tudunk nyújtani extra-munka nélkül.

További információért lásd.: [8].

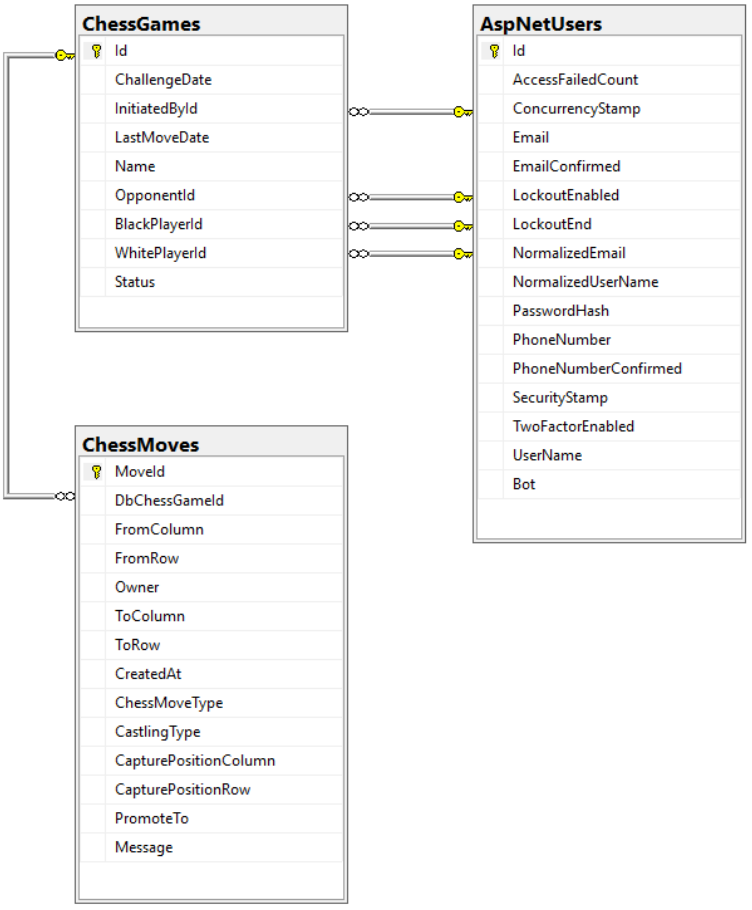
7.3.1. Adatbázis

Az adatbázis felhasználók regisztrációját és beléptetését kezelő része adott, hiszen része a Microsoft .NET Core MVC alapbeállításainak, így csak a hozzá kapcsolódó és magával a sakkal foglalkozó részt kell megtervezni. Ez EF Core használata miatt elég egyszerű, hiszen C♯ osztályokból épít sémát, amelyet ha módosítás történik, karban is tud tartani. A 7.10 ábrán láthatók a táblák és kapcsolataik.

A `ChessGame` tábla tartalmazza a partikat. Ehhez kapcsolódik a `ChessMoves` tábla, amely magukat a lépéseket tartalmazza.

Az `AspNetUsers` tábla tartalmazza a regisztrált felhasználókat, illetve ide vannak bekötve a `ChessGame` tábla olyan mezői is, mint `Opponent`, `BlackPlayer`, `WhitePlayer` és az `InitiatedBy`.

A kötések `FOREIGN KEY`-ek így biztosítják az adatbázis konzisztenciáját. (Nem lehet törölni egy entitást addig, amíg van rá hivatkozás.)



7.10. ábra. Adatbázis diagram

7.3.2. Kommunikáció a szerver és a kliensek között

A kommunikáció **JSON** üzenetek formájában, HTTP verb-ekkel és HTTP status kódokkal történik, ahogy a 6.6 ábrán látható. Sikeres bejelentkezés során a szerver küld egy `JWT Token`-t, amelyt minden kérésnél a kliensnek mellékelnie kell a kérés fejlécében.

A JWT-token (egy) problémája

A JWT-token tartalmazza az érvényességi idejét. Alapvetően ez befolyásolja, hogy meddig használható fel, ezért amíg le nem jár, addig a szerver el is fogja fogadni. Ezért néha meg kell újítani, illetve nem visszavonható és módosítható alapból. Tehát például abban az esetben, ha egy token a következő fél órában még érvényes de a hozzá tartozó felhasználót ki akarjuk tiltani a szerverről, akkor egy "visszavonási lista" nélkül (vagy valami hasonló megoldással) ezt nem tudjuk megtenni.

Továbbá általában tartalmazza a felhasználó jogait is. Ha például egy felhasználót promótálunk, akkor a régi token még mindig a régi jogait fogja tartalmazni, ezért ha frissíteni akarjuk, akkor ki kell lépnie és vissza, hogy az újonnan kapott token már a friss információt tartalmazza a felhasználóról.

További információ a JWT token-ekről elérhető a JWT.io oldalon.

7.3.3. Dokumentáció

A később részletezett open-API dokumentációját a swagger nevű generátorral oldjuk meg, amely felépítését mutatja a 6.3. ábra.

Az API dokumentációt a *swagger* [9] generálja a kód, az annotációk és '*summary*' tag-ek alapján.

A swagger generál egy az API-t leíró sémafile-t is, amelyet felhasználva a swagger oldalán generáltathatunk egy kliens a legtöbb nyelvre.

7.2. Kódrészlet. Health-check példakód 'summary' tag-ekkel.

```
/// <summary>
/// The health controller. Only used to return the version
     information.
/// (Can only respond when the service is up and running.)
/// </summary>
[Route("api/Health")]
public class HealthController : Controller
{
  /// <summary>
  /// Health check. Returns the version number of the service.
  /// </summary>
  /// <returns>The version information of the service.</returns>
  [ProducesResponseType(typeof(string), (int)HttpStatusCode.OK)]
  [HttpGet]
  public async Task<IActionResult> Get()
  {
    var version = Microsoft.Extensions
            .PlatformAbstractions
            .PlatformServices
            .Default
            .Application
            .ApplicationVersion;

    return Ok(version);
  }
}
```

A fenti kódban a `Get()` függvény *summary* szövege lesz az API végpont dokumentációjában, a `ProducesResponseType` attribute class-szal pedig jelezhetjük, hogy milyen HTTP státuszjelzés mellett

milyen modell jön vissza a szervertől. Például: ’200 OK’ mellett visszajön egy várt komplex objektum, de mondjuk ’500 ERROR’ mellett visszajön egy komplex hibaobjektum, nem csak maga a hibakód és az esetleges hibaüzenet.

7.4. Kliens-könyvtárak (`ServiceClient` projekt)

A `ServiceClient` projektben találhatók azok az osztályok, amelyek segíthetik a fejlesztőt egy kliens írásában. Két főbb osztály van implementálva, az egyik közelebb van az API-hoz, abban segít, hogy könnyű legyen összeállítani a kéréseket. Ez a `ChessServiceClient` osztály. A másik "business object"-ként segít: ezen keresztül lehet vizsgálni az aktuális állapotát a játékos partijainak. Ezt valósítja meg a `ServiceConnection` osztály.

7.5. Két példakliens

7.5.1. Sakktábla megjelenítése (`ChessBoardGamePanel`)

A `ChessBoardGamePanel`-en keresztül lehet látni a játék aktuális állapotát és ezen keresztül lehet kiválasztani a következő lépést. `OnValidMoveSelected` eseménye akkor aktiválódik, amikor egy lépés ki lett jelölve.

Az elemzés részben megemlítettük, hogy jelöléseket ragaszthatunk a bábukra, hogy gyorsítsuk egy-egy algoritmusunkat. A megjelenítéskor látható "mvd" és "ep" szavak pont ezek a jelzések. Az "mvd" jelentése "moved", azaz már elmozdult az eredeti helyéről, illetve az "ep" azt jelenti, hogy a gyalog "en passant üthető".

7.5.2. A játékkliens (`HumanClient` projekt)

A `ServiceConnection` osztályon keresztül kommunikál a szerverrel. Bizonyos időközönként frissíti a háttérben az állapotát és ha történt változás, akkor megmutatja azt.

7.5.3. A botkliens (`BotClient` projekt)

A `ServiceConnection` osztályon keresztül kommunikál a szerverrel. Bekapcsolás után folyamatosan frissíti az állapotát és ha talál olyan játékot, amelyre válaszolhat, akkor a megadott paraméterekkel kigenerálja a választ.

7.6. Egyéb segédkönyvtárak (`Tools` projekt)

E könyvtár tartalmazza azokat a komponenseket és alkalmazásokat, amelyek segítségével ki lehetett próbálni egy-egy új funkcionalitást a még készülő szoftverben.

7.6.1. A `ChessboardVisualizer` osztály

Tartalmazza a kódot a Visual Studio visualizer-hez. Sokat segít debug-olás közben, hiszen a segítségével lehet látni a sakktáblát és lehet navigálni a lépések között is.

7.6.2. Algoritmus és API tesztelő alkalmazások

Az `AlgorithmOfflineTester` és a `ChessServiceTestApp` alkalmazások a fejlesztés elején segítettek abban, hogy kipróbálhassuk: mennyire működőképes egy-egy elgondolás. További információk a függelékben 10.

8. fejezet

Tesztelés

A tesztelésnek különböző szintjei vannak:

- **Unit tesztek**
  Egyszerű és gyors tesztek, általában egy kisebb egységet: metódust, függvényt tesztel. Ebben a szoftverben főleg a központi elemeket, a reprezentációt és a mechanizmust teszteli hiszen központi részei a dolgozatnak. Ha ezek nem működnek megfelelően, akkor az egész programcsomag nem működik megfelelően.

- **Integrációs tesztek**
  A rendszer különböző komponenseinek együttműködését teszteli. Technológiai szemszögből néz a problémára: nem feladata a helyesség megállapítása, a komponensek közötti együttműködést vizsgálja. Könnyen elkészíthető, hiszen a .NET Core ren-

delkezésünkre bocsájt egy `TestServer` nevű osztályt, amely memóriában futva szimulálja az alrendszereket.

- **API tesztek**
  Ettől a szinttől felfelé már a helyesség is fontos. A kiajánlott API tesztelésekor a publikus API felületet hívjuk meg előre definiált adatokkal és figyeljük a működést.

- **GUI tesztek**
  Ezek lehetnek automatikus vagy manuális tesztek. Többnyire utóbbit szokták választani, mert egy gyorsan változó környezetben nagyon nehéz a teszteket módosítani és validálni.

- **Teljesítmény-teszt**
  Automatikus teszt, ahol a rendszert hatalmas bejövő adatnak teszik ki és figyelik a viselkedését, különösképpen arra, hogy leáll-e, elfogy-e a CPU vagy a memória, esetleg megjelennek-e deadlock-ok, illetve okoz-e anomáliákat a többi rendszerben.

- **Rendszerteszt**
  Manuális teszt, ahol felállítanak egy tesztkörnyezetet és megpróbálják szimulálni a mindennapos használatot.

Felülről lefelé haladva nő ezen tesztek ára és komplexitása. Jelen dolgozatban időhiány miatt csak unit és manuális tesztek vannak, de az automatikusan válaszoló botokkal a rendszert így is meg lehetett tesztelni (integrációs-, teljesítmény- és API teszt), amely fel is fedett jó néhány hibát.

8.1. Kódrészlet. Sakk-matt detektálás teszt

```
[Fact]
public void GameStateCheck_CheckMateTest_BlackWins()
{
  var expected = GameState.BlackWon;

  // Fool's mate
  var game = new ChessRepresentationInitializer().Create();
  var mechanism = new ChessMechanism();
  var moves = new List<BaseMove>(4)
  {
    new ChessMove(ChessPlayer.White, Positions.G2, Positions.G4),
    new ChessMove(ChessPlayer.Black, Positions.E7, Positions.E5),
    new ChessMove(ChessPlayer.White, Positions.F2, Positions.F3),
    new ChessMove(ChessPlayer.Black, Positions.D8, Positions.H4)
  };

  foreach (var move in moves)
  {
    game = mechanism.ApplyMove(game, move);
  }

  var actual = mechanism.GetGameState(game);

  Assert.Equal(expected, actual);
}
```

8.1. Reprezentáció

A tesztelés unit-teszteken keresztül történik, amelyben előre megadott szituációkat mutatva a kódnak megvizsgáljuk, hogy a lehetséges lépéseket kapjuk-e meg vagy sem, illetve manuálisan végignézve állásokat megállapítjuk, hogy jók-e az állások. Egy sakk-matt tesztelő példát láthatunk a 8.1 kódrészletben.

8.1.1. Kiterjesztések

A sakktáblán való navigáláshoz készült sok segédfüggvény is, hogy könnyebben lehessen navigálni illetve egyik formátumból a másikba konvertálni, illetve az olyan segédfüggvényekhez is, mint az "északra kettőt", "lólépés" satöbbi.

8.1.2. Külön vizsgálandó esetek

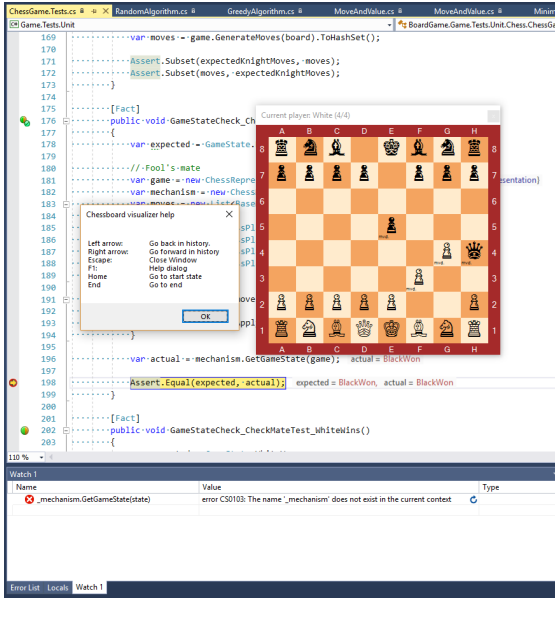
Főleg a különleges lépéseket kellett jól letesztelni. Például:

- **Sáncolás** Nagyon szigorú szabályok vannak arra, mikor lehet sáncolni. Ezeket az eseteket végig kellett venni.
- **Sakk-matt** A tesztek között van két nevezetes sakk-matt szituáció: a suszter-matt és a bolond-matt. Egyiket a világos, a másikat a sötét játékos kaphatja.
- **Speciális lépések** Például Feladás, döntetlen felajánlása, stb.
- **En passant** En passant lépés tesztelése.
- **Promóció**Amikor a gyalog átér, akkor átváltozik egy tiszté.

8.1.3. Visual Studio Visualizer

A unit-tesztek írása közben sok hiba jött elő és sokat közülük nehéz volt kijavítani, mert nagyon nehéz elképzelni, hogy mi is történik éppen. Ezért ki lett fejlesztve egy **Visualizer** a `ChessRepresentation`

osztályhoz, amely beépül a Visual Studio-ba és ott ha debug közben rávisszük a kurzort az objektumra, akkor megnézhetjük vizuálisan is a sakktáblát, illetve oda-vissza lépkedhetünk a történetében is. Ez a projekt része, habár fordítás után be kell másolni a `C:\Users\FELHASZNÁLÓNÉV\Documents\Visual Studio 2017\Visualizers\` könyvtárba és egy VS újraindítás után már használható. Ez látszik a 8.1 ábrán.



8.1. ábra. Visualizer működés közben, a súgó ablakával nyitva.

8.1.4. Manuális tesztelés játék közben

Sok szituáció volt kipróbálva manuális játék közben is, ahol két klienst futtatva mindkettőbe bejelentkezve különböző szituációkat előidézve teszteltem a játékot. Ezt end-to-end tesztnek is vehetjük, hiszen a játék-infrastruktúra minden elem le lett ezáltal tesztelve, illetve speciális helyzeteket előidézve a játéklogika is:

- Sáncolás tesztelése,
- En passant tesztelése,
- Promóció minden bábutípusra, külön figyelve arra, hogy az átváltozás okoz-e sakkot,
- Patt helyzet előállítása, a kód felismeri-e

8.2. Algoritmusok

A tesztelés unit-teszteken keresztül történik, amelyben előre megadott adatokra azt kapjuk-e, amit elvárunk. Példa unit teszt látható a 8.2. kódrészletben.

Ezekkel a tesztekkel az a probléma, hogy nagyon nehéz őket összeállítani: fel kell építeni egy reprezentációt, majd abban megírni egy kiértékelő osztályt, egy generátor osztályt. Rengeteg a hibalehetőség. Ezért inkább a manuális tesztek felé fordultam: Először a botokat játszattam egymással és figyeltem, hogy lefagynak-e, rosszat lépnek-e, mekkora a memória és CPU igényük. Végül elkezdtem ellenük játszani.

8.2. Kódrészlet. Mohó-algoritmus teszt

```
[Fact]
public void Test1()
{
  var evaluator = new TestCase1.Evaluator();
  var generator = new TestCase1.Generator();
  var applier = new TestCase1.Applier();
  var algorithm = new GreedyAlgorithm<TestCase1.State, TestCase1.Move
       >(evaluator, generator, applier);

  var initState = new TestCase1.State(0, 0);

  var move1 = algorithm.Calculate(initState);
  var state2 = applier.Apply(initState, move1);
  var move2 = algorithm.Calculate(state2);

  Assert.Equal('a', move1.Label);
  Assert.Equal('d', move2.Label);
}
```

8.2.1. Szolgáltatás

Az infrastruktúra tesztelését manuális játékkal illetve a botok közötti játékkal lehetett kvázi-automatikusan tesztelni. Erre volt jó például a véletlen-algoritmus és a mohó-algoritmus, hiszen nagyon gyorsak, így elég sok lépést meg tudnak tenni viszonylag rövid idő alatt.

8.2.2. API

POSTMAN használatával történt az API hívások tesztelése. Mivel HTTP-n keresztül, JSON-ben megy a kommunikáció, ezért könnyű volt a különböző eseteket összállítani és letesztelni minden esetet. (Bad request, autentikációs probléma, stb.) Később a botok egymás elleni

játéka illetve emberi játékosok egymás és botok elleni játéka által is tesztelve volt az API és fel is derített néhány hibát.

8.2.3. Szerver UI

A szerver oldalon minimális UI van, ezért nincs szükség túl sok manuális tesztelésre. Amit meg kellett nézni:

- Mi történik, ha nem megfelelő a megadott jelszó?
- Mi történik, ha regisztrációkor az egyik megadott adat nem megfelelő?
- Elérhetők-e a letölthető file-ok és a dokumentáció?

8.3. Játékkliens

Elejétől a végéig manuális teszteléssel.

- Mi történik rossz jelszó megadásakor?
- Mi történik, ha nem megy a szerver? Jól kezeli a kliens?
- Mi történik, ha akkor próbálok lépni, amikor nem az én köröm van? Mi van, ha nem megengedett lépést akarok beküldeni?
- Mi történik normál esetben?
- Mi történik, ha a szerver leterhelt és lelassul?

8.4. Botkliens

Az infrastruktúra tesztelésekor önmagát is tesztelte. Ugyanazon aspektusokból kellett tesztelni, mint a játékklienst.

- Mi történik rossz jelszó megadásakor?
- Mi történik, ha nem megy a szerver? Jól kezeli a kliens?
- Mi történik, ha akkor próbálok lépni, amikor nem az én köröm van? Mi van, ha nem megengedett lépést akarok beküldeni?
- Mi történik normál esetben?
- Mi történik, ha a szerver leterhelt és lelassul?

8.5. Megfigyelések

8.5.1. Algoritmusok sebességkülönbsége

Az alfa-béta algoritmus egy jelentős gyorsítása a minimax algoritmusnak. Megfigyelések alapján körülbelül fele annyi idő kellett neki, hogy kiszámolja ugyanazt a mélységű eredményt.

8.5.2. Hamis biztonságérzet

Nem megfelelő unit-teszt lefedettség mellett ha a tesztelendő algoritmusaink bonyolultak, akkor az adhat egy hamis-biztonságérzetet,

hogy "ad valamilyen eredményt". Ez nagyon veszélyes, hiszen ha nincsenek letesztelve a szélsőséges esetek, akkor előfordulhat, hogy az algoritmus nemhogy rossz, hanem semmilyen eredményt sem fog adni.

8.5.3. Erőforráshiány

Egy idő után lehetetlenné vált a tesztelés a lokális gépen, mert a sok bot kliens, a service, az adatbázis szerver illetve a Visual Studio együttesen már annyira leterhelte a gépet, hogy lehetetlenné vált minden a munka a gépen. Ezért lett a project publikálva egy külső szerverre.

IV. rész

Függelék

9. fejezet

Néhány érdekesség a sakkról

9.1. Játékok osztályozása

A **„kétszemélyes, teljes információjú, diszkrét, véges és determinisztikus, zéró-összegű, játékok”** definíciója egy kis magyarázatot érdemel. A játékokat a következő szempontok alapján lehet osztályozni:

- **Játékosok száma**
  Ez lehet egy, kettő, vagy n.
- **Elérhető információ mennyisége**
  A játékosok számára mennyi információ érhető el? Minden in-

formáció birtokában vannak-e (pédlául sakk), vagy nem (például egy kártyajáték, ahol nem látják egymás lapjait.)

- **Körökre-osztott-e vagy sem?**
  Lépések diszkrét sorozatából áll-e a játék vagy folyamatos.

- **Lehetséges lépések száma**
  Diszkrét esetben a különböző állapotokban lehetséges lépések száma véges vagy végtelen? A játék véges sok lépésben véget-ér?

- **Véletlen-faktor**
  A véletlennek van-e szerepe a játékban? Ha nincs, akkor a játékot *determinisztikusnak* hívjuk.

- **Játékosok nyereségeinek összege**
  Ha a játékosok nyereségeinek összege 0 (amennyit nyer az egyik, annyit veszít a másik), akkor *zérusösszegű* a játék.

Ez alapján a sakk egy:

- **Kétszemélyes**

- **Tejes információjú**
  Ránézve a sakktáblára és a játéklépések jegyzékére minden játékos tudja a pontos állást.

- **Diszkrét**
  Mert diszkrét lépések sorozata egy játék.

- **Véges**
  Mivel véges sok lépésben végetér a játék és az állapotokból lehetséges lépések száma is véges.

- **Determinisztikus**
  Mivel a szerencsének nincs szerepe a játékban.

- **Zéró-összegű**
  A szerzett pontok száma mindig egy: győztes játékos 1 pontot kap, a vesztes 0-t, döntetlen esetén 0.5-0.5 pontot kapnak a felek.

játék.

9.2. A sakk története

A játék neve a perzsa „shāh" szóból ered, amely uralkodót jelent.[1] Története a legendák világába nyúlik vissza. Egy ismert mese szerint egy brahmin találta ki a sakkot. Jutalmul a rádzsától első hallásra jelentéktelennek tűnő fizetséget kért, mindössze annyi búzaszemet, amennyi a sakktábla kockáira a következő szabály szerint képletesen rátehető: az első kockára egy, a másodikra kettő, a harmadikra négy, vagyis az előzőnek mindig duplája. Hamar kiderült, hogy ennyi búza nem terem a Földön, sőt az emberiség egész történelme alatt nem termelt ennyit.

[1] A "sakk-matt" kifejezés pedig a perzsa „Shāh Māt" kifejezésből származik, amelynek jelentése: "a királyt lerohanták" vagy "védtelen", "legyőzött". [10] [11]

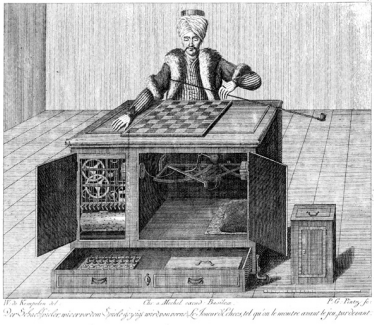
Az első ismert változata valószínűleg Indiában alakult ki. A maihoz képest egyszerűbb volt, de ugyanakkora táblán és ugyanazokkal a bábukkal játszották, a lépések viszont különböztek.

Indiából először Perzsiába jutott el. Amikor az arabok elfoglalták Perzsiát, a muszlim világ átvette és ennek következményeként el is terjesztette a játékot Dél-Európában. A sakk mai formáját körülbelül a XV.-ik században érte el.

Keleti terjedése már kissé bizonytalan: először Kínába majd Japánba jutott el és közben sokkal nagyobb változásokon ment keresztül. *Egy kevésbé elfogadott elmélet szerint eredetileg Kínából származik és terjedt el.* [12] [13]

9.3. Egy "gondolkodó gép": a Török

9.1. ábra. Kempelen Farkas - A Török



A Török Kempelen Farkas által 1769-ben alkotott legendás sakkozógép volt, melyet azért épített, hogy elkápráztassa vele Mária Teréziát. A gép sakkpartikat játszott és nyert. Végül végigtúrázta Európát és Észak-Amerikát míg 1854-ben Philadelphiában egy tűzben meg nem semmisült.

A gép titka nem egy zseniális korát meghaladó "mechanikus mesterséges intelligencia"

volt, hanem egy benne elrejtőző emberi kezelő. Ez a gyanú többször is felvetődött, de Kempelen a játszmák előtt felnyitva a gépet mutatta be, hogy nincs senki a gépben. Valójában az eléggé apró termetű kezelő el tudott bújni, amíg elöl illetve hátul is kinyitották a szerkezetet, hogy belenézhessenek. A gépet végül 1857-ben leplezte le nyíltan a The Chess Monthly című amerikai sakklap. [14] [15] [16]

9.4. Kaissza

Az 1950-es évektől kezdődően egyre több, egyre erősebb sakkprogramot írtak. 1947-ben készült el a világ első sakk-komputer programja, amelyet Alan Turing írt. 1974-ben volt az első sakkprogram-világbajnokság, ahol a győztes a szovjet Kaissza(Каисса) nevű program volt, amely 1974 és 1977 között uralta a nemzetközi számítógépes sakkot. [17]

1990-ben elkészíteték PC-s verzióját, amely a londoni számítógépes olimpián negyedik helyezett lett. [18]

9.5. Deep Blue

A Deep Blue az IBM által körülbelül 20 millió dolláros költséggel kifejlesztett számítógépe, amely

9.2. ábra. Deep Blue

sakkjátékban 1997-ben egy szabályszerű hatjátszmás páros mérkőzésen New Yorkban legyőzte Garri Kaszparovot, az emberi sakkozás akkori világbajnokát.

Első meccs-győzelmét 1996 február 10.-én érte el a világbajnok ellen, de Kasparov három partit nyert és kettőt hozott ki döntetlenre, így 4-2-re győzött a számítógép ellen.

Ezután Deep Blue-t továbbfejlesztették és ekkor már 3,5–2,5-re verte Kasparov-ot 1997 májusában, és ezzel az első számítógépes rendszer lett, amely meg tudta verni az aktuális sakk-világbajnokot a sakkbajnokságokra jellemző időmegszorítás mellett. Kasparov csalást gyanított, és követelt egy visszavágót, ám ebbe az IBM nem ment bele. [19] [20]

10. fejezet

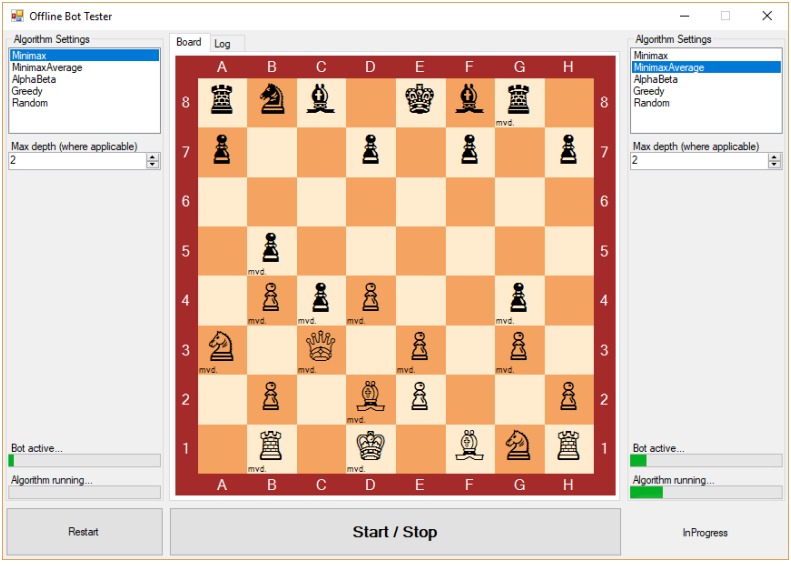
Egyéb segédprogramok

10.1. Az algoritmus-teszter

Az algoritmusok egymás elleni tesztjében lehet segítség egy egyszerű alkalmazás, amelyben az algoritmusokat különböző beállításokkal egymásnak lehet engedni a szerver kihagyásával. Az alkalmazás nagyon egyszerű és nem "prodution ready". A solution-ben `OfflineAlgorithmTester` néven található, 10.1. ábrán látható a főképernyője.

10.2. Szolgáltatást tesztelő alkalmazás

Még a fejlesztés elején készült, hogy ki lehessen próbálni az API-hívásokat, illetve lehessen elemezni a sakkjátszmákat. Ez sem



10.1. ábra. Algoritmusok tesztelése egymás ellen

"production-ready", a kódminősége is mutatja: inkább csak amolyan "proof-of-concept". A solution-ben `ChessServiceTestApp` néven található, 10.2. ábrán látható a főképernyője.

10.2. ábra. A szolgáltatást tesztelő alkalmazás

10.3. "Proof of Concept"

Ezen programok jellemzője, hogy csak belső használatra készültek a programcsomag fejlesztésének elején, hogy minél előbb látszódjon, ha egy koncepció működik, vagy nem működik. Ezt hívjuk "**proof of**

concept"-nek, avagy röviden "**POC**"-nek. Ezekkel szemben nem elvárás a jó kódminőség vagy a felhasználóbarátság, bár nem is tiltott természetesen. Célja az, hogy be lehessen mutatni: amit kitaláltunk az egy működőképes koncepció. Általában új termékek vagy ötletek bevezetésénél szoktak ilyet írni nagyon rövid idő alatt, így minimalizálva a veszteséget abban az esetben, ha a POC azt mutatja, hogy mégsem működhet a koncepció vagy a megrendelőnek nem tetszik.

Murphy törvénye

Ami elromolhat, az el is romlik. Nincs száz százalékos bizonyosság sosem. Hiába a rengeteg manuális vagy automatikus teszt, a program valamely része mindig akkor és ott romlik el, amikor és ahol a legkevésbé számítunk rá, főleg ha egy olyan bonyolult rendszerről van szó, mint egy mesterséges intelligencia algoritmusáról, amely működése kívülálló szemszögéből nem egyértelmű. Megfigyelt jelenség, hogy a betonbiztosan működő szoftver is általában a megrendelő számára tartott bemutató előtt összeomlik. Ezért mindig legyen B-tervünk. És egy C.

10.4. Fordított CAPTCHA

Felmerült egy érdekes probléma a kliensprogramok tervezése során. Minden ilyen rendszer fejlesztésekor gondolni kell a rosszindulatú felhasználásra, jelen esetben például arra, mi van akkor és hogy szűrhető ki, ha valaki csalni akar. Többféle módja van a csalásnak, azonban van két eset, amely külön érdekes:

Rossz sakkozó, de erős programozó

Az egyik az, ha valaki nem jó sakkozó, de tud jó programot írni. Ekkor felmerülhet az ötlet, hogy amikor egy másik játékos ellen játszik, akkor erre az esetre ír egy programot, amelyikbe betáplálva a lépéseket az egy olyan lépést mutat, amit esetleg a játékos magától nem lépett volna, ezáltal hozzásegítve egy jobb helyezés eléréséhez.

Ez a probléma nem ismeretlen a levelezési-sakk világban, és igazából elfogadják. Megfigyelések szerint jelentősen nem befolyásolja a játékok kimenetelét, inkább csak a különbséget csökkenti a kezdő és haladó játékosok között. [21] Kérdés, hogy ez mennyire teszi tönkre a játékélményt, de ez inkább morális kérdés, mint a dolgozathoz szorosan kötődő technológiai.

Jó sakkozó, de gyenge programozó

A másik - és számunkra érdekesebb eset, - hogy mi van akkor, ha valaki jó sakkozó, de nem tud jó programot írni. Ekkor lehetséges az, hogy regisztrál egy bot felhasználói fiókot és azon keresztül játszik. Mivel mindenki a neten otthonról játszik, ezért itt sem ellenőrizhető, hogy a lépést egy bot vagy egy ember találta-e ki. A válasz erre talán egy **fordított CAPTCHA** lenne, amellyel lehetne ellenőrizni, hogy a lépéseket tényleg egy gépi intelligencia hozta-e meg, vagy egy ember. Hívhatjuk **fordított Turing-tesztnek** is. Vajon lehet-e olyan programot írni, amely eldönti egy interakcióról, hogy az géptől jön-e. Lehet-e a kaotikus emberi viselkedést majdnem pontosan detektálni és a szuper-

komplex gépi interakciótól megkülönböztetni? Felismerhető-e, ha egy ember megpróbálja imitálni egy bot viselkedését?

Irodalomjegyzék

[1] Sakk.hu. Sakk szabályok. http://www.sakk.hu/help/sakk_szabalyok.html, 2008. [Online; Megtekintve: 2018 szeptember 4].

[2] Chess StackExchange. What is the average length of a game of chess? https://chess.stackexchange.com/questions/2506/what-is-the-average-length-of-a-game-of-chess, 2017. [Online; Megtekintve: 2018 szeptember 30.].

[3] Quora. What is the average length of a game of chess? https://www.quora.com/What-is-the-average-number-of-moves-in-a-chess 2017. [Online; Megtekintve: 2018 szeptember 30.].

[4] Buğra Fırat ("ebemunk"). A Visual Look at 2 Million Chess Games. https://blog.ebemunk.com/a-visual-look-at-2-million-chess-games/, 2016. [Online; Megtekintve: 2018 szeptember 30.].

[5] Wikipedia. Shannon-number. https://en.wikipedia.org/wiki/Shannon_number, 2018. [Online; Megtekintve: 2018 szeptember 30.].

[6] Nagy Sára Fekete István, Gregorics Tibor. *Bevezetés a mesterséges intelligenciába*. ELTE Eötvös Kiadó, 2006.

[7] Wikipedia. Representational State Transfer. https://hu.wikipedia.org/wiki/Alfa-b%C3%A9ta_v%C3%A1g%C3%A1s, 2018. [Online; Megtekintve: 2018 december 10.].

[8] Wikipedia. Representational State Transfer. https://en.wikipedia.org/wiki/Representational_state_transfer, 2018. [Online; Megtekintve: 2018 december 10.].

[9] Various. Swagger Homepage (swagger.io). https://swagger.io/, 2018. [Online; Megtekintve: 2018 szeptember 30.].

[10] Wikipedia. Sakk-matt. https://hu.wikipedia.org/wiki/Sakk-matt, 2018. [Online; Megtekintve: 2018 október 20.].

[11] Wikipedia. Checkmate. https://en.wikipedia.org/wiki/Checkmate, 2018. [Online; Megtekintve: 2018 október 20.].

[12] Wikipedia. A sakk története. https://hu.wikipedia.org/wiki/A_sakk_t%C3%B6rt%C3%A9nete, 2018. [Online; Megtekintve: 2018 október 20.].

[13] Wikipedia. History of chess. https://en.wikipedia.org/wiki/History_of_chess, 2018. [Online; Megtekintve: 2018 október 20.].

[14] Wikipedia. A Török (sakkozógép). https://hu.wikipedia.org/wiki/A_T%C3%B6r%C3%B6k, 2018. [Online; Megtekintve: 2018 október 20.].

[15] Wikipedia. The Turk (Automaton Chess Player). https://en.wikipedia.org/wiki/The_Turk, 2018. [Online; Megtekintve: 2018 október 20.].

[16] Walter B. Gibson. *Csodák könyve avagy vigyázat csalók!* Sport Lap- és Könyvkiadó, 1988.

[17] Wikipedia. Sakk - Számítógépes sakkprogramok. https://hu.wikipedia.org/wiki/Sakk#Sz%C3%A1m%C3%ADt%C3%B3g%C3%A9pes_sakkprogramok, 2018. [Online; Megtekintve: 2018 október 21.].

[18] Wikipedia. Kaissa. https://en.wikipedia.org/wiki/Kaissa, 2018. [Online; Megtekintve: 2018 október 21.].

[19] Wikipedia. Deep Blue (chess computer). https://en.wikipedia.org/wiki/Deep_Blue_(chess_computer), 2018. [Online; Megtekintve: 2018 október 21.].

[20] Wikipedia. Deep Blue (magyar). https://hu.wikipedia.org/wiki/Deep_Blue, 2018. [Online; Megtekintve: 2018 október 21.].

[21] Wikipedia. Levelezési sakk - A számítógép alkalmazása. https://hu.wikipedia.org/wiki/Sz%C3%A1m%C3%ADt%C3%B3g%C3%A9ppel_t%C3%A1mogatott_sakkoz%C3%A1s, 2018. [Online; Megtekintve: 2018 december 10.].