SZAKDOLGOZAT

Hucker Dávid

Debrecen

2010

Debreceni Egyetem

Informatikai Kar

Kétszemélyes játékok fejlesztése Java-ban

Témavezető:

Jeszenszky Péter

Egyetemi adjunktus

Készítette:

Hucker Dávid

Programtervező informatikus

Debrecen

2010

Tartalomjegyzék

Bevezetés

A játék fontos része az emberek életének, gyermekként a játék során tudjuk megismerni a körülöttünk lévő világot, felnőttként pedig kikapcsolódást nyújthat nekünk. A mai rohanó világban sajnos egyre kevesebb lehetősége van az embereknek játszani, a hétköznapi társasjátékok már közel sem olyan kedveltek és gyakran játszottak mint pár évvel ezelőtt, a játékvilágra is nagyon nagy hatással volt az informatikai fejlődés. A gyerekek már nem legóznak, hanem számítógépes szimulációkon próbálják elsajátítani és megtanulni a fontos értékeket és képességeket, a kamaszok közös, személyes jelenlétet igénylő játékok helyett inkább internetes közösségekhez csatlakozva játszanak és szocializálódnak, a felnőttek pedig pihenés gyanánt szintén valamilyen számítógépes játékot választanak. Nem hiába léteznek és nem hiába olyan sikeresek a világ legnagyobb játékgyártó cégei (Blizzard, EA Sports, Firaxis, …). Ők egy olyan piacra termelnek, aminek mindig lesz felvevő oldala, az emberek szeretnek játszani életkortól, nemtől, beosztástól és mindentől függetlenül. Sajnos az emberek többsége ha választhat egy szép időben és környezetben eltöltött szabadidős tevékenység és egy számítógépes játék között, akkor az emberek nem kis hányada az utóbbit lehetőséget fogja választani (akár kényelemből, akár lustaságból vagy más indokok miatt). Persze nem könnyű olyan játékokat fejleszteni, amelyek hatalmas tömegeket vonzanak, de mindenképpen nagyon érdekes és kihívásokkal teli lehet egy nagy közönség által kedvelt és elfogadott játék létrehozása. A dolgozat íróját is hasonló indokok vezették, hogy belekezdjen egy kétszemélyes, egyszerű szabályokon és logikán alapuló játék fejlesztésébe. Szeretnénk, ha minél több ember ismerhetné meg ezt az egyaránt kellemes időtöltésre alkalmas és mellette gondolkodásra is késztető játékot, és egy kis kikapcsolódást és a világ gondjaitól egy kis időre elszakadást tudnánk hozni a játékosoknak. Feladatunk nem egyszerű, de kis odafigyeléssel és jó kapott tanácsokkal sikeresen véghezvihetjük a célkitűzéseinket. Az író reméli hogy a leírásban mindenki fog találni valamilyen számára érdekes részt vagy valamilyen segítséget egy hasonló jellegű probléma megoldásához.

Célunk egy kétszemélyes játék asztali és webes verziójának elkészítése: az asztali verzió elsődleges célja az ismerősök elleni és emberi-gépi játékosok közötti játék biztosítása, a webes változatnak pedig a tetszőleges emberi ellenfelek közötti játék megvalósítása.

1. A kétszemélyes játékokról általánosságban

A kétszemélyes játékok a játékoknak már egy szűkebb csoportja, hiszen nem csak kétszemélyes játékok léteznek, hanem 1,2,..,n személyes játékokról is tudunk beszélni, illetve a játékok legfelsőbb szintű osztályzása szerint létezik két nagy csoport:

- szerencsejátékok, ahol a játék során minden a véletlen műve, a játékosoknak nem tudják befolyásolni a játékok kimenetét,
- a másik nagy csoport pedig a stratégiai játékok, ahol bár előfordulhat, hogy a véletlennek van bizonyos méretű szerepe a játékban, de a játékosok befolyásolni tudják a játék kimenetelét.

A stratégiai kétszemélyes játékokat különböző jellemzők alapján még több csoportba lehet sorolni. A leggyakoribb jellemzők:

- a játékban van-e szerepe a véletlennek (*determinisztikus / sztochasztikus*)
- a játékosok minden információval rendelkeznek-e a játékról a játék folyamán (*teljes / részleges információjú*)
- az egyik játékos nyereménye megegyezik-e a másik játékos veszteségeivel (*zérusösszegű* vagy *nem zérusösszegű*)
- minden állásban a játékosoknak végtelen sok lehetséges lépése van-e és a játszmák véges sok lépés után befejeződnek-e (*véges / nem véges játékok*)

Ezekkel a jellemzőkkel lehet a legjobban leírni egy-egy stratégiai játékot. A továbbiaknak olyan játékot tekintünk, amely **determinisztikus** (a véletlennek nincsen szerepe), **diszkrét** (a játék állásból állásba vivő lépések sorozata), **teljes információjú** (mindkét játékos rendelkezik minden információval a játékkal kapcsolatban), **zérus összegű** (egy játékos nem tud többet nyerni, mint amennyit a másik veszített) és **véges** (minden állásból véges sok lépés hajtható végre és véges sok lépés után véget ér a játék).

Ha megvannak a főbb tulajdonságaink a játékról, valamilyen módon a számítógép számára feldolgozhatóvá kell tennünk. Erre többféle módszer is adott, és a játékunk problémavilágától és komplexitásától függ, hogy melyiket érdemes alkalmazni. A leggyakrabban használt játékreprezentációs módszer az állapottér-reprezentáció, amely során megadunk egy olyan *<A, kezdő, C, O>* négyest, ahol:

- *A* halmaz elemei olyan (a, J) párok ahol a ∈ H egy állása a játéknak, J ∈ {A, B} a soron következő játékos,
- kezdő ∈ H a játék kezdőállapota,
- *C* ⊂ H olyan (c, J) párok ahol c ∈ H a játék egy végállapota és J játékos veszít,
- *O* nem üres halmaz az operátorok halmaza.

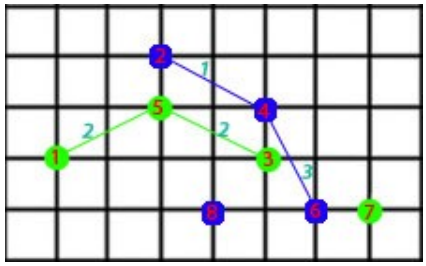
Egy ilyen állapotér-reprezentációt szemléltethetni tudunk egy játékfával, ahol a fa gyökere a kezdőállapot, a faelemek a kezdőállapotból elérhető játékállapotok, a fa levélelemei pedig a célállapotok.

2. A megvalósítandó játékról

Az implementálásra kerülő játék egy 20*20 darab rácspontot tartalmazó négyzetrácsos területen lehet játszani az alábbi egyszerű szabályok keretében:

- a játékosok célja, hogy az általuk kiválasztott irányban (az egyik játékos függőlegesen, a másik vízszintesen) a rácspontokat tartalmazó terület két egymással szemben szélét (alapvonalakat) egy összefüggő vonallal összekösse (az irányokról a játék elején meg kell egyezni),
- a játékosok egymás után, a játékmezőn lévő bármely rácspontot kiválaszthatják, kivéve azokat, amelyek már korábban ki lettek választva, illetve az ellenfél által összekötendő két alapvonalon lévő rácspontokat,
- ha a játékos lépése után van két olyan rácspont, amelyik egymáshoz képest lólépésben helyezkedik el, akkor az összes ilyen pontot össze kell kötnie, kivéve ha a behúzandó vonal metszené az ellenfél egy egyenesét,
- az a játékos nyer, aki előbb köti össze egy szakadás nélküli vonallal a két alapvonalát.

A szabályok első olvasásra lehet bonyolultnak tűnnek, de egy ábrával szemléltetve könnyebben megérthetővé válnak:



1. ábra: Példa a szabályok bemutatására

Ahogy az ábrán is látható, a 4. pont kiválasztása után vonalat húzunk közte és a tőle lólépésben lévő 2. pont között. Az 5. ponttól az 1. és a 3. pont is lólépésnyi távolságra van, így két vonalat is be kell húznunk. Viszont a 7. pont és 3. között nem húzunk vonalat, mert akkor kereszteznénk a 4. és 6. közötti vonalat. Ugyanilyen megfontolás miatt a 8. pontot sem kötjük össze a 4. ponttal.

2.1 A játék formalizálása

A játékot állapottér-reprezentációs módszerrel fogjuk formalizálni, melynek lépései:

1. A két játékost A és B fogja jelölni.
2. Meghatározzuk a játékállások halmazát. Egy játékállás egy 20*20 méretű mátrix. A különböző játékállásokra nem fogunk külön kényszerfeltételeket szabni, mert az operátorok alkalmazási előfeltételeit úgy alkotjuk meg, hogy azok mindig játékállásból játékállásba vigyenek.
3. Meghatározunk egy kezdőállapotot, ami nem lesz más, mint az a 20*20 mátrix, amelynek minden eleme nulla, és vagy A vagy B játékos kezdi a játékot.
4. Definiálunk egy kivalaszt(sor, oszlop) operátort. Alkalmazásának előfeltétele a játék szabályaiban leírtakat tükrözi, végrehajtása után a mátrix megfelelő sor és oszlop indexű tagja {1,2} lesz attól függően, hogy az A vagy B játékos lépett.
5. Megadjuk a célállapotok halmazát, azaz minden olyan játékállást, ahol a játék szabályaiban leírtak szerint véget ér a játék.

2.1.1 Állapottér

A játék világának állapotait rendezett elem 400-asokkal írhatjuk le. Az állapottér $A \subseteq A_{1,1} \times \cdots \times A_{20,20} \times J$ ahol a Descartes-szorzatban szereplő halmazok:

$$A_{ij} = \{0,1,2\}, J = \{1,2\} \qquad i = 1,\ldots,20 \qquad j = 1,\ldots,20.$$

Az $\left(\begin{pmatrix} a_{1,1} & a_{1,20} \\ \cdots & \cdots \\ a_{20,1} & a_{20,20} \end{pmatrix}, l \right) \in A$ állapotban $a_{ij} = \begin{cases} 0, \text{ ha még senki se választotta ki a pontot} \\ 1, \text{ ha az első játékos választotta ki a pontot} \\ 2, \text{ ha a második játékos választotta ki a pontot} \end{cases}$

ahol $i, j = 1,\ldots,20$ és l annak a játékosnak a sorszáma aki éppen lépni készül.

2.1.2 Kezdőállapot

$$k=\left(\begin{pmatrix} 0 & 0 \\ \cdots & \cdots \\ 0 & 0 \end{pmatrix}, 1\right)$$

2.1.3 Operátorok

Egy operátort definiálunk, amely az adott állapotban a paraméterként megadott sorban és oszlopban lévő pontot fogja kiválasztani az éppen soron következő játékosnak. A (sor, oszlop) értékpár adja meg, hogy melyik pontot kell kiválasztani.

A kivalaszt(sor, oszlop) az alábbi módon adható meg:

$$kivalaszt : dom(kivalaszt) \to A,$$

ahol

$$dom(kivalaszt) \subseteq P, \qquad P=\{(i,\ j)\,|\,1 \leqslant i \leqslant 20 \wedge 1 \leqslant j \leqslant 20\}$$

Az operátor hatása az $\left(\begin{pmatrix} a_{1,1} & a_{1,20} \\ \cdots & \cdots \\ a_{20,1} & a_{20,20} \end{pmatrix}, l\right) \in A$ állapot és $(s, o) \in P$ paraméter esetén, amelyekre teljesül az alkalmazási előfeltétel:

$$kivalaszt\left(\left(\begin{pmatrix} a_{1,1} & a_{1,20} \\ \cdots & \cdots \\ a_{20,1} & a_{20,20} \end{pmatrix}, l\right), sor, oszlop\right)=\left\{\left(\begin{pmatrix} a'_{1,1} & a'_{1,20} \\ \cdots & \cdots \\ a'_{20,1} & a'_{20,20} \end{pmatrix}, l'\right)\right\}$$

ahol

$$a'_{ij}=\begin{cases} l, ha\ i=sor\ és\ j=oszlop \\ a_{ij}\ egyébként \end{cases} \qquad l'=\begin{cases} 1, ha\ l=0 \\ 0, ha\ l=1 \end{cases}$$

Az operátor alkalmazásának előfeltételei:

Csak olyan pontot választhatunk ki, ami még korábban nem volt kiválasztva:

$$a_{sor,\ oszlop} \neq 0$$

A második játékos nem választhatja ki az első játékos alapvonalán lévő pontokat, csak a sarokban lévőt:

$$((sor=1 \vee sor=20) \wedge (oszlop \neq 1 \wedge oszlop \neq 20)) \supset j=1$$

Az első játékos nem választhatja ki a második játékos alapvonalán lévő pontokat, csak a sarokban lévőt:

$$((oszlop=1 \vee oszlop=20) \wedge (sor \neq 1 \wedge sor \neq 20)) \supset j=2$$

2.1.4 Célállapot

A célállapotok halmaza

$$C=\left\{\left(\begin{pmatrix} a_{11} & a_{120} \\ \cdots & \cdots \\ a_{201} & a_{2020} \end{pmatrix}, l\right) \in A : célfeltétel\left(\left(\begin{pmatrix} a_{11} & a_{120} \\ \cdots & \cdots \\ a_{201} & a_{2020} \end{pmatrix}, l\right)\right)\right\} \quad ,$$

ahol a célfeltételt megadó formula:

$$(((\exists(i,j)(i=1\wedge(i,j)\in S^1)\wedge\exists(x,y)(x=20\wedge(x,y)\in S^1)))\wedge(i,j)\in L^1\wedge(x,y)\in L^1)\vee$$
$$\vee(((\exists(i,j)(j=1\wedge(i,j)\in S^2)\wedge\exists(x,y)(x=20\wedge(x,y)\in S^1)))\wedge(i,j)\in L^2\wedge(x,y)\in L^2)$$

ahol

$$lolepes((i,j),(x,y))\equiv(i+1=x\wedge j+2=y)\vee(i-1=x\wedge j+2=y)\vee(i+1=x\wedge j-2=y)\vee$$
$$\vee(i-1=x\wedge j-2=y)\vee(i+2=x\wedge j+1=y)\vee(i-2=x\wedge j+1=y)\vee$$
$$\vee(i+2=x\wedge j-1=y)\vee(i-2=x\wedge j-1=y)$$

$$S^1=\{(i,j)|a_{ij}=1 \quad i,j=1,\ldots,20\}$$
$$S^2=\{(i,j)|a_{ij}=2 \quad i,j=1,\ldots,20\}$$
$$L^1=\{(i,j)|(i,j)\in S^1\wedge\exists(x,y)(lolepes((i,j),(x,y))\wedge(x,y)\in L^1)\}$$
$$L^2=\{(i,j)|(i,j)\in S^2\wedge\exists(x,y)(lolepes((i,j),(x,y))\wedge(x,y)\in L^2)\}$$

A $lolepes$ akkor lesz igaz ha a mátrixban két elem lólépésnyi távolságra van egymástól, S^1 az első játékos által kiválasztott mátrixelemek pozícióinak halmaza, S^2 a második játékos által kiválasztott mátrixelemek pozícióinak halmaza. L^1 az első játékoshoz tartozó, általa kiválasztott olyan mátrixelemek pozícióinak halmaza amelyek egymástól lólépésnyi távolságra vannak, L^2 pedig a második játékoshoz tartozó. Ezen halmazokra egy megkötés hogy az első elem amit tartalmaznak az vagy $(i,j)(i=1\wedge(i,j)\in S^1)$ az L^1 halmaz esetén, és $(i,j)(j=1\wedge(i,j)\in S^2)$ ha az L^2 halmazról van szó.

Ezzel a $<A,k,C,kivalaszt>$ négyessel megadtuk a játéknak egy lehetséges állapottér-reprezentációját.

3. A játék megvalósítása

A játék két formában lesz elkészítve: először egy tisztán Java nyelven megírt és Swing felülettel megvalósított asztali verziót készítünk el. Ennek elkészülte után a Google Web Toolkit (GWT) segítségével egy Java és JavaScript nyelveken megvalósított, JDBC és Apache Maven segítségével elkészített, SmartGWT felülettel rendelkező, webböngészőben is futtatható és interneten keresztül bárhonnan játszható megvalósítást implementálunk.

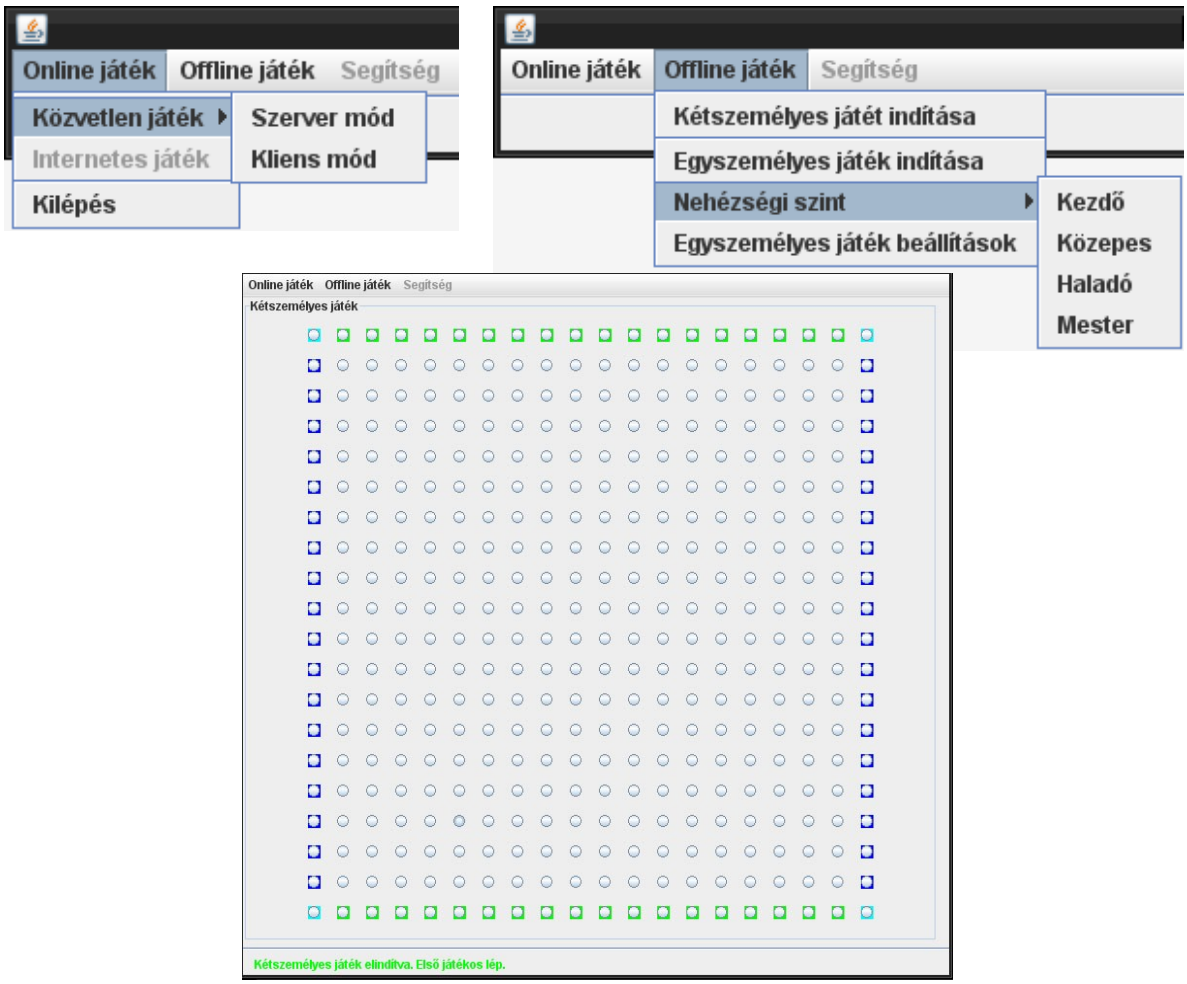
3.1 A játék asztali verziója

A feladat elkészítése során kezdetben a NetBeans fejlesztői környezettel fogunk dolgozni, mert a segítségével nagyon gyorsan és szép grafikus felhasználói felületet (GUI) lehet létrehozni, majd ezek után az Eclipse Galileo IDE keretein belül fogjuk folytatni a fejlesztést.

Az implementáció során a következő funkciókat kell beépíteni a rendszerbe:

- Kétszemélyes játék egy számítógépen: két játékos tudjon egymás ellen játszani, ha mindketten egy számítógép előtt ülnek.
- Kétszemélyes játék két számítógépen: két játékos tudjon egymás ellen játszani hálózaton keresztül.
- Egyszemélyes játékmód: egy játékos tudjon a számítógép ellen játszani különböző nehézségi fokozatokon.
- Egyszemélyes játékmód állíthatósága: különböző keresőalgoritmusok és különböző heurisztikák biztosítása.
- A keresőalgoritmusok által a keresés során felépített játékfák megjelenítése igény szerint.

Mielőtt nekiállnánk a fő funkcióknak, először egy egyszerű GUI-t fogunk elkészíteni. A felhasználói felület a 2. ábrán látható menürendszert és játékfelületet tartalmazza. Külön csoportosítottuk hálózati és a hálózat nélküli módokat, a játék fő tábláját ahol majd a játék zajlani fog, és legalulra pedig elhelyeztünk egy információsávot ahol majd értesítjük a felhasználót a játékkal kapcsolatos különböző eseményekről.



2. ábra : A létrehozott GUI

3.1.1 Az állapottér-reprezentáció implementálása

Következő lépésben el kell készítenünk kód szinten a játék állapottér-reprezentációját. A játékunkat egy saját `AbstractGame` osztályból fogjuk származtatni, amely osztály definiálja a legfontosabb konstansokat, attribútumokat (a tábla mérete, az aktuálisan lépő játékos, operátorok halmaza) illetve a legfontosabb metódusokat, amiket minden ebből az osztályból kiterjesztett játékosztálynak tartalmaznia kell:

- `protected byte[][] gameField`

  A játéktábla, dimenzióinak mérete a *`TableSize`* konstans értéke lesz.

- `public abstract AbstractGame executeOperator(AbstractOperator op)`

  Az operátor alkalmazásának hatását ebben a metódusban kell implementálni.

- `public abstract boolean isGameEndState()`

  A célállapot vizsgálatát ebben a metódusban kell implementálni.
- `public abstract boolean isOperatorUsable(AbstractOperator op)`

  Az operátor alkalmazási előfeltételének vizsgálatát ebben a metódusban kell implementálni.
- `public abstract double heurisztika()`

  Az adott játékállapothoz tartozó heurisztika értéke ennek a metódusnak a visszatérési értéke lesz.

A `Game` osztály fogja kiterjeszteni az előbbi osztályunkat, itt fogjuk megvalósítani az állapottér-reprezentációhoz tartozó feladatokat. A kezdőállapotunk az az állapot lesz, amikor a `gameField` mátrix mindegyik eleme nulla. A nulla fogja azt jelenteni, hogy az adott sor és oszlop indexű csomópontot még nem kiválasztotta ki egyik játékos sem. Ez a kinullázás az osztály mindegy egyes példányosításakor végre fog hajtódni. Mint azt korábban említettük, az egyes játékállapotokra nem fogunk külön kényszerfeltételeket szabni, mert majd az operátorunkat úgy fogjuk mindjárt megalkotni, hogy az mindig állapotból állapotba fogja vinni a játékállásokat. Az operátorunkat a `SelectNodeOperator` osztály valósítja meg, melyben látszik, hogy két paramétere lesz, egyik a kiválasztott csomópont sorindexe, másik pedig a csomópont oszlopindexe. Létezik egy absztrakt operátor osztályunk is, az `AbstractOperator`, amit akkor használunk ha több operátorunk van, ebből kell származtatni minden operátort. Az operátorunk alkalmazásának előfeltételének az implementálása nyilvánvaló az állapottér-reprezentációból ezért ezt nem adjuk meg pszeudó-kóddal. Az operátor hatása pedig nem más, mint a `gameField` megfelelő sor és oszlopindexű elemének a játékos kódjára való állítása. A célállapot-vizsgálatot a következőképpen fogjuk elvégezni:

```
FUNCTION celallapot(elsojatekosvonalak, masodikjatekosvonalak)
1; IF lehetnyert(allapot, elsojatekos) THEN
2;   FOR ALL vonal ∈ elsojatekosvonalak DO
3;     IF vonal.kezdopont.sorindex = 0 THEN
4;       RETURN utkereso(elsojatekosvonalak,
                                  vonal.kezdopont, elsojatekos)
5;     ELSE
6;       IF vonal.vegpont.sorindex = 0 THEN
```

```
7;             RETURN utkereso(elsojatekosvonalak,
                                    vonal.vegpont, elsojatekos)
8;         END IF
9;       END IF
10;   END FOR
11;  ELSE
12;   IF lehetnyert(allapot, masodikjatekos) THEN
13;         FOR ALL vonal ∈ masodikjatekosvonalak DO
14;              IF vonal.kezdopont.oszlopindex = 0 THEN
15;                   RETURN utkereso(masodikjatekosvonalak,
                                vonal.kezdopont, masodikjatekos)
16;              ELSE
17;                   IF vonal.vegpont.oszlopindex = 0 THEN
18;                        RETURN utkereso(masodikjatekosvonalak
                                    vonal.vegpont, masodikjatekos)
19;                   END IF
20;              END IF
21;         END FOR
22;   END IF
23;  END IF
24;  RETURN false
END FUNCTION


FUNCTION utvonalkereses(jatekosvonalak, aktvegpont, jatekos)

1;  IF alapvonalpont(aktvegpont, jatekos) THEN
2;    RETURN true
3;  END IF
4;  IF jatekosvonalak.meret = 0 THEN
5;    RETURN false
6;  END IF
7;    vonalak ← jatekosvonalak
8;    FOR ALL vonal ∈ vonalak DO
9;         IF vonal.kezdopont = aktvegpont THEN
10;             jatekosvonalak ← jatekosvonalak \ { vonal }
11;             IF utvonalkereses(jatekosvonalak, vonal.vegpont,
                                          jatekos) = true
12;                  RETURN true;
13;             END IF
14;        ELSE
15;             IF vonal.vegpont = aktvegpont THEN
16;                  jatekosvonalak ← jatekosvonalak \ { vonal }
17;                  IF utvonalkereses(jatekosvonalak,
                                vonal.kezdopont, jatekos) = true
18;                       RETURN true;
19;                  END IF
```

```
20;             END IF
21;         END IF
22;   END FOR
23; RETURN false
END FUNCTION
```

Az *elsojatekosvonalak* és *masodikjatekosvonalak* az adott játékos által húzott vonalak halmaza. A pszeudó-kódban két olyan függvény szerepel, amelyek értelmezésre szorulnak. A `lehetnyert(`*allapot, jatekos*`)` megvizsgálja, hogy az adott állapotban az adott játékosnak a két alapvonalán külön-külön van-e egy-egy pont kiválasztva (ha igen, akkor lehet hogy célállapottal van dolgunk). Az `alapvonalpont(`*aktvegpont, jatekos*`)` pedig azt figyeli, hogy ha az első játékosról van szó, akkor az aktuális végpont az alsó alapvonalon van-e, ha a második játékosról, akkor a végpont a jobb oldali alapvonalon található-e.

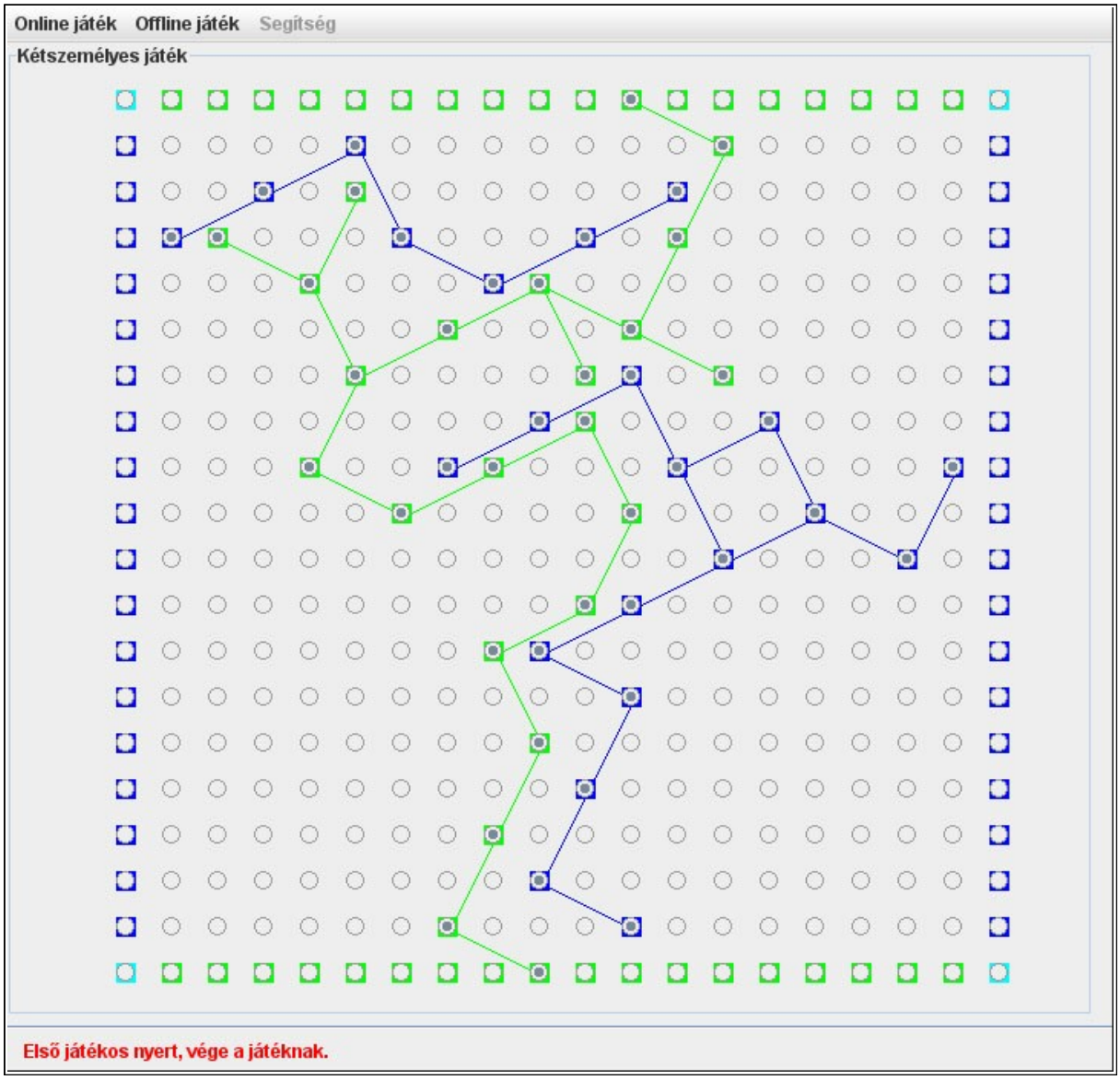
Ha az első játékosnak lehet nyerő állása (azaz a `lehetnyert` függvény értéke igaz), akkor elindulunk a legfelső alapvonalról, és onnan a játékosnak a már behúzott vonalain keresztül megpróbálunk eljutni az alsó alapvonalra. Ha a második játékosnak, akkor hasonlóan a bal oldali alapvonalról indulunk és próbálunk meg eljutni a jobb oldali alapvonalra. Bármilyen bonyolult vonalrendszert kialakíthatnak a játékosok, akár hurkokat tartalmazót is, hiszen amikor találunk egy olyan vonalat, amelyiknek az egyik alappontja megegyezik az aktuális végpontunkkal, akkor a vonal másik alappontját fogjuk végpontnak tekinteni, és ezzel a vonallal a keresés során többé nem fogunk foglalkozni, így nem futhat az algoritmus egy hurok mentén örökké. Tulajdonképpen az összes, alapvonalon lévő kiválasztott rácsponthoz létrehozunk egy olyan irányított gráfot, ahol a gráf egymást követő csúcsai az egymástól lólépésnyi távolságra lévő, adott játékos által kiválasztott rácspontok lesznek. Ezt a gráfot rekurzívan bejárva próbálunk meg egy olyan pontot találni, amelyik a másik alapvonalon található.

Ezzel befejeztük a játék állapottér-reprezentációjának az implementálását, a következő feladatunk az, hogy egy eseménykezelő rendszeren keresztül összekössük a felhasználói felületet a reprezentációnkkal, amely után már kétszemélyes játékot fogunk tudni játszani. Ha ez elkészül, onnan csak egy lépés a hálózaton keresztüli játék megvalósítása, végezetül pedig jöhetnek a keresőalgoritmusok és a heurisztikák az egyszemélyes verzióhoz.

3.1.2 Összekapcsolás a felhasználói felülettel

A GUI megvalósításának nagy része az `SzdView` osztályban található meg. A játékmezőn kívül mindent itt hozunk létre és állítunk be, példának okáért a menüelemeket és a hozzájuk tartozó funkciókat (nehézségi szintnek a beállítása) vagy az alsó információs panelen megjelenő üzeneteket.

Sokkal érdekesebb azonban a `Grid` osztály. Ez egy JPanel lesz, amelyet majd később szabadon hozzáadhatunk a főablakunkhoz. Az egyes megjelenítendő csomópontok a `Node` osztály példányai lesznek, amelyek tartalmazni fogják az adott csomópontot szimbolizáló `JRadioButton`-t, az ezt kiválasztó játékos sorszámát, a csomópont sor és oszlopindexeit, illetve azt a pontot, ahol a rádiógombunk a játék paneljén elhelyezkedik. Ezeket a csomópontokat a *`nodes`* attribútumban fogjuk tárolni, emellett nyilván lesznek tartva az első, illetve a második játékos által kiválasztott csomópontok. Amikor létrehozzuk az egyes csomópontokat, ahhoz hogy kezelni tudjuk az egyes játékosok lépését, hozzá kell adnunk minden egyes rádiógombhoz egy eseménykezelőt, amely majd a gombra kattintás esetén megpróbálja végrehajtani a játékos által meglépett lépést. Ezt a funkciót a *`nodeSelected`*`(x, y)` metódus fogja ellátni, mely első lépésben megvizsgálja, hogy az adott játékálláson a lépést el lehet-e végezni az állapottér-reprezentációban megadott operátor alkalmazási előfeltételek szerint. Ha úgy találja, hogy végrehajtható, akkor engedi hogy az állapoton végrehajtódjon az operátor, regisztrálja az új állapotot, feljegyzi magának a megfelelő attribútumaiban is a lépést, informálja a játékost hogy az ellenfél következik, és a GUI-n is elvégzi a szükséges változtatásokat. Végül megvizsgálja, hogy az újonnan létrejött játékállapot célállapot-e, és ha igen, akkor arról tájékoztatja a játékosokat. Mindezek után meghívódik a panel kirajzolásáért felelős módosított `public void paintComponent(Graphics g)` metódus, amely kirajzolja a játékosok azon vonalait, amelyeket a játékszabályok szerint hoztak létre.



3. ábra : Egy példajáték

3.1.3 Hálózati játék

Eddig a pontig elkészítettük a játéknak azt a kétszemélyes verzióját, amivel már lehet játszani, viszont nem praktikus, hiszen a két vagy többszemélyes játékokat az emberek napjainkban már általában nem egy számítógép előtt ülve szokták játszanak, hanem mindenki a saját gépe előtt ülve szeretne játszani másokkal. A dolgozat második felénél létre fogjuk hozni ennek a játéknak egy webböngészőben játszható verzióját, ami ezt a problémát le fogja fedni. Viszont ha nem szerepel a terveink között ilyen rendszer készítése, akkor valamilyen

más módon kell biztosítanunk a játékosok közötti kapcsolatot. Mi egy nagyon egyszerű módszerrel fogjuk ezt megvalósítani, a Java hálózati könyvtára (java.net) által szolgáltatott TCP/IP protokollra épülő szerver-kliens kommunikáció segítségével. Ezzel két játékos úgy fog tudni játszani egymással, hogy az egy játékos gépe szerverként fog működni, amelyhez megadja az IP címét és hozzá egy portot egy másik játékosnak. Ő ezek ismeretében képes lesz kliensként csatlakozni a szerverhez és két különböző számítógép előtt ülve fognak tudni egymás ellen játszani. A kapcsolat létrejötte után a játékosok lépéseit el kell juttatni a másik játékoshoz is. A játék befejeztével a kapcsolat megszakad a két játékos között, de bármikor indíthatnak akár egymás vagy egy másik ellenfél ellen egy új játékot.

A meglévő programunkon nem kell sok változtatást eszközölni, hogy az előbbiekben leírt funkcionalitást is beletegyük. Négy új osztályra lesz szükségünk, melyek a `Client`, `ClientServiceThread`, `Server` és `ServerServiceThread`. A `Client` és `Server` osztályok a megfelelő dialógusablakoknak felelnek meg, a `ClientServiceThread` és a `ServerServiceThread` pedig a tényleges a kapcsolat felépítéséért és a két végpont közötti kommunikációért lesznek felelősek. A szerver oldalon az IP címet a játékosnak nem kell előkeresnie, a program automatikusan lekéri, viszont mód van az alapértelmezett portot átállítani egy másikra (értelemszerűen a csatlakozni kívánó játékosnak is ugyanarra a portra kell csatlakozni). A szerver elindítása után a kliensnek 30 másodperce van arra, hogy becsatlakozzon, ha ennyi idő alatt ez nem történik meg, akkor a szerver leáll, a kapcsolódási hibáról pedig értesítve lesz a kliens és a szerveroldali játékos is. Számtalan oka lehet a kapcsolódás sikertelenségének, leggyakrabban azonban vagy egy tűzfal, vagy pedig egy hálózati eszköz tilthatja le a kapcsolódást. Az információcseréért felelős másik két osztály implementálja a `Runnable` interfészt, ami azt jelenti, hogy a kommunikáció a két fél között egy-egy külön szálon fog a háttérben futni. Ha egy játékos kiválaszt egy pontot, akkor egy „(x,y)” alakú üzenet kerül elküldésre automatikusan az ellenfélhez, amit az üzenet fogadója úgy fog értelmezni, hogy az ellenfél az x. sorban és y. oszlopban lévő pontot választotta ki, és ennek megfelelően változtatni kell mind a játékállapoton, mind a játékmezőn.

Kódszinten annyit kell csak változtatnunk, hogy a rádiógombokhoz tartozó eseménykezelőbe bele kell írni, hogy abban az esetben ha egyszemélyes játékot játszunk akkor a lépésünknek a (sor,oszlop) koordinátái legyenek eljuttatva az ellenfél játékoshoz is. Az üzenetküldésért felelős osztályokban pedig figyelnünk kell a másik játékostól érkező

üzeneteket. Ha érkezik egy (sor,oszlop) koordinátapár, akkor mindent ugyanúgy kell végrehajtani, mintha mi választottuk volna ki az adott csomópontot, tehát csak meg kell hívnunk a *nodeSelected*(sor, oszlop) metódust.

Ezzel a játék kétszemélyes részének a fejlesztését befejeztük, minden kitűzött célt elértünk és minden problémát megoldottunk, a programunk több módon képes az ember-ember játékot megvalósítani, így most az egyszemélyes, gépi ellenféllel való játék megvalósítása fog következni.

3.1.4. Keresőalgoritmusok

Már korábban megemlítettük hogy egy játék állapottér-reprezentációját lehet szemléltetni egy gráf segítségével. Ha ezt a gráfot „kiegyenesítjük" akkor kaphatunk egy a játék minden lehetséges játszmáját szemléltető fát. Erre a játékfára azért lesz szükségünk, mert ebben a fában különböző kereséseket és műveleteket végrehajtva értékelni tudjuk majd az állásokat és lépéseket, és ezek segítségével lépést fogunk tudni ajánlani a gépnek. Az emberi játékok során is hasonló történik, például egy sakkjátszma során annak a játékosnak van több esélye a játék megnyerésére, aki több lépésnyire tud előregondolkozni és minél több lépéskombinációt tud mérlegelni. Ez a játékfában szemléltetve annak valamilyen mélységű és szélességű bejárását fogja jelenteni. Még mielőtt kitérünk az algoritmusok jellemzőire és megvalósításukra, megjegyezzük, hogy a problémamegoldás sebessége és eredménye, illetve a lépésajánlás nagyon függ a játék komplexitásától és a játékfa méreteitől. A mi esetünkben, ahogy látni fogjuk később, sajnos a fa túl nagy mérete nagy mértékben korlátozni fogja a lépésajánlást. Ha azt mondjuk, hogy a játékfát négy mélységig építjük föl, és mondjuk az emberi játékos első lépése után szeretnénk a gépnek egy lépést ajánlani, akkor a 20*20 méretű táblán a már kiválasztott és az ellenfél alapvonalain kívül 400 – 18 * 2 – 1 = 363, a következő szinten az emberi játékos 362, utána a gép 361, végül a negyedik lépésként pedig az ember 360 lehetséges pont közül tudna a választani. Ez az jelenti, hogy a játékfa első szintje 363, a második szintje 363 * 362, a harmadik szinten 363 * 362 * 361, a negyedik szinten pedig 363 * 362 * 361 * 360 faelemet tartalmazna, ami összesen 17.124.829.920, azaz több mint tizenhét milliárd bejárandó faelem. Ennyi elemnek a bejárására és kiértékelésére az algoritmus képtelen reális időn belül.

A lépésajánláshoz három keresőalgoritmust fogunk implementálni, egy minimax algoritmust, egy alfa-béta vágásos keresőt és egy heurisztikus minimax keresőt. Mindhárom keresőt egy `AbstractKereso` osztályból fogunk származtatni, az absztrakt osztály legfontosabb attribútumai és metódusai:

- **`protected int`** `maximumDepth`
  Az adott keresőalgoritmus milyen mélyen építse fel a játékfát.
- **`protected`** `Csucs gyokerElem`
  Az az állás a játékban, amelyből fel kell építeni a játékfát. A `Csucs` osztály példányai lesznek a faelemek, tárolni fogják a szülő faelemet, a csúcsot a szülőből létrehozó operátort, az operátor alkalmazása utáni állapotot, a mélységi számot és egy heurisztikus értéket.
- **`public`** `SelectNodeOperator operator`
  A játékos számára ajánlott lépést.
- **`protected abstract double`** `createGameTree(int depth, DefaultMutableTreeNode parentNode, Csucs actRoot)`
  A játékfát létrehozó metódus, ezt az absztrakt metódust kell mindegyik keresőnek implementálnia.

A `gyokerElem`hez tartozó kiinduló játékállapotot paramétert mindegyik kereső konstruktorhívásánál meg kell adni, majd utána a `createGameTree(…)` metódus meghívásával hozható létre a játékfa.
Az absztrakt osztályban definiálunk egy **`public void`** `showTreeWindow(AbstractKereso kereso)` metódust, aminek a segítségével meg tudjuk jeleníteni a felépített játékfát egyszerű módon. Explicit módon meghívódik a játékfa felépítése után, ha a **`public`** `<`*`keresőNév`*`>(Game g, int depth, boolean treeVisualization)` konstruktor hívása során a `treeVisualization` paraméter értéke **`true`**.

3.1.4.1 Minimax algoritmus

Klasszikus lépésajánló algoritmus, nagy fák esetén időigényes lehet a végrehajtás mert adott mélységig minden csúcsot megvizsgál. Működésének pszeudó-kódja:

```
FUNCTION MiniMaxKereso(faelem)
1; IF faelem.melyseg = maxmelyseg OR faelem.jatekallapot ∈ C
                                                          THEN
2;   return faelem.jatekallapot.heurisztika
3; END IF
4; max ← -INF
5; FOR ALL operator ∈ faelem.hasznalhatooperatorok DO
6;   gyermek ← ujelem(faelem,operator)
7;   max ← MAX(max,-(MiniMaxKereso(gyermek)))
8;   faelem.heurisztika ← max
9; END FOR
10; RETURN max
END FUNCTION


FUNCTION ujelem(szulo, operator)
1; gyermek.melyeg ← szulo.melyseg + 1
2; gyermek.jatekallapot ← alkalmaz(szulo.jatekallapot, operator)
3; gyermek.operator ← operator
4; gyermek.szulo ← szulo
5; szulo.gyermekek = szulo.gyermekek ∪ { gyermek }
6; RETURN gyermekelem
END FUNCTION
```

A kód írása során felhasználtuk a $max(a,b) = -min(-a,-b)$ azonosságot, így mindkét játékost egységesen fogunk tudni kezelni.

A keresés végén visszakapott max érték alapján úgy határozzuk meg a használandó operátort, hogy a gyökérelem gyermekein végigjárva kiválasztjuk azt, amelyiknek a heurisztikája egyenlő ezzel az értékkel és a hozzá tartozó részfa mélység a legkisebb. Így két azonos heurisztikájú állapotba vivő lépések esetén azt fogja választani, amelyik kevesebb lépésből éri el azt az állapotot. Az algoritmus úgy fogja az egyes faelemek heurisztikáját meghatározni, hogy tényleges heurisztikát csak a levélelemeknél fog számolni, az összes többi faelem heurisztikája pedig a gyerekelemek heurisztikájának minimuma / maximuma lesz attól függően, hogy páros / páratlan szinten állunk a játékfában. Ha páratlan szinten állunk, akkor azon olyan játékállások vannak, ahol az emberi játékos a soron következő

játékos. Mivel a heurisztikáinkat úgy fogjuk létrehozni, hogy azok mindig az első játékos számára adják meg az állásnak a jóságát, így ilyenkor a lehetséges lépések közül a legjobbat fogjuk választani, ezért a faelem heurisztikája a gyerekelemek heurisztikájának a maximuma lesz. Amikor páros szinten járunk, akkor a gépi játékos következik lépni, neki a célja az emberi játékost a lehető legrosszabb állásba vinni, így ezeknek a faelemeknek a heurisztikája egyenlő lesz a gyerekelemek heurisztikájának minimumával.

Ahogyan már említettük és ahogyan látszik is, az algoritmusnak szükséges adott mélységig a teljes játékfát bejárnia ahhoz, hogy lépést tudjon ajánlani, viszont ez a lépés a lehető legjobb lépés lesz. A szükséges műveletigény felső korlátja $O(l^m)$, ahol az l konstans, az egyes állásokban lévő lehetséges lépések száma és m pedig a játékfa mélysége. A fentebb leírt ujelem(*szulo, operator*) függvényt fogjuk használni az összes többi keresőben is. A keresés sebességére vonatkozó adatok a függelékben találhatóak.

3.1.4.2 Alfa-béta vágás

Ennek a keresőalgoritmusnak a minimaxhoz képest nem szükséges adott mélységig a játékfa összes elemét kiértékelnie a lépésajánláshoz. Az eredménye pedig ugyan olyan jó lépés lesz, amit a minimax algoritmus is szolgáltat. Viszont mivel csak a fának egy részét járjuk be, sokkal gyorsabban kapjuk meg az eredményt. A kereső pszeudó-kódja:

```
FUNCTION AlfaBetaKereso(faelem)
1; alfa ← - INF
2; beta ← + INF
3; IF faelem.melyseg % 2 = 0 THEN
4;   RETURN AlfaBetaMax(faelem, alfa, beta)
5; ELSE
6;   RETURN AlfaBetaMin(faelem, alfa, beta)
7; END IF;
END FUNCTION


FUNCTION AlfaBetaMax(faelem, alfa, beta)
1; v ← - INF
2; IF faelem.jatekallapot ∈ C THEN
3;   RETURN faelem.jatekallapot.heurisztika
4; ELSE
5;   FOR ALL operator ∈ faelem.hasznalhatooperatorok DO
```

```
6;        gyermek ← ujelem(faelem,operator)
7;        gyerekmin ← AlfaBetaMin(gyermek, alfa, beta)
8;        IF gyerekmin > v THEN
9;             v ← gyerekmin
10;       END IF
11;       IF v >= beta THEN
12;            return v;
13;       END IF;
14;       alfa ← MAX(alfa, v)
15;  END FOR
16;  faelem.heurisztika ← v;
17;  RETURN v
END FUNCTION

FUNCTION AlfaBetaMin(faelem, alfa, beta)

1; v ← + INF
2; IF faelem.jatekallapot ∈ C THEN
3;   RETURN faelem.jatekallapot.heurisztika
4; ELSE
5;   FOR ALL operator ∈ faelem.hasznalhatooperatorok DO
6;        gyermek ← ujelem(faelem,operator)
7;        gyerekmax ← AlfaBetaMax(gyermek, alfa, beta)
8;        IF gyerekmax < v THEN
9;             v ← gyerekmin
10;       END IF
11;       IF v <= beta THEN
12;            return v;
13;       END IF;
14;       beta ← MIN(beta, v)
15;  END FOR
16;  faelem.heurisztika ← v;
17;  RETURN v
END FUNCTION
```

Látható, hogy annak érdekében, hogy a minimax keresésnél gyorsabban kapjunk eredményt, egy bonyolultabb és többet számoló keresőt kell megalkotnunk. Az algoritmus működésének alapötlete, hogy egy adott lépés utáni állásban az ellenfélnek van egy olyan lépése ami számunkra rosszabb mint az eddigi lehetséges lépései, akkor ennek az állásnak a további lépéseit felesleges kiszámolnunk. Hiszen hogy az ellenfél ne tudja ezt a lépést választani végső soron úgyse ezt a lépést fogjuk választani. Ennek a részfának a vizsgálatának a kihagyását nevezzük alfa illetve béta vágásnak. Mivel kevesebb faelemet kell kiértékelnünk és bejárnunk, ezért ugyan annyi idő alatt a fának nagyobb részét tudjuk majd bejárni vagy

ugyan akkora farészt gyorsabban tudunk átvizsgálni, mint a minimax kereső esetén. Persze nagyon sok tényezőtől függ, hogy a fának mekkora részét nem fogjuk megvizsgálni. Az algoritmus műveletigénye $O(\sqrt{l^m})$ a legoptimálisabb esetben (amikor minden faelemnél az első lépésre megkapjuk a legjobb lépést) .

A vágások két érték alapján, az alfa és béta értékek alapján hajthatóak végre, ahol alfa nem más, mint az gépi játékos által az adott játékállásból elérhető minimális érték, a béta pedig a emberi játékos által az adott állásból elérhető maximális érték. A keresés sebességére vonatkozó adatok a függelékben megtalálhatóak, és azt várjuk, hogy gyorsabban kapjunk megoldást mint a minimax algoritmus esetén.

3.1.4.3 Heurisztikus minimax

Ha szeretnénk a korábbiaknál még gyorsabban lépést ajánlani a játékosunknak, akkor még tovább kell csökkentenünk a keresés során kiértékelt faelemeinknek a számát. Az alfa-béta vágás már nem vizsgálta meg az egész játékfát adott mélységig annak csak egy kisebb részét, de még a minimax kereséssel megegyező jóságú megoldást tudta szolgáltatni. Ha még tovább csökkentjük a megvizsgált elemeknek a számát, akkor sajnos már nem fogjuk tudni garantálni, hogy a legjobb operátort fogja nekünk az algoritmus ajánlani. A heurisztikus minimax kereső egy ilyen módszert fog megvalósítani, nagy mértékben le fogjuk tudni csökkenteni a bejárt elemek számát azzal, hogy nem csak a levélelemeknél fogunk tényleges heurisztikát számolni. Minden egyes faelem összes gyermekére kiszámoljuk a hozzájuk tartozó játékállások heurisztikus értékét, viszont a faelem gyermekei közül csak azoknál fogjuk tovább építeni a fát, amelyeknek a heurisztikája a legmagasabb / legalacsonyabb volt (attól függően hogy a fa páratlan / páros szintjén járunk). Az eljárás pszeudó-kódja a következő:

```
FUNCTION HeurisztikusMiniMaxKereso(faelem)
1; IF faelem.melyseg = maxmelyseg OR faelem.jatekallapot ∈ C
                                                        THEN
2;   return faelem.jatekallapot.heurisztika
3; END IF
4; max ← -INF
5; IF faelem.melyseg % 2 = 1 THEN
6;   heurisztika ← + INF
7;   FOR ALL operator ∈ faelem.hasznalhatooperatorok DO
```

```
8;        gyermek ← ujelem(faelem,operator)
9;        gyermek.heurisztika ←
                              gyermek.jatekallapot.heurisztika
10;       IF gyermek.heurisztika <= heurisztika THEN
11;            faelem.gyermekek ← faelem.gyermekek ∪ {gyermek}
12;            heurisztika ← gyermek.heurisztika
13;       END IF
14;  END FOR
15; ELSE
16;  heurisztika ← - INF
17;  FOR ALL operator ∈ faelem.hasznalhatooperatorok DO
18;       gyermek ← ujelem(faelem,operator)
19;       gyermek.heurisztika ←
                         gyermek.jatekallapot.heurisztika
20;       IF gyermek.heurisztika >= heurisztika THEN
21;            faelem.gyermekek ← faelem.gyermekek ∪ {gyermek}
22;            heurisztika ← gyermek.heurisztika
23;       END IF
24;  END FOR
25; END IF
26;  FOR ALL elem ∈ faelem.gyermekek DO
27;       IF elem.heurisztika = heurisztika
28;            max ← MAX(max, -
                         (HeurisztikusMiniMaxKereso(elem)))
29;            faelem.heurisztika ← max
30;       END IF
31;  END FOR
32; RETURN max
END FUNCTION
```

A sima minimax kereséshez képest annyit változtattunk meg a keresőn, hogy a játékfa bizonyos mélységig történő összes elemének bejárása helyett mindegyik elemnél attól függően, hogy páros / páratlan szinten vagyunk meghatározzuk az összes gyerekeleméhez tartozó játékállások heurisztikáját, és gyerekelemek közül csak a legkisebb / legnagyobb heurisztikájúaknál fogjuk tovább építeni a fát, a többivel nem foglalkozunk. A keresés előnye hogy ha jó a heurisztikánk, akkor faelemenként mindig csak egyetlen irányba kell tovább mennünk, így a keresés műveletigénye leredukálható $O(m*l)$ -re. Viszont mivel nem a részfa alapján határozzuk meg a faelem heurisztikáját, hanem az adott állás szerint, ezért előfordulhat, hogy bár pillanatnyilag egy állást nagyon jónak értékelünk, viszont hosszútávon lehet hogy egy másik lépéssel jobban jártunk volna. A keresés sebességére vonatkozó adatok a függelékben megtalálhatóak, és majd látni fogjuk hogy a különböző heurisztikák milyen különbségeket fognak okozni a lépésajánlásokban.

3.1.5 Heurisztikák

Különböző heurisztikákra lesz szükségünk ahhoz, hogy a már meglévő keresőink érdemi információt tudjanak nyújtani nekünk a játékfáról. A heurisztika nem lesz más, mint az egyes játékállások „jóságának” becslése az emberi játékos számára. Egy állás heurisztikus értéke minél nagyobb / kisebb lesz, annál kedvezőbb / rosszabb az adott állás az első játékos számára. Három heurisztikát fogunk létrehozni a keresőalgoritmusokhoz, egy *AKADALYOZO, egy NYERNIAKARO* és egy *VEGYES* heurisztikát.

A heurisztikák számítási módja:

- *AKADALYOZO*: Végigjárjuk az első játékos által kiválasztott rácspontokat, és mindegyiknél először megvizsgáljuk, hogy van-e olyan, az első játékos által kiválasztott másik csomópont, amellyel van olyan közös pont amelyik mindkét csomóponttól egy lólépésnyi pozícióban van, ha van akkor minden ilyen csomópont után eggyel növeli a heurisztika értékét. Ezek után megnézi, hogy a vizsgált csomópont összeköthető-e egy olyan ponttal, amelyik az első játékoshoz tartozó valamelyik alapvonalon van. Ha igen, akkor növeljük a heurisztika értékét öttel, ha az alapvonalon lévő ponttal van olyan közös csomópont, amelyik az aktuális csomóponttól és az alapvonalon lévőtől is lólépésnyi távolságra van, akkor az ilyen csomópontok számának háromszorosát adjuk a heurisztikához. Legvégül az első játékos vonalainak a számát megszorozzuk tízzel és hozzáadjuk a heurisztika értékéhez, ez lesz a végleges érték.
- *NYERNIAKARO*: Ugyanazokat számoljuk ki mint ez előbbi heurisztika esetén, csak az első játékos helyett most a második játékoshoz.
- *VEGYES*: Az előbbi két heurisztika keveréke: a heurisztika értéke az első heurisztika értékének és a második heurisztika értékének különbsége.

A heurisztikák játéktapasztalatok alapján lettek kitalálva és implementálva. Az akadályozó kizárólag az első játékos lépései alapján fogja meghatározni a heurisztika értékét, és mivel a második játékosnak akarunk lépést ajánlani, ezért minél magasabb lesz ennek az értéke, annál jobban fog járni az adott állásban az első játékos. Tehát amikor ezzel a heurisztikával számolunk, akkor a gép egyedül arra fog törekedni, hogy az emberi játékos minél rosszabbul járjunk, próbál majd minket megakadályozni a győzelem elérésében. Ő maga nem fog

törekedni rá, bár ha látni fogja hogy van esélye nyerni, akkor már nem fog minket akadályozni, hanem megnyeri a játékot. A nyerni akaró heurisztika ennek az ellentéte, a második, azaz a gépi játékos lépései alapján számolja ki a heurisztika értékét ki. Ez az érték minél magasabb, annál jobban fog járni a gép. Nem fog törődni az emberi játékos lépéseivel, megpróbál a lehető leggyorsabban nyerni és mindig a számára legkedvezőbb állásba jutni, még akkor is, ha ezzel az első játékosnak is esélye nyílik a nyerésre. A vegyes változat a kettő keveréke, hiszen mindkét játékos lépéseit figyelembe véve határozza meg a heurisztika értékét, így ez a legreálisabb és legjobban az emberi játékot tükröző heurisztika.

A függelékben megtalálhatók az egyes heurisztikák alapján végzett keresések végeredményei.

Most már három keresőalgoritmus és három különböző heurisztika van a birtokunkban. Ahhoz, hogy a gép ellen tudjon az emberi játékos játszani nincsen más dolgunk, mint az egy illetve kétszemélyes játék elindítása szerint az `SzdApp` osztályban található *`isOnePlayerGame`* és *`isTwoPlayerGame`* attribútumokat a megfelelő értékekre állítani. A játék folyamán ezeket felügyelve vagy hagyjuk az emberi játékosokat egymás ellen játszani, vagy az emberi játékos lépése után részleges felépítjük a játékfát és abból a megadott kereső és heurisztika segítségével egy lépést ajánlanunk és azt a lépést végrehajtjuk.

Ezzel a játék asztali verziójának a fejlesztése véget ért, most következhet ugyanennek a játéknak a webböngészőben futtatható és a világon bárhonnan és bárki ellen játszható verziójának elkészítése.

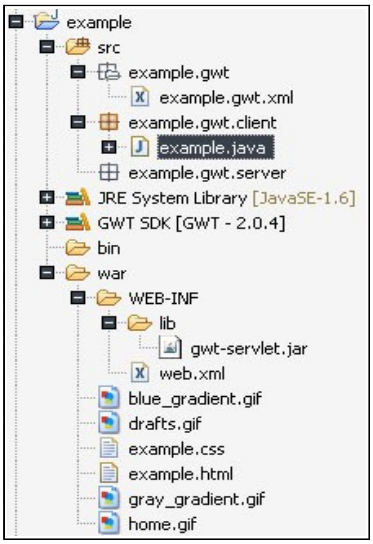
3.2 A játék online verziója

A fejlesztéshez továbbra is az Eclipse fejlesztői környezetet fogjuk használni, a legfontosabb eszközünk a nemsokára bemutatott Google Web Toolkit lesz, amiben egyszerre fogjuk a játék kliens és szerver oldalát is fejleszteni. A szerver oldalon valamilyen módon tárolnunk kell a felhasználóknak az adatait, ehhez az Apache Derby beágyazott adatbázisszervert fogjuk használni aminek segítségével fogjuk a JDBC-t elérni.

3.2.1 Google Web Toolkit

A Google Web Toolkit (GWT) a Google által létrehozott nyílt forráskódú fejlesztői környezet, melynek segítségével komplex böngészőben futtatható alkalmazásokat tudunk fejleszteni és optimalizálni. A GWT nagyban segíti a fejlesztést, mert a használatához nem szükséges a JavaScript nyelv ismerete, segítségével mégis könnyen tudunk AJAX alkalmazásokat fejleszteni Java nyelven, majd előállítani egy olyan optimalizált JavaScript kódot, amelynek futtatására minden böngésző képes különleges optimalizáció nélkül. Ez még nem is lenne annyira érdekes, viszont teljes kliens / szerver architektúrát, szolgáltat, és ami szinte a legfontosabb, egy olyan keretrendszert is kapunk, amely segítségével aszinkron hívásokat tudunk kezdeményezni a webalkalmazásunk és a szerver között. Mindezek mellé kapunk egy felhasználói felület elem készletet, amelyek minden böngészőben egységesen néznek ki és mégis flexibilisek, továbbá egy eseménykezelő rendszert, melynek segítségével könnyen tudjuk kezelni a kliensoldali eseményeket, és egy eszközt, amellyel állapotos böngésző előzményeket tudunk létrehozni és kezelni, amivel kezelhetővé válik az oly rettegett „Vissza” gomb a böngészőkben.

A fejlesztést még egyszerűbbé teszi, ha az Eclipse IDE-hez elérhető GWT bővítményt használjuk, amellyel a fejlesztés során felmerülő legfontosabb feladatokat tudjuk elvégezni. Ezen kívül hogy egyszerűen és hatékonyan tudunk szép GUI-t létrehozni a külön letölthető GWT Designer-t fogjuk használni, amely szintén bővítmény formában könnyen beszerezhető Eclipse-hez. Amikor létrehozunk egy új GWT projektek, a következőt fogjuk látni:



4. ábra : Egy újonnan létrehozott GWT projekt

Nézzük rendre hogy miket kaptunk:

- <*modulnév*>: Ebben a csomagban találunk egy <*modulnév*>.gwt.xml fájlt, amely a <*modulnév*> nevű modulhoz tartozó információkat tartalmaz. Egy modul újrafelhasználható funkciók gyűjteménye.
- <*modulnév*>/client: Itt lesznek a kliensoldali Java osztályaink, végső soron ezek lesznek JavaScript-re fordítva, amelyben nem használhatunk szabadon bármilyen beépített csomagot.
- <*modulnév*>/server: Itt lesznek a szerver oldalon használt osztályaink, amelyben szabadon használhatunk minden csomagot.

Még gyakran szükségünk lehet egy <*modulnév*>/shared csomagra, itt fogunk minden olyat osztályt tárolni, amelyre mind a kliens, mind a szerver oldalon szükség van. Ajánlott egy <*modulnév*>/public csomag, ahova statikus erőforrásainkat (HTML fájlok, CSS fájlok, képek stb…) érdemes elhelyezni. Elhelyezhetnénk ugyan ezeket a „war” mappába is, de a fordító modulonként külön mappát hoz létre. Így ha van egy <*modul1*> nevű modulunk és annak a public csomagjában egy *index.html*, akkor a nyitó oldalunkat elérhetjük a /index.html linkkel, míg a modulunkat a /modul1/index.html linken fogjuk elérni.

A gwt.xml állományban meg kell jelölni egy EntryPoint (belépési) osztály, amelynek az `onModuleLoad()` metódusa fog meghívódni az adott modul betöltődésekor. A legegyszerűbb példa (amit mi is csinálni fogunk), hogy csak egy modult fogunk létrehozni. Ha a HTML oldalunk nem tartalmaz semmi mást csak a lefordított JavaScript kódot, akkor az

`onModuleLoad()` metódus akkor fog meghívódni, amikor a böngészőben betöltődik az oldalunk.

A szervernek valamilyen módon kapcsolatot kell tartania a felhasználóval illetve valamilyen módon reagálnia kell a felhasználó egyes műveleteire. A GWT erre két megoldást is szolgáltat, az egyik egyszerű HTTP GET és POST utasítások révén küldi az információt, a másik pedig a GWT RPC keretrendszer, mely távoli eljáráshívásokkal (Remote Procedure Call vagy más néven szerverhívásokkal, alapja a Java Servlet technológia) éri el a szerver oldali erőforrásokat és információkat. Az RPC előnye hogy teljesen *aszinkron*, így egy AJAX alkalmazás minden előnyével rendelkezik. Tehát nem szükséges minden felhasználói interakció után egy új HTML oldalt betöltenünk, elég egyszerűen a felhasználói felületet a szükséges módon átalakítani a szerverrel való kommunikáció nélkül. Az GWT RPC használatához három dologra van szükség:

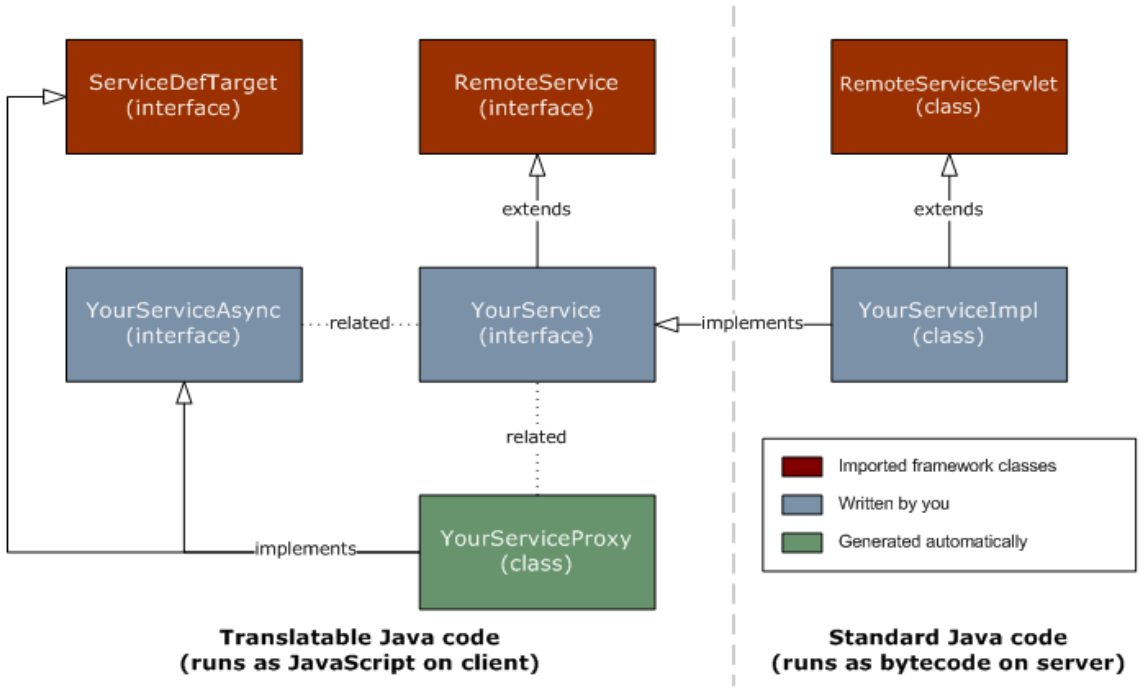
1. Definiálni kell egy a `RemoteService` interfészből kiterjesztett interfészt, ami magában foglalja az összes RPC metódusunkat.
2. Létre kell hozni egy a `RemoteServiceServlet` osztályból kiterjesztett osztályt, ami implementálja az előbb létrehozott interfészt.
3. Definiálni kell egy aszinkron interfészt a szerverhez a kliens oldalon.

Szerencsére nem kell mindenre odafigyelni, elég a <*modulnév*>/client csomagban létrehoznunk az 1. pont szerint megadott interfészt. Az aszinkron párja illetve a szerver oldali ezen interfészt implementáló osztály automatikusan létrejön. Ezen kívül még ahhoz, hogy tesztelni tudjuk majd a programunkat, a web.xml fájlhoz hozzá kell adni egy leírást, hogy a webszerveren hol lesz megtalálható az adott szolgáltatás, például:

```
<servlet>
      <servlet-name>Services</servlet-name>
      <servlet-class>???</servlet-class>
</servlet>
<servlet-mapping>
      <servlet-name>Services</servlet-name>
      <url-pattern>???</url-pattern>
</servlet-mapping>
```

A `servlet-class` elemben kell megadnunk a server csomagban az interfészt implementáló osztályt, az `url-pattern` elemben pedig meg kell adnunk a modulunknak a nevét és az

interfészben a `@RemoteServiceRelativePath("RelativePath")` annotációval megadott elérési utat (például `<url-pattern>/szakdolgozat/Services</url-pattern>`).



5. ábra : RPC-hez szükséges osztályok hierarchiája és kapcsolata[1]

A rendszerünkbe a következő funkciókat szeretnénk megvalósítani:

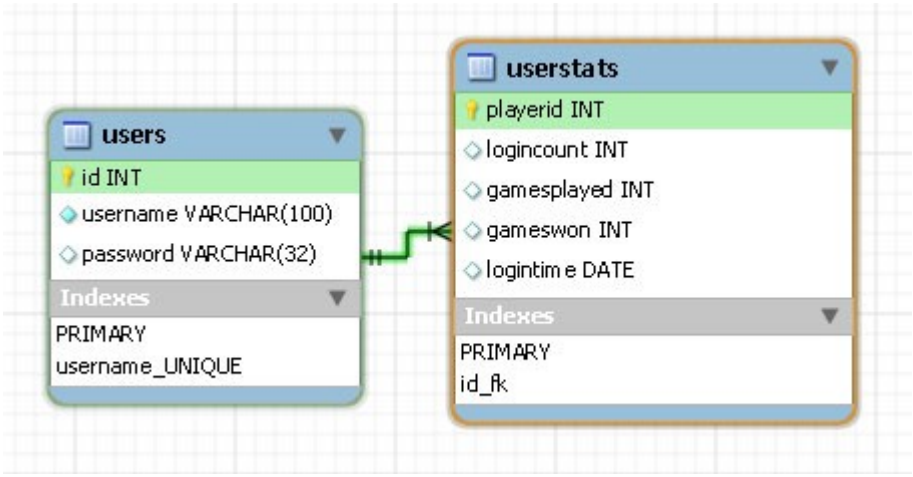
- Egy regisztrációs rendszert, amelyen keresztül bárki szabadon beregisztrálhat a rendszerbe. A beregisztrált játékosok adatait egy adatbázisban fogjuk eltárolni.
- Egy bejelentkező rendszert, ahol a regisztráció során megadott felhasználónévvel és jelszóval be lehet lépni a rendszerbe (bejelentkezés után lehetséges a játék).
- Valamilyen módon tudatni a felhasználóval, hogy kik az aktuálisan bejelentkezett felhasználók, akik ellen játszhat.
- Az egyes játékosokhoz tartozó statisztikák lekérdezésére valamilyen eszközt kell nyújtani.
- Magát a játékot is meg kell valósítani, hogy két játékos egymás ellen tudjon játszani.

1 Kép forrása: http://code.google.com/p/google-web-toolkit/source/browse/changes/jat/test-mult-iframe/doc/src/AnatomyOfServices.gif?r=9149

A felhasználói felületünkön két fő panelünk lesz, mindkettő `DecoratedTabPanel` lesz, aminek a segítségével füleket tudunk majd létrehozni és a különböző füleken fogunk különböző funkciókat ellátni. Az egyik panel fogja tartalmazni az egyik fülön a beléptető ablakot, a másik fülön pedig a regisztrációs ablakot. A második panel akkor fog betöltődni, ha sikeresen be tudtunk lépni, az első fülön fog megjelenni a játékosok listája illetve egyéb funkciók, a második fülön lesznek a bejelentkezett felhasználó eddigi elért eredményei és statisztikái. Ezen a panelen akármennyi fül lehet, mert amikor egy játékos egy másik játékos statisztikáira lesz kíváncsi, akkor egy ezen a panelen megjelenő új fülön fogjuk az adott játékos statisztikáit megjeleníteni. Mindegyik fül tartalmát külön osztályokba fogjuk létrehozni, így szép és átlátható kódot fogunk kapni.

3.2.2 Adatbázis

Szükségünk lesz egy adatbázisra hogy a beregisztrálásokat és a bejelentkezéseket kezelhetővé tegyük. Azért esett az Apache Derbyre a választásunk, mert a fejlesztői környezetünkhöz elérhető hozzá egy bővítmény, amivel nagyon könnyen kezelhetővé tudjuk tenni egy projekt adatbázis-kezelését, illetve mert támogatja a JDBC, JAVA és SQL szabványokat és a méretét tekintve pici (pár megabyte méretű az alapeszköz és a JDBC meghajtó). Az adatbázist természetesen szerver oldalon fogjuk létrehozni, két táblánk lesz, melyek közül az egyikben fogjuk tárolni a játékos regisztrációs adatait, míg a másikban a játékban elért eredményeit fogjuk nyilván tartani.



6 ábra : A felhasználói adatokat tároló adatbázis szerkezete

A users tábla oszlopai:

- *id* : A tábla elsődleges kulcsa, értéke minden egyes regisztráció során nő, ezzel fogjuk azonosítani a felhasználót.
- *username* : A regisztráció során megadott felhasználónév, egyedinek kell lennie.
- *password* : A regisztráció során megadott jelszó MD5 kódolás utáni eredménye.

A userstats tábla elemei:

- *playerid* : A tábla elsődleges kulcsa, külső kulcsként a users tábla id mezőjét hivatkozza.
- *logincount* : A felhasználó sikeres bejelentkezéseinek a száma, alapértéke 0.
- *gamesplayed* : A felhasználó által megkezdett játékok száma, alapértéke 0.
- *gameswon* : A felhasználó által megnyert játékok száma, alapértéke 0.
- *logintime* : A felhasználó utolsó bejelentkezésének dátuma.

3.2.3 Regisztráció

A regisztráció során ellenőrizni fogjuk, hogy az adatbázisunk szerint létezik-e már ilyen nevű felhasználó. Ha igen, akkor nem engedjük meg a regisztrációt, minden más esetben a users táblához hozzáadjuk az új felhasználót a megfelelő adatokkal. Hogy a felhasználónak a jelszava biztonságban legyen az adatbázisban, nem a valódi jelszót fogjuk eltárolni, hanem annak az MD5 kódolt képét, így még ha sikerülne is valakinek hozzáférnie az adatbázishoz nem tudná belőle kinyerni a felhasználók jelszavait. Hogy még biztonságosabb legyen a regisztráció és majdan a bejelentkezés is, már kliens oldalon lekódoljuk a jelszót és a kódolt verzióját fogjuk továbbítani majd a szervernek. A userstats táblában ezért is 32 hosszúságú a jelszómező, mert az MD5 kódolás után a jelszavakból egy 32 karakter hosszú szó lesz. A regisztráció során mindkét táblába egy-egy új sor fog kerülni a felhasználó által megadott adatok szerint. [1]

3.2.4 Bejelentkezés

A bejelentkezés során ellenőriznünk kell, hogy a felhasználó megfelelő felhasználónév / jelszó párost adott-e meg. A szerverre a felhasználónév és a már MD5 kódolású jelszó fog megérkezni, ezeket fogjuk összevetni az adatbázisban található adatokkal, és ha a jelszó és a

felhasználó név is egyezik, akkor engedélyezzük a belépést, máskülönben jeleznünk kell a felhasználó felé hogy valamilyen adatot rosszul adott meg. Ha sikeres a belépés akkor az adatbázist is egy picit módosítani kell, inkrementálni kell a bejelentkezett felhasználóhoz tartozó logincount értéket, és valamilyen módon tudatnunk kell a többi bejelentkezett felhasználóval a tényt, hogy bejelentkeztünk. Ezt megoldhatnánk úgy, hogy mindegyik kliens bizonyos időközönként lekéri a bejelentkezett felhasználók listáját, azonban ez a megoldás nagyon nem lenne optimális. Minden egyes híváskor egy TCP kapcsolatot kell felépíteni a szerver és kliens között (ún. háromutas kézfogás), amit létrehozni sok erőforrást és sávszélességet igényel. Ezért szeretnénk a szerverhívások számát minimalizálni és csak akkor kérni a szervertől információt, amikor az az információ létezik és szükségünk is van rá. A GWTEventService [2] keretrendszert fogjuk használni, melynek segítségével olyan (a Java AWT-re hasonlító) eseménykezelő rendszert tudunk majd létrehozni, ahol a kliensek fel fognak tudni iratkozni különböző szervereseményekre és az események bekövetkezéséről jelzést kapnak. A GWT önmagában csak a kliensoldali eseménykezelést tartalmazza (például egy gomb megnyomására fel lehet iratkozni), szerver oldali eseményekre nem lenne lehetőségünk figyelni (esetleg bizonyos időközönkénti szerverhívásokkal). Hogy használni tudjuk a keretrendszert, először a gwt.xml fájlhoz hozzá kell adni a következő sort:

```
<inherits name='de.novanic.eventservice.GWTEventService'/>
```

hogy egyrészt a kliens oldalon kezelni tudjuk majd a bekövetkező eseményeket, illetve meg kell változtatnunk a szerver oldali osztályunkat, hogy a `RemoteEventServiceServlet` osztályból származzon, hogy képesek legyünk a szerver oldalon eseményeket létrehozni. Ezek után a szerver a `addEvent(domain, event)` paranccsal fog tudni eseményeket létrehozni, ahol az adott domain-re feliratkozott kliensek fogják érzékelni az általunk létrehozott tetszőleges `event` eseményt. Egy eseményt az alábbi módon tudunk saját magunk deklarálni:

```
import de.novanic.eventservice.client.event.Event;
public class UserLoginEvent implements Event {}
```

Az eseményosztályunknak tetszőleges attribútumai lehetnek, azonban mindenképpen szükséges egy alapértelmezett, paraméterek nélküli konstruktort is létrehozni, mert különben nem fogjuk tudni az eseményt a kliens oldalon értelmezni! A kliens oldali eseményfeldolgozásra érdemes egy saját eseménykezelőt írni, amely így nézhet ki:

```
import
   de.novanic.eventservice.client.event.listener.RemoteEventListener;
public abstract class ServerEventsListener implements
                                                    RemoteEventListener {
      @Override
      public void apply(Event anEvent) {
            if (anEvent instanceof UserLoginEvent) {
                  onUserLogin((UserLoginEvent) anEvent);
            }
      }

      public abstract void onUserLogin(UserLoginEvent event);
}
```

majd az eseményekre az alábbi módon tudnak feliratkozni a kliensek:

```
RemoteEventServiceFactory theEventServiceFactory =
                                    RemoteEventServiceFactory.getInstance();
theEventService = theEventServiceFactory.getRemoteEventService();

theEventService.addListener(GAME_DOMAIN, new ServerEventsListener()
                                                                        {});
```

Természetesen a kliensek csak azokról az eseményekről fognak értesülni, amelyek a megadott `GAME_DOMAIN` domainen érkeznek. A Listener hozzáadása során a saját eseménykezelő osztályunk megfelelő absztrakt metódusait felül kell írni a tényleges tevékenységekkel, amit az esemény bekövetkeztekor végre szeretnénk hajtani.

Utolsó tennivalónk a bejelentkezési felületünkön a jelszómezőhöz egy `KeyPressHandler-t` (amely az Enter leütésre figyel), illetve a bejelentkezés gombhoz egy `ClickHandler`-t hozzáadni (amely a gombra kattintására figyel). Valamelyik esemény bekövetkezése esetén megkísérelünk bejelentkezni az alábbi módon:

```
isUserPasswordValid(user, new AsyncCallback<Integer>() {

            @Override
            public void onSuccess(Integer result) {}

            @Override
            public void onFailure(Throwable caught) {}
      });
```

Az előbb használt metódust a Services osztályban van definiálva, amelynek paraméterként a `User` osztály egy példányát kell neki átadni, ami az aktuális játékos felhasználónevét és jelszavát tartalmazza, a szerver aszinkron válaszát pedig a `new AsyncCallback<T>()` két ágában tudjuk lekezelni, ahol a <T> generikus paraméter mindig a

szervertől kapott válasz típusát jelöli, ami a mi esetünkben most `Integer`.

A kliens és a szerver között néhány feltétel betartása mellett könnyen tudunk objektumokat küldeni. Ahhoz, hogy objektumokat tudjunk átküldeni egy hálózati kapcsolaton keresztül, mindenekelőtt szükséges, hogy az objektum szerializálható legyen (egy szerializált objektum nem más, mint az adott objektum bináris formája, amit a fogadó oldalon deszerializálással visszakaphatunk). Viszont mivel itt JavaScript és Java kóddal is dolgozni kell, ezért a GWT megteszi helyettünk az objektumok szerializációját. Így nem elég a sima szerializálhatóság követelményeinek eleget tenni, mivel a GWT szerializáció nem egyezik a sima szerializációval. Az alábbiaknak kell hogy teljesüljenek ahhoz, hogy egy a felhasználó által definiált osztályt is át tudjunk küldeni RPC-n keresztül:

- Az `com.google.gwt.user.client.rpc.IsSerializable` vagy a `java.io.Serializable` interfészek valamelyikét implementálnia kell az osztálynak
- Az osztály minden attribútuma szerializálható és egyik sem **`final`** vagy **`transient`**.
- Az osztálynak van egy publikus alapértelmezett paraméterek nélküli konstruktora.

A GWTEventService kapcsán már említettük, hogy egy-egy saját eseményhez mindenképpen hozzá kell adni egy üres konstruktort is, különben gondok lehetnek az esemény létrehozása vagy fogadása során. Mivel az események elküldése szintén RPC alapú, így ahhoz, hogy át tudjunk küldeni a szervertől a kliensnek egy eseményt, annak a fenti feltételek közül mindegyiknek eleget kell tennie.

Visszatérve az aszinkron hívásunk eredményére, annak sikerességétől függően két metódus hívódhat meg. Ha a szerveren nem következett be semmilyen váratlan hiba és a kérésünk sikeresen lefutott, akkor az `AsyncCallback<T>` interfészben definiált **`void`** `onSuccess(T result)` metódus fog lefutni, a T típus pedig a Services osztályunkban az adott szerverhíváshoz megadott metódus visszatérési típusa lesz. Ha valamilyen oknál fogva a szerverünk nem tudta feldolgozni a kérésünket, akkor a **`public void`** `onFailure(Throwable caught)` rész fog lefutni, és a visszatérési érték a szerveren bekövetkezett kivétel lesz.

A bejelentkezés, regisztráció, játékok indítása és játékok befejezése után minden alkalommal módosítanunk kell az adatbázist. Az adatbázishoz való hozzáféréshez két lehetőség kínálkozik: vagy elrejtjük a felhasználó elől az SQL utasításokat, vagy nem. Az első

megoldás során használunk kell valamilyen komolyabb objektum-relációs leképező keretrendszert (pl. Hibernate). Jelen esetben olyan keveset fogunk az adatbázissal dolgozni, hogy a Hibernate használata csak megbonyolítaná a fejlesztést, nem lesz rá szükség. Ehelyett a függelékben megtalálható segédosztályt fogjuk használni, amely egy lekérdezés ResultSet eredményét alakítja át egy adott paraméterként kapott osztály példányainak listájává.

A szerver oldalon bejelentkezés során az adatbázisból SQL lekérdezéssel lekérjük a kapott felhasználónévhez tartozó adatokat, majd a segédosztályunkkal használatával azt a `User` osztály egy példányává konvertáljuk. A kapott és adatbázisbeli felhasználónevet és jelszavat összevetjük. Ha a kettő megegyezik, akkor válaszként elküldjük az adatbázisból kinyert felhasználóhoz tartozó id értéket, valamint létrehozunk egy `UserLoginEvent` eseményt hogy az összes felhasználó tudjon a bejelentkezésről, amit a bejelentkezett felhasználók megkapnak. Aki éppen bejelentkezik, annak is látnia kell a már bejelentkezett játékosokat, így a belépés után a szervertől lekérjük ezen felhasználók listáját egyszer és eltároljuk. Így később csak növelni / csökkenteni kell majd mindegyik kliensoldalnak a saját felhasználói listáját, és nem kell minden egyes alkalommal a szervertől lekérnie a teljes listát. A kijelentkezés során is hasonló folyamat megy végbe, csak akkor egy `UserLogoutEvent` eseményt fogunk körbeküldeni a felhasználók között, és kitöröljük a felhasználót a bejelentkezettek listájáról. A kijelentkezett felhasználó elől eltüntetjük a bejelentkezés után elérhető felületet és a bejelentkező ablakot fogjuk mutatni neki.

3.2.5 Főmenü

Bejelentkezés után valamilyen módon a felhasználó tudtára kell adni a többi, jelenleg a rendszerbe bejelentkezett felhasználó listáját és a hozzájuk tartozó adatokat. A felhasználók megjelenítésére használhatnánk az alap GWT-ben megtalálható eszközöket (például egy `FlexTable` táblázat egyes mezőibe illeszthetnénk be az egyes játékosokhoz tartozó adatokat). Viszont ha esztétikus megjelenítést szeretnénk biztosítani az adatokhoz, akkor érdemes a SmartGWT-ben lévő `ListGrid` osztályt használni. Ennek a listának könnyen lehet az egyes oszlopainak illetve mezőinek a típusait / stílusait változtatni. Egyetlen hátrány, hogy a használatához még létre kell hoznunk olyan, a `ListGridRecord` osztályt kiterjesztő osztályt, melynek nem lesz attribútuma, viszont a listában szereplő minden oszlophoz kell írni bele egy beállító / lekérdező metóduspárt az alábbiaknak megfelelően:

```
public void setUserName(String username) {

	setAttribute("név", username);
}

public String getUserName() {
	return getAttributeAsString("név");

}
```

ahol a `"név"` a lista adott oszlopának a neve. Az összes primitív típushoz rendelkezésre áll a megfelelő `getAttributeAsXXX()` metódus.

7. ábra : A bejelentkezés utáni főablak

Játékot úgy fogunk tudni kezdeményezni, hogy kiválasztjuk a listából azt, akivel játszani szeretnénk és utána a „Játék indítása” gombra kattintunk. A felkérőnek egy várakozóablakot mutatunk, a felkért játékosnak pedig egy játékot kezdeményező ablakot, ahol megtekintheti a felkérő statisztikáit és lehetősége van elfogadni vagy elutasítani a felkérést. Ha elutasítja, akkor a felkérőt értesítjük az elutasításról, ha elfogadja, akkor elindítunk egy játékot köztük. Előfordulhat, hogy akit felkértek, valamilyen oknál fogva nem tud válaszolni a felkérésre hosszú ideig. Ezért a felkért játékosnak harminc másodperce van elfogadni a játékkezdeményezést, ha ennyi idő alatt nem válaszol, akkor értesítjük a felhasználót, hogy nem érkezett időben a felkérésére válasz. Egy játékos statisztikáinak megtekintéséhez a játékos listából való kiválasztása után a „Játékos statisztikái” gomb lenyomásával egy új fülön van lehetőségünk, amely tartalmazza az adott játékosra vonatkozó összes fontos információt. Ha nem szeretnénk, hogy játékra vonatkozó felkérés érkezzen hozzánk, hanem mi szeretnénk kiválasztani az ellenfelünket, akkor lehetőség van a „Felkérések automatikus elutasítása” mezőt kipipálni. Így anélkül kapja meg az elutasítást a felkérő felhasználó, hogy megjelenne a felkért játékosnál a játékkezdeményező ablak. A saját statisztikák fülön a bejelentkezésünkkor érvényes adataink fognak megjelenni. Ha kíváncsiak vagyunk, hogy mennyit sikerült az átlagainkon javítani egy bejelentkezés során, akkor ha kiválasztjuk a felhasználók közül a saját nevünket, és ugyan úgy ahogyan minden más felhasználó esetén lekérjük a legfrissebb statisztikákat egy új fülre, akkor össze tudjuk hasonlítani a két fülön található eredményeket. Természetesen amikor mi vagyunk kiválasztva a listáról, akkor a játékindító gomb inaktív, hiszen csak mások ellen tudunk játszani, egyszemélyes játékra nincsen lehetőség. Hogy ki van kiválasztva a bejelentkezett felhasználók közül azt a két gomb között megjelenő feliraton fogjuk látni.

3.2.6 Játékablak

A játékablak nem fog sokban különbözni az asztali verzió játékpanelétől, a pontok ugyanúgy rádiógombok lesznek, a tábla mérete ugyanakkora és a játékosok színe és alapvonalaiknak elhelyezkedése is megegyezik. A lényeges változtatás az, hogy a tábla alján elhelyezkedő `TextArea` és `TextBox` segítségével a játékosok beszélgethetnek egymással. A szövegdoboza beírható a másik félnek szánt üzenet, ami az Enter leütése után kerül elküldésre. Az üzenet pedig mindkét felhasználónál megjelenik a felületen. A szövegek

küldését a GWTEventService keretrendszer segítségével fogjuk elküldeni egyik játékostól a másiknak, és ezzel a megoldással fogjuk a játékosok lépéseit is elküldeni az ellenfeleknek. Az egyetlen különbség az, hogy erre a két eseményre egy külön osztályt fogunk létrehozni az alábbi módon:

```
public abstract class GameEventsListener implements
                                                    RemoteEventListener {

        @Override
        public void apply(Event anEvent) {
              if (anEvent instanceof MoveEvent)
                    onMoveEvent((MoveEvent) anEvent);
              else if (anEvent instanceof MessageEvent)
                    onMessageEvent((MessageEvent) anEvent);
        }

        public abstract void onMessageEvent(MessageEvent event);
        public abstract void onMoveEvent(MoveEvent event);
}
```

A `MessageEvent` esemény egy String objektumot fog tartalmazni, ami nem lesz más, mint az egyik játékos által az ellenfélnek címzett üzenet. A `MoveEvent` pedig egy sor és oszlopindexet fog tartalmazni, amelyek az egyik játékos által kiválasztott pont sorának és oszlopának értékei lesznek. Végül a két játszó fél egy olyan domain-re fog feliratkozni, melynek alakja `játékosnév + "-gamelistener"` lesz, az üzeneteket, és a lépéseket pedig a `ellenfélnév + "-gamelistener"` domain-re fogja küldeni a `public void sendMoveToOpponent(User opponent, int row, int col)` és `public void sendMessageToGamePartner(User partner, String message)` szerverhívással, így amikor lép vagy üzenetet küld akkor azt csak az aktuális ellenfél látja, senki más. Amikor egy játékra felkérést küldünk, akkor is igaz, hogy csak a megfelelő kliensnél fog megjelenni a kezdeményező ablak (akit felkértek). De ilyenkor minden bejelentkezett és a mindenki által feliratkozott (akár nevezhetnénk „broadcast” domainnek is) domainen fog megjelenni az esemény, és a kliens oldalon fogjuk lekezelni hogy csak annál a játékosnál jelenjen meg az ablak akit ténylegesen felkértek egy játékra. Ha ugyanígy tennénk a lépéseknél és üzeneteknél is, akkor az túl nagy hálózati terheléssel járna a már ismertetett „háromutas” kézfogás felépítésének erőforrásigénye miatt, valamint amiatt, hogy feleslegesen küldenénk el a lépést / üzenetet sok felhasználó számára.

Ahhoz persze, hogy egyáltalán lépni tudjunk a játékban, elő kell vennünk a játék

állapottérreprezentációját és kis módosításokkal újra kell implementálni. Két lehetőségünk van: tehetnénk úgy is hogy a szerveren „fut” minden játék, és a klienseknek csak a grafikus felületet mutatjuk, és ezek eseményeire reagálva a szerveren megváltoztatjuk a szerveren lévő adott játszmához tartozó állapotot, vagy a kliens oldalon kezelhetnünk mindent és a szervernek ekkor csak az üzenetek és információk tárolása és továbbításába lenne a feladata. Az első lehetőségnél a szerveroldali terhelés túl nagy lenne, ami miatt korlátokat kellene szabnunk például a párhuzamosan futható játékok maximális számára, így egy ún. vékony klienst kapnánk. A második lehetőségnél a kliens oldalon képesek lennénk mindent feldolgozni a szerver oldali erőforrások használata nélkül, ez egy „vastag” klienst eredményezne. Mi a vastag kliens mellett fogunk dönteni, hiszen az AJAX alkalmazások túlnyomó többsége ilyen. Tehát mindegyik kliens nyilván fogja tartani az aktuális játékot, és az egyes böngészőablakban bekövetkezett eseményeket kliensoldalon fogjuk feldolgozni. A szerver feladata kizárólag a felhasználók közötti kapcsolattartás lesz. Mivel kliens oldalon olyan kódot kell létrehozni, amit a GWT fordító JavaScript kóddá tud alakítani, ezért lesz szükség az osztály minimális átírására.

- `public` User opponent;

  Az ellenfél játékos, amelyet azért szükséges eltárolnunk, hogy tudjuk, hogy kinek kell az üzeneteket és lépéseket elküldeni.

- `private int currentPlayer;`

  A lépni következő játékos kódja.

- `private Set<Point> selectedNodes;`

  A kiválasztott pontok halmaza.

- `private ArrayList<Line> greenPlayerLines;`

  Az első / zöld játékos által húzott vonalak listája.

- `private ArrayList<Line> bluePlayerLines;`

  A második / kék játékos által húzott vonalak listája.

- `private int[][] gameTable;`

  A játéktábla, dimenzióinak mérete a *`TABLE_SIZE`* konstans értéke lesz.

A metódusok között ugyanúgy megtalálható a célállapotot, operátor alkalmazási előfeltételt és operátor hatását megvalósító, ahogy a játék asztali verziójában is, egyedül a heurisztika marad ki, mivel most nem lesz rá szükségünk. Hogy könnyen kezelhetőek legyenek a pontok és az egyenesek létre kell hoznunk két segédosztályt, mert nem használhatjuk a `java.awt.geom.Point2D` és `java.awt.geom.Line2D` osztályokat, ahogy azt az asztali verziónál tettük (hiszen most kliens oldali kódot kell írunk és a GWT fordító nem támogatja a java.awt csomagot). Amikor az éppen soron következő felhasználó lép, akkor először elküldjük az ellenfélnek a szerver segítségével a kiválasztott gomb sor és oszlop indexét, utána a kiválasztott gombon végrehajtjuk a megfelelő grafikai változtatásokat. Végül megváltoztatjuk a játék állapotát, és ha az új állapot bővült egy vagy több vonallal, akkor azokat kirajzoljuk. Ahhoz, hogy egyenest tudjunk rajzolni szintén a SmartGWT-re lesz szükségünk, mert az alap GWT nem nyújt ehhez megfelelő eszközöket. Ahhoz, hogy a gombok között vonalakat tudjunk húzni, először szükségünk lesz egy `com.smartgwt.client.widgets.Canvas` alapvászonra, amelyre rárakjuk az alap GWT-ben lévő `com.google.gwt.widgetideas.graphics.client.GWTCanvas` objektumot. Ezen a vásznon fogjuk kirajzolni megfelelő színekkel a vonalakat négyzeteket a rádiógombok alá, továbbá szükségünk van még egy `com.smartgwt.client.widgets.WidgetCanvas` vászonra, amihez hozzárendeljük a gombjainkat tartalmazó `FlexTable` táblázatot, majd az alap vászonhoz hozzáadjuk ezt a vásznat is. Azért kell ezt ilyen bonyolultan megoldani, mert ha az alap GWT-s vászont használnánk, akkor bár tudnánk vonalakat húzni, viszont a vászonra nem lehetne rátenni a rádiógombokat Ezt kiküszöböli a SmartGWT-s verzió, amely engedi, hogy a vászonra bármilyen más elemet rátegyünk. Viszont ebben az esetben kénytelenek lennénk mindegyik rádiógomb referenciáját nyilvántartani, nem lenne módunk együttesen kezelni őket. Ha viszont a `WidgetCanvas` objektumot használjuk az alap vásznon belül, az olyan, mintha AWT-ben vagy Swing-ben panelekkel dolgoznánk. Ha erre a vásznunkra ráhelyezzük a gombokat tartalmazó táblázatot és ezt a vásznat rátesszük az alap vászonra, akkor vonalakat is fogunk tudni húzni, és a gombjainkat is könnyen fogjuk tudni kezelni. Most már minden eszköz a kezünkben van, nincs más dolgunk, mint

- a GWT Designer segítségével esztétikusan összerakni mindent,
- implementálni a szerver oldalon az összes olyan metódust, amit a kliensek meghívhatnak,

- figyelni hogy a szerveroldali eseményeink mindig a jó domaineken jelenjenek meg,
- a különböző adatbázis-műveletek során ne forduljon elő hiba és a különböző SQL injekciós támadások megelőzése végett mindenhol `PreparedStatement`-t használjunk.

Még egy dolgot fogunk megoldani, a sok fejlesztő rémálmaiban megjelenő „Vissza” böngésző gomb kezelését, amely a GWT használatával könnyen megvalósítható a `com.google.gwt.user.client.History` csomag segítségével. Nincs más dolgunk, mint hogy az `EntryPoint` osztályban a `ValueChangeHandler` interfészt implementálni, majd az osztályon belül meghívni a `History.addValueChangeHandler(this)` parancsot. Ezek után már kezelni fogjuk tudni a vissza gomb hatását oly módon, hogy a paranccsal állapotos objektumként el tudjuk tárolni az adott oldalakat egy láncolt listában, így a visszalépés gomb hatására be tudjuk tölteni a listában legutoljára szereplő oldalt tartalmakkal együtt. A lista hibátlan működéséhez el kell helyezni benne egy kezdőelemet:

```
if (History.getToken().isEmpty()) {
      History.newItem("login");
 else {
      changePage(History.getToken());
}
```

Amikor lefut a `History.newItem("login")` parancs, akkor a böngészőben a címlistában a *http://www.weboldal.hu/index.html*`#login` cím fog szerepelni, és az implementált interfésszel érkező `public void onValueChange(ValueChangeEvent event)` felülírásával tudjuk majd kezelni, ha az URI-ban az erőforrás-azonosító (# utáni rész) megváltozik. Erre egy ehhez hasonló metódust célszerű használni:

```
public void changePage(String token) {
      if (History.getToken().equals("login")) {
      } else if (History.getToken().equals("main")) {
      }
}
```

3.2.7 Deploy

Miután elkészültünk a játékkal szeretnénk azt meg is osztani a nagyvilággal valamilyen módon. A fejlesztés során ún. „Hosted Mode”-ban dolgoztunk, ami azt jelentette, hogy egy ideiglenesen elindított Jetty webszerver segítségével jelenítettük meg a webböngészőben az

alkalmazást, most viszont keresünk kell egy olyan helyet, ami mindenki számára elérhető. Egyik lehetőségünk a Google Apps Engine (GAE) lehetne, ide egy gyors regisztráció után pillanatok alatt feltölthetnénk a projektünket, miután a GWT fordítóval létrehoztuk a megfelelő JavaScript fájljainkat. Sajnos azonban a GAE nem engedélyezi a JDBC-t. Megoldásként átírhatnánk a programot úgy, hogy ne legyen szükség JDBC-re, vagy szerezhetünk egy saját webszervert, amire minden további nélkül feltehetnénk a játékot. Mi egy Tomcat szerverre fogjuk feltölteni a játékot, melynek menete:

1. Tomcat installáción belül a webapps mappában létrehozunk egy mappát a projektünk nevével.
2. Az előbb létrehozott mappán belül létre kell hoznunk egy WEB-INF nevű mappát.
3. Az első lépésben létrehozott mappánkba másoljuk be az összes GWT által előállított JavaScript oldalt és a sima Java forrásállományt is.
4. A webapps/projektnév/WEB-INF/classes mappába bemásoljuk az összes bájtkódú Java fájlunkat, vagy előállítunk belőlük egy JAR fájlt és azt a webapps/projektnév/WEB-INF/lib mappába helyezzük el.
5. Végül a projektünkben található web.xml állományból az összes összetartozó `servlet` és `servlet-mapping` elempárt át kell másolnunk a Tomcat web.xml fájljába.

4. Összefoglalás

Sikeresen megalkottunk egy játék két működő és használható variánsát. Az ember-ember játékon volt a fejlesztés során inkább a hangsúly, hiszen ha csak a játékot szerettük volna megalkotni akkor nem kellett volna a webes részt fejlesztenünk, megelégedhettünk volna az asztali verzióval. Ez viszont sajnos az ember-gép játék rovására ment, nem sikerült egy, az emberi játékmódot teljes mértékben tükröző ellenfelet megalkotnunk. Ez nem a heurisztikák hibája, hanem a játék bonyolultságához köthető. Ahogyan már korábban is említettük ahhoz, hogy egy több mint tizenhét milliárd elemet tartalmazó fával dolgozzunk fejlettebb és kiforrottabb keresőalgoritmusokra lett volna szükségünk. Esetleg másik megoldásként használhattunk volna neurális hálókat, azonban akkor a fejlesztése túl sok időt emésztett volna fel. Az általunk használt eszközök mindegyike kiválóan teljesített, segítségükkel nagyon hatékonyan tudtuk az alkalmazásunkat fejleszteni, egyedüli

hiányosságként az Eclipse IDE-ből egy, a NetBeanshez hasonló UI fejlesztő felület hiányzott, de ezt a problémát könnyen át tudtuk hidalni. A GWT használata tökéletesen bevált, ténylegesen sikerült egy nagyon minimális JavaScript tudással egy olyan komplex és összetett oldalt megalkotnunk, amire egyébként semmilyen más lehetőségünk nem lett volna. A feladat során sok olyan eszközt is használnunk, ami az alap GWT-nek nem része, de sok fejlesztő készít segéd keretrendszereket hozzá. Így szinte nincs is olyan probléma amire nem lehetne megoldást találni.

Remélem sikerült egy picit felkelteni az érdeklődést az olvasóban akár a játék, akár a GWT használata vagy a mesterséges intelligencia iránt, és jó szórakozást és jó időtöltést kívánok a játék használatához!

Függelékek I.

A keresőalgoritmusok különböző táblaméret és heurisztikák melletti eredményei:

Táblaméret = 5 Mélység = 4	Minimax	Alfa-béta	Heurisztikus Minimax
Akadályozó Bejárt elemek Átlag Futási idő	85460 58215 csomópont/s 1468 ms	9719 38876 csomópont/s 250 ms	3702 7404 csomópont/s 500 ms
Nyerni akaró Bejárt elemek Átlag Futási idő	85460 89674 csomópont/s 953 ms	6304 40152 csomópont/s 157 ms	31857 32910 csomópont/s 968 ms
Vegyes Bejárt elemek Átlag Futási idő	85460 37205 csomópont/s 2297 ms	6830 25676 csomópont/s 266 ms	16969 8103 csomópont/s 2094 ms

Táblaméret = 7 Mélység = 4	Minimax	Alfa-béta	Heurisztikus Minimax
Akadályozó Bejárt elemek Átlag Futási idő	1877832 56689 csomópont/s 33125 ms	64699 51105 csomópont/s 1266 ms	55092 3533 csomópont/s 15593 ms
Nyerni akaró Bejárt elemek Átlag Futási idő	1877832 95915 csomópont/s 19578 ms	9711 56789 csomópont/s 171 ms	1110304 40769 csomópont/s 27234 ms
Vegyes Bejárt elemek Átlag Futási idő	1877832 33495 csomópont/s 56063 ms	19789 33314 csomópont/s 594 ms	160578 5324 csomópont/s 30157 ms

Ezek az adatok (futási idő és sebesség) nem szükségszerűen egyeznek meg, ha valaki másik számítógépen vagy más körülmények között teszteli az algoritmusokat, csak viszonyítási alapként használjuk az itt kapott eredményeket. Látható, hogy a minimax algoritmus minden esetben kielemzi a teljes játékfát, viszont mivel nem végez sok műveletet az egyes csomópontokban, nagyon gyorsan tud a játékfában haladni. Viszont mivel minden

csomópontot kielemez, mégis nagyon sok ideig tart a lépésajánlás. A sebességét igazából tehát csak az határozza meg, hogy a levélelemeknél milyen sokat kell számolni a heurisztikák megkapásához. A nyerni akaró és akadályozó heurisztikák esetén elvileg ugyanannyit kell számolni, mégis valamiért az akadályozó heurisztika használatakor gyorsabban halad a keresés, mint a nyerni akaró esetén, ami előtt csodálkozva állunk. A vegyes heurisztikánál kétszer többet kell számolni, mint a másik kettő során, így teljesen jogos, hogy ilyenkor a keresés valamivel lassabb, mint a másik két esetben, körülbelül a másik két keresés keresési sebességének az átlagának a fele (ez a heurisztikus minimaxra nem lesz igaz). Az alfa-béta kereső lassabban fog futni, mint a minimax, hiszen egy csomópontban több műveletet és számolást végez el, viszont ezek a számítások drasztikusan le tudják csökkenteni a játékfa méretét, amiért mégis gyorsabban fog egy ugyanolyan jó eredményre jutni, mint a minimax. Látszik is, hogy a bejárt csomópontok száma még a legrosszabb esetben sem éri el a minimax által bejárt elemek számának a tizedét, tehát a fának nagyon nagy részét sikerült mindegyik keresés során elhagyni, így a futási idő is alig tizede a minimax futási idejének. A bejárt elemek számának változása pedig a különböző heurisztikák miatt van, más heurisztikák mellett nem biztos, hogy a játékfának mindig ugyanazt a részét és ugyanannyi elemét járjuk be. A heurisztikus minimax esetén a sebesség drasztikusan lecsökkent a másik két keresőhöz képest, hiszen itt nagyon sok elemnek kell a heurisztikáját kiszámítani, ami nagyon lelassítja a keresőt, viszont ha jó a heurisztika akkor még az alfa-béta algoritmusnál is jobban le tudjuk csökkenteni a játékfa méretét. A táblázat alapján úgy tűnik, hogy a legjobb eredményeket az akadályozó heurisztika során éri el a kereső, és bár nagyon kevés elemet jár ilyenkor, de viszont a megnövekedett számítások miatt mégis lassabb lesz a keresés összességében, mint az alfa-béta vágás során. Vegyes heurisztika használatakor már több elemet fog bejárni, mint az alfa-béta kereső, a nyerni akaró heurisztika esetén pedig az eredményt mondhatnánk akár katasztrofálisnak is, a kereső még a minimax keresőnél is lassabban ad eredményt, amely ráadásul nem is biztos hogy olyan jó, mint a másik két algoritmussal kapott eredmény.

A fenti táblázat függvényében azt mondhatjuk, hogy a három algoritmus közül az alfa-béta teljesített (ahogyan vártuk is) a legjobban, a heurisztikák közül pedig a nyerni akaró került ki győztesként.

Függelékek II.

Minimax kereső kódja:

```
protected double createGameTree(int depth, Csucs actRoot) {

	if (actRoot.getDepth() == maximumDepth
			|| actRoot.getGameState().isGameEndState() == true) {
		actRoot.heurisztika = actRoot.getGameState().heurisztika();
		return actRoot.heurisztika;
	}
	double max = Integer.MIN_VALUE;
	Csucs child = null;

	for (SelectNodeOperator op : actRoot.getUsableOperators()) {
		child = new Csucs(actRoot, op);
		max = Math.max(max, -(createGameTree(depth + 1, child)));
		actRoot.heurisztika = max;
	}
	return max;
}
```

Alfa-béta kódja: [3]

```
protected double createGameTree(int depth, Csucs actRoot,
                                        double alfa, double beta) {
	if (actRoot.getGameState().getCurrentPlayer() ==
		AbstractGame.SECONDPLAYER) {
			return maxValue(actRoot, alfa, beta);
		} else
			return minValue(actRoot, alfa, beta);
	}
}

protected double maxValue(Csucs csucs, double alfa, double beta) {

	double v = Integer.MIN_VALUE;
	if (csucs.gameState.isGameEndState() == true
			|| csucs.depth == maximumDepth) {
		csucs.heurisztika = csucs.gameState.heurisztika();
		return csucs.heurisztika;
	} else {
		for (SelectNodeOperator op : csucs.getUsableOperators()) {
			Csucs child = new Csucs(csucs, op);
			double minimumValueOfSuccessor = minValue(child,
                                                        beta,alfa);
			if (minimumValueOfSuccessor > v)
				v = minimumValueOfSuccessor;
			if (v >= beta)
				return v;
			alfa = Math.max(alfa, v);
		}
		csucs.heurisztika = v;
		return v;
```

```
        }
}
public double minValue(Csucs csucs, double alfa, double beta) {

        double v = Integer.MAX_VALUE;
        if (csucs.gameState.isGameEndState() == true
                    || csucs.depth == maximumDepth) {
              csucs.heurisztika = csucs.gameState.heurisztika();
              return csucs.heurisztika;
        } else {
              for (SelectNodeOperator op : csucs.getUsableOperators()){
                    Csucs child = new Csucs(csucs, op);
                    double maximumValueOfSuccessor = maxValue(child,
                                                            beta, alfa);
                    if (maximumValueOfSuccessor < v)
                          v = maximumValueOfSuccessor;
                    if (v <= beta)
                          return v;
                    beta = Math.min(beta, v);
              }
        csucs.heurisztika = v;
        return v;
        }
}
```

Heurisztikus Minimax kódja:

```
protected double createGameTree(int depth, Csucs actRoot) {

        if (actRoot.getDepth() == maximumDepth
                    || actRoot.getGameState().isGameEndState() == true) {
              actRoot.heurisztika = actRoot.getGameState().heurisztika();
              return actRoot.heurisztika;
        }
        double max = Integer.MIN_VALUE;
        Csucs child = null;
        if (actRoot.getGameState().getCurrentPlayer() ==
                                                AbstractGame.FIRSTPLAYER) {
              double heurisztika = Integer.MAX_VALUE;
              for (SelectNodeOperator op : actRoot.getUsableOperators()) {
                    Csucs cs = new Csucs(actRoot, op);
                    cs.heurisztika = cs.getGameState().heurisztika();
                    if (cs.heurisztika <= heurisztika) {
                          actRoot.children.add(cs);
                          heurisztika = cs.heurisztika;
                    }
              }
        } else if (actRoot.getGameState().getCurrentPlayer() ==
                                                AbstractGame.SECONDPLAYER) {
              double heurisztika = Integer.MIN_VALUE;
              for (SelectNodeOperator op : actRoot.getUsableOperators()) {
                    Csucs cs = new Csucs(actRoot, op);
                    cs.heurisztika = cs.getGameState().heurisztika();
                    if (cs.heurisztika >= heurisztika) {
                          actRoot.children.add(cs);
```

```
                        heurisztika = cs.heurisztika;
                    }
                }
            }
            for (Csucs cs : actRoot.children) {
                if (cs.heurisztika == heurisztika) {
                    child = cs;
                    max = Math.max(max, -(createGameTree(depth + 1, child)));
                    actRoot.heurisztika = max;
                }
            }
            return max;
        }
```

Bejelentkezés a webes verzióban:

LoginWindow.java (részlet)

```
protected void loginUser(String username, String password) {
        final User user = new User();
        user.setUsername(username);
        user.setPassword(MD5.getMD5(password));
        user.setLoginDate(new Date());

        UI.servicesHandler.isUserPasswordValid(user,new
                                                AsyncCallback<Integer>() {

            @Override
            public void onFailure(Throwable caught) {
                errorLabel.setText("Sikertelen bejelentkezés! Rossz
                                                felhasználónév / jelszó.");
                userNameTextBox.setText("");
                passwordTextBox.setText("");
            }

            @Override
            public void onSuccess(Integer result) {
                if (result >= 0) {
                    user.setId(result);
                    user.setPassword("&");

                    UI.currentUser = user;
                    History.newItem("main");

                } else if (result == -2) {
                    errorLabel.setText("Sikertelen bejelentkezés!
                                Felhasználó már be van jelentkezve.");
                } else
                    errorLabel.setText("Sikertelen bejelentkezés! Rossz
                                                felhasználónév / jelszó.");
                    userNameTextBox.setText("");
                    passwordTextBox.setText("");
                }
        });
}
```

ServicesImpl.java (részlet)

```
public synchronized int isUserPasswordValid(User user) {
	Connection con = initConnectionToDB();
	int code = -1;
	try {
		PreparedStatement s = con.prepareStatement(
				"SELECT * FROM users WHERE username = ?",
				ResultSet.TYPE_SCROLL_INSENSITIVE,
				ResultSet.CONCUR_READ_ONLY);
		s.setString(1, user.getUsername());
		ResultSet rs = s.executeQuery();

		List result = ObjectFactory.convertToObjects(rs, User.class);
		User userFromDB = (User) (result.get(0));

		if (user.getPassword().equals(userFromDB.getPassword())) {
			for (User u : getLoggedInUsersList())
				if (u.getUsername().equals(user.getUsername())) {
					code = -2;
					con.close();
					return code;
				}

		PreparedStatement update = con.prepareStatement("UPDATE
				userstats SET logincount = logincount + 1,
				logintime = ? WHERE playerID = ?");
		update.setDate(1, new Date(user.getLoginDate().getTime()));
		update.setInt(2, userFromDB.getId());
		update.execute();

		user.setPassword("&");
		user.setId(userFromDB.getId());
		DB.registerUserLogin(user);
		code = user.getId();
		rs = con.createStatement().executeQuery("SELECT * FROM
				userstats WHERE playerID = "  + user.getId() + "");
		result = ObjectFactory.convertToObjects(rs, UserStats.class);
		UserStats us = (UserStats) (result.get(0));

		addEvent(GAME_DOMAIN, new UserLoginEvent(user, us));

		}
	} catch (SQLException ignore) {
	} finally {
		try {
			con.close();
		} catch (SQLException ignore) {
		}
	}
	return code;
}
```

A ResultSet-ek feldolgozására szolgáló segédosztály: [4]

```
public class ObjectFactory {

    public static String convertPropertyName(String name) {
        String lowerName = name.toLowerCase();
        String[] pieces = lowerName.split("_");
        if (pieces.length == 1) {
            return lowerName;
        }
        StringBuffer result = new StringBuffer(pieces[0]);
        for (int i = 1; i < pieces.length; i++) {
            result.append(Character.toUpperCase(pieces[i].charAt(0)));
            result.append(pieces[i].substring(1));
        }
        return result.toString();
    }

    public static List<?> convertToObjects(ResultSet rs, Class cl) {
        List result = new ArrayList();
        try {
            int colCount = rs.getMetaData().getColumnCount();
            while (rs.next()) {
                Object item = cl.newInstance();
                for (int i = 1; i <= colCount; i += 1 ) {
                    String colName = rs.getMetaData().getColumnName(i);
                    String propertyName = convertPropertyName(colName);
                    Object value = rs.getObject(i);
                    PropertyDescriptor pd = new
                                    PropertyDescriptor(propertyName, cl);
                    Method mt = pd.getWriteMethod();
                    mt.invoke(item, new Object[] {value});
                }
                result.add(item);
            }
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
        return result;
    }
}
```

A kliens oldalról meghívható szerverhívások (Services.java):

```
public int isUserPasswordValid(User user);
public void logoutUser(User user);
public boolean registerNewUser(User user);
public UserStats getUserStats(User user);
public ArrayList<User> getLoggedInUsersList();
public ArrayList<UserStats> getLoggedInUsersStats();
public void sendMessageToGamePartner(User partner, String message);
public String proposeGameTo(User from, User to);
public void replyToProposal(User opponent, boolean answer);
public void sendMoveToOpponent(User opponent, int row, int col);
public void gameStarted(User player1, User player2);
public void gameWonByUser(User user);
```

Célállapotvizsgálat teljes kódja:

```
public boolean isGameEndState() {
	if (isGreenBasesSelected())
		for (Line line : greenPlayerLines) {
			if (line.getP1().getX() == 0) {
				return findWay(greenPlayerLines, line.getP1(),
						GREEN_PLAYER);
			} else if (line.getP2().getX() == 0) {
				return findWay(greenPlayerLines, line.getP2(),
						GREEN_PLAYER);
			}
		}
	if (isBlueBasesSelected())
		for (Line line : bluePlayerLines) {
			if (line.getP1().getY() == 0) {
				return findWay(bluePlayerLines, line.getP1(),
							BLUE_PLAYER);
			} else if (line.getP2().getY() == 0) {
				return findWay(bluePlayerLines, line.getP2(),
							BLUE_PLAYER);
			}
		}
	return false;
}

private boolean findWay(ArrayList<Line> userlines, Point currentEndPoint,
							int player) {
	if (currentEndPoint.getX() == TABLE_SIZE - 1 && player ==
							GREEN_PLAYER) {
		return true;
	} else if (currentEndPoint.getY() == TABLE_SIZE - 1 && player ==
							BLUE_PLAYER) {
		return true;
	}
	if (userlines.size() == 0)
		return false;

	ArrayList<Line> lines = new ArrayList<Line>();
	lines.addAll(userlines);
	for (Line line : lines) {
		if (line.getP1().equals(currentEndPoint)) {
			userlines.remove(line);
			boolean result = findWay(userlines, line.getP2(),
								player);
			if (result == true)
				return true;
		} else if (line.getP2().equals(currentEndPoint)) {
			userlines.remove(line);
			boolean result = findWay(userlines, line.getP1(),
								player);
			if (result == true)
				return true;
		}
	}
	return false;
```

```
}

private boolean isBlueBasesSelected() {
	int i, j;
	for (i = 0; i < TABLE_SIZE; i++) {
		if (gameTable[i][0] < 0) {
			for (j = 0; j < TABLE_SIZE; j++) {
				if (gameTable[j][TABLE_SIZE - 1] < 0) {
					return true;
				}
			}
			if (j == TABLE_SIZE)
				return false;
		}
	}
	return false;
}

private boolean isGreenBasesSelected() {
	int i, j;
	for (i = 0; i < TABLE_SIZE; i++) {
		if (gameTable[0][i] > 0) {
			for (j = 0; j < TABLE_SIZE; j++) {
				if (gameTable[TABLE_SIZE - 1][j] > 0) {
					return true;
				}
			}
			if (j == TABLE_SIZE)
				return false;
		}
	}
	return false;
}
```

Irodalomjegyzék

[1] MD5 algoritmus JSNI verziója: http://groups.google.com/group/Google-Web-Toolkit/browse_thread/thread/ad09475a9944c9f8

[2] GWTEventService: http://code.google.com/p/gwteventservice/

[3] Noel Rappin: Build an Ajax application using Google Web Toolkit, Apache Derby, and Eclipse, Part 1: The fancy front end http://www.ibm.com/developerworks/library/os-ad-gwt1/

[4] Noel Rappin: Build an Ajax application using Google Web Toolkit, Apache Derby, and Eclipse, Part 2: The reliable back end http://www.ibm.com/developerworks/opensource/library/os-ad-gwt2/index.html

[5] Noel Rappin: Build an Ajax application using Google Web Toolkit, Apache Derby, and Eclipse, Part 3: Communication http://www.ibm.com/developerworks/library/os-ad-gwt3/

[6] Noel Rappin: Build an Ajax application using Google Web Toolkit, Apache Derby, and Eclipse, Part 4: Deployment http://www.ibm.com/developerworks/java/library/os-ad-gwt4/

[7] Darel Rex Finley: Line-Line Intersection Method http://alienryderflex.com/intersect/

[8] Server Push: http://code.google.com/p/google-web-toolkit-incubator/wiki/ServerPushFAQ

[9] 5 GWT Anti-Patterns: http://www.zackgrossbart.com/hackito/antiptrn-gwt/ http://www.zackgrossbart.com/hackito/antiptrn-gwt/

[10] 4 More GWT Anti-Patterns: http://www.zackgrossbart.com/hackito/antiptrn-gwt2/ http://www.zackgrossbart.com/hackito/antiptrn-gwt2/

[11] Alfa-béta vágás: http://aima.cs.berkeley.edu/java/src/aima/games/Game.java http://aima.cs.berkeley.edu/java/src/aima/games/Game.java

[12] Jeszenszky Péter: Minta állapottér reprezentáció http://www.inf.unideb.hu/~jeszy/download/mestint/knightstour.pdf

[13] Dr. Várterész Magda: Mesterséges intelligencia 1 http://www.inf.unideb.hu/~varteres/mi1folia/foliafo.pdf

[14] Stuart J. Russell - Peter Norvig: Mesterséges intelligencia-Modern megközelítésben. 2005 , Panem kiadó

Köszönetnyilvánítás

Szeretnék köszönetet nyilvánítani a barátnőmnek a lelki támogatásért, a végtelen türelméért és a sok csokiért amiket tőle kaptam, a barátaimnak a megértésükért és építő jellegű és hasznos tanácsaikért, és külön köszönetet érdemel Mayer Ádám, aki megismertette velem ezt a játékot még középiskolában és segített a heurisztikák megalkotásában sok játék közös elemzése során.