

**Szegedi Tudományegyetem
Informatikai Tanszékcsoport**

Heurisztikák összevetése a TRON játékra

Comparing heuristics for TRON

Szakdolgozat

Készítette:
Bajzáth Dávid
programtervező
informatikus hallgató

Témavezető:
Dr. Iván Szabolcs
adjunktus

**Szeged
2015**

Tartalom

Feladatkiírás	4
Heurisztikák összevetése egy NP-nehez labirintusjátékra	4
Tartalmi összefoglaló	5
1. Játékszabályok	6
1.1. Tron játék szabályai	6
1.2. A szakdolgozatban szereplő módosított Tron.....	6
2. Elméleti háttér.....	7
2.1. Eldönthetőség.....	7
2.2. Bonyolultságelmélet	7
2.3. Problémák visszavezetése	8
2.4. Hamilton-út	8
2.5. Tron visszavezetése a Hamilton-útra	8
3. Gépi játékosok	9
3.1. Google AI Challenge	9
3.2. Random AI.....	9
3.3. Minimax AI.....	9
4. Pályageneráló algoritmus.....	12
5. Implementációhoz használt technológiák	13
5.1. MVC	13
5.2. Sun Java kódolási konvenció	13
5.3. Verziókezelő	14
5.4. Java Swing	14
6. Implementáció.....	15
6.1. Indító képernyő	15
6.2. Szimulációs mód.....	16
6.3. Grafikus mód	17

6.4.	Játéktér	18
6.5.	Játéktérre elhelyezhető elemek	19
6.6.	Játéktér generálása	19
6.7.	Játéktér kezelése.....	21
6.8.	Játéktér megjelenítése	22
6.9.	Kliensek	24
6.10.	Kliensek kezelésének alapja.....	25
6.11.	Emberi játékosok kezelése	25
6.12.	Gépi játékosok kezelésének alapja.....	26
6.13.	Random gépi játékos kezelése	26
6.14.	Minimax algoritmust használó gépi játékos kezelése	26
6.15.	Heurisztikák	28
7.	Teszteredmények	30
7.1.	Tesztbeállítások.....	30
7.2.	Szimulációs módban futtatott tesztek	30
7.3.	Grafikus módban futtatott tesztek	32
8.	Konklúzió.....	35
9.	Fejlesztési lehetőségek	36
	Irodalomjegyzék.....	37
	Nyilatkozat	38

Feladatkiírás

Heurisztikák összevetése egy NP-nehéz labirintusjátékra

Az egyszemélyes logikai és puzzle játékok jelentős része *NP*-teljes, a kétszemélyesek pedig gyakran *PSPACE*-teljesek. A legismertebb ilyen probléma a Földrajzi Játék, melyben két játékos lépked felváltva egy $G = (V, E)$ gráf csúcsain. Az I. játékos kezd az első, kijelölt kezdőcsúcs megnevezésével, majd a soron következő játékosnak mindig egy olyan csúcsot kell megnevezni, mely még nem szerepelt addig és melybe vezet él a legutóbb megnevezett csúcsból. Aki nem tud ilyen csúcsot megnevezni, veszített. A hallgató feladata definiálni a Földrajzi Játék egy továbbra is legalább *NP*- és *coNP*-nehéz variánsát, melyet számítógépen ember számára „átlátható” módon lehet megjeleníteni (pl. gridgráfon játszott variáns jöhet szóba), megmutatni, hogy az általa választott variáns valóban nehéz ezen bonyolultsági osztályokra nézvést, és fejleszteni hozzá legalább két gépi játékost, valamint egy lehetőleg grafikus felületet, melyen emberi játékos játszhat más ember, vagy valamelyik gépi játékos ellen. Szimulációs üzemmódban futtatva elemezze ki a fejlesztett gépi játékosok hatékonyságát egymás, ill. emberi játékos ellen.

Tartalmi összefoglaló

A szakdolgozat témája egy földrajzi játék definiálása volt, *NP*- és *coNP*-nehézségének bizonyítása, majd a problémára írt heurisztikák hatékonyságának összehasonlítása egymással és emberi játékos ellen. A dolgozatban először ismertetem a Tron játékot, amely a dolgozatban szereplő földrajzi játék alapjául szolgált. A témaként szolgáló játék szabályainak ismertetésével folytatom, ezek után felvázolom a bonyolultságához az elméleti háttérrel a játék nehézségének megértéséhez. Ezután következnek hasonló problémák, amelyek visszavezethetők a játékra és ezek segítségével bizonyítom *NP*- és *coNP*-nehézségét. Bemutatom a játékhoz használt mesterséges intelligenciák elméleti háttérét és az optimalizálásukhoz használt vagy kipróbált technikákat. Ezt a megvalósított pályageneráló algoritmus ismertetése követi. Utána következik a játék Java implementációjához használt technológiák ismertetése, ahol bemutatok különböző ajánlásokat és segédeszközöket, amelyek megkönnyítették a program elkészítését, vagy a megértéséhez nyújtanak segítséget. Ezt követi maga az implementáció ismertetése. Végül a különböző heurisztikákat hasonlítom össze egymással és emberi játékosokkal szemben nyújtott teljesítményük alapján, és összegzem az elért eredményeket.

Kulcsszavak: NP-nehéz, Földrajzi játék, Gráf, Minimax algoritmus, Alfa-béta vágás

1. Játékszabályok

1.1. Tron játék szabályai

A Tron egy egyszerű nullaösszegű kétszemélyes játék, melyben két játékos halad egy pályán, akik minden lépésükkel csíkot húznak maguk után. A játékos, aki hamarabb ütközik falba vagy egy játékos által húzott csíkba, az veszít. Ha a két játékos egyszerre ütközik vagy ugyanarra a pozícióra próbálnak lépni, akkor az eredmény döntetlen.

1.2. A szakdolgozatban szereplő módosított Tron

Az eredeti játékban a játékosok egy gráf élein haladnak, míg a dolgozat alapjául szolgáló játékban cellákon, valamint bevezetésre került egy nehezítés, ugyanis a pályán random generált akadályok is nehezíthetik a haladást, amelynek ütközve ugyancsak kiesik az adott játékos. Ezek érdekes új kihívást állítanak a már létező játékhoz tervezett gépi játékosok elé.

2. Elméleti háttér

2.1. Eldönthetőség

Eldöntési problémának nevezzük azokat a problémákat, melyek megoldása „igen” vagy „nem” lehet. Optimalizálási problémákat is visszavezethetünk ilyen alakra, amely akkor lesz „igen”, ha az optimum kisebb vagy egyenlő (maximalizálási problémánál nagyobb vagy egyenlő) a megadott értéknél. Egy probléma akkor eldönthető, ha létezik őt eldöntő algoritmus. Közismert eldönthetőségi problémák például a legrövidebb út és a Hamilton kör.

2.2. Bonyolultságelmélet

A bonyolultságelmélet a problémákat a megoldásukhoz szükséges idő és tár alapján különböző osztályokra bontja^[4].

- $TIME(f(n))$: legfeljebb $f(n)$ idő alatt eldönthető problémák osztálya
- $NTIME(f(n))$: nemdeterminisztikus algoritmussal legfeljebb $f(n)$ idő alatt eldönthető problémák osztálya
- $SPACE(f(n))$: legfeljebb $f(n)$ tárból eldönthető problémák osztálya
- $NSPACE(f(n))$: nemdeterminisztikus algoritmussal legfeljebb $f(n)$ tárból megoldható problémák osztálya

Egy algoritmus polinomiális idejű, ha n méretű bemeneten a futási idejük legrosszabb esetben is $O(n^k)$ valamely k nemnegatív konstansra. Általánosságban elfogadott, hogy egy polinomiális idejű algoritmust már jó megoldásnak tekinthetünk, de könnyen belátható, hogy túl nagy k esetén ezek az algoritmusok is nagy erőforrásokat emészthetnek fel. A polinomiális időben eldönthető problémák osztályát $P = TIME(n^k)$ jelöli. A nemdeterminisztikus polinom idejű algoritmusokkal eldönthető problémák az $NP = NTIME(n^k)$ osztályba tartoznak. A nemdeterminizmust csak tökéletes párhuzamosítással érhetnénk el, amely feltételezi, hogy például a teljes tárat átmásoljuk egyetlen lépésben. Ez nem reális, így valós architektúrán az NP osztályba tartozó problémákra adott algoritmusok exponenciális idejűek. Egy döntési probléma a $coNP$ osztályba tartozik, ha komplementere (amely azonos bemenetre ellentétes eredményt ad) NP -beli probléma. Ezekből a definíciókból következik, hogy $P \subseteq NP$. Egyelőre nem bizonyított, hogy $P \subset NP$ vagy $P = NP$, de intuitíve az első látszik igaznak.

2.3. Problémák visszavezetése

Egy A probléma visszavezethető B -re, ha létezik olyan polinom idejű $f(x)$ visszavezető algoritmus, melyre $A(x) = B(f(x))$ minden x bemenetere A -nak. Ha A visszavezethető B -re, akkor A legfeljebb olyan nehéz, mint B , ha pedig B is visszavezethető A -ra, akkor egyforma nehezek. Egy probléma NP -nehéz, ha minden NP -beli probléma visszavezethető rá, és NP -teljes, ha emellett még NP -beli is. Ezt megfordítva ha B NP -nehézségét akarjuk bizonyítani, akkor keresünk egy NP -nehéz problémát (ilyen például a SAT vagy a Hamilton-út probléma^[7]) és B -re visszavezetjük. Ha sikerül, akkor igazoltuk, hogy B legalább olyan nehéz, mint a rá visszavezetett probléma.

2.4. Hamilton-út

Egy P út egy $G = (V, E)$ gráfban Hamilton-út, ha P a V összes elemét pontosan egyszer tartalmazza. A Hamilton-út probléma további megszorításokkal nehezített változatainak egy jelentős része is NP -teljes. A Hamilton-út probléma síkbeli irányított gráfokban, melyekben a csúcsok fokszáma 3 is NP -teljes^[6]. Ez pedig visszavezethető a Hamilton-út problémára gridgráfokban, amely így szintén NP -teljes^[1]. A gridgráf egy olyan gráf, melynek csúcsai (i, j) egész koordinátájú pontok a síkon, és két csúcs akkor és csak akkor szomszédos, ha távolságuk pontosan 1. Ha egy gridgráf néhány csúcsát véletlenszerűen tilosra állítjuk, akkor ebben ugyancsak NP -teljes marad a Hamilton-út probléma eldöntése.

2.5. Tron visszavezetése a Hamilton-útra

Ha egy X szabad mezővel rendelkező gridgráf egyik pontjáról indítunk egy játékost, valamint egy $X - 1$ hosszú folyosón elindítjuk az ellenfelet, akkor és csak akkor nyerhet a játékos, ha a gráfban van Hamilton-út. Ugyanis az ellenfél biztosan $X - 1$ lépés után fut falba, és csak akkor tud a játékos X -et lépni, ha a gráfban található Hamilton-út, így a probléma NP -nehéz. Ezt megfordítva, ha a játékost indítjuk a folyosóról és az ellenfelet a gráfból, akkor megkapjuk, hogy a probléma $coNP$ -nehéz is. Ez tehát azt jelenti, hogy egy állásból NP - és $coNP$ -nehéz eldönteni, hogy melyik játékos fog nyerni, így csak heurisztikus gépi játékosokat lehet a játékhoz fejleszteni (hiszen mindig gyorsan és optimálisan lépő, „verhetetlen” ellenfél csak akkor lenne létrehozható, ha $P = NP$).

3. Gépi játékosok

3.1. Google AI Challenge

2010 telén a University of Waterloo Computer Science Club a Google szponzorálásával elindított egy kihívást^[3]. Egymással versenyző Tron AI-k implementálása volt a feladat, azzal a kikötéssel, hogy 1 másodperce van eldönteni a választott irányt minden körben. Ehhez adtak több programozási nyelven is alapot, valamint az oldalon találhatóak cikkek különböző stratégiákról is. Ezek remek kiinduló alapot adtak a saját implementációmhoz.

3.2. Random AI

A Random AI a probléma egyszerű megoldása, ugyanis a jelenlegi pozíciója körüli padlók közül választ véletlenszerűen egyet, és arra lép. Mivel nem gondolkodik előre, így gyakran kanyarodik zsákutcákba, nem túl kifinomult megoldás.

3.3. Minimax AI

A Tron egy kétszemélyes zéróösszegű játék, ugyanis amennyi pontot szerez az egyik játékos, az ellenfele pontosan annyit veszít. Az ilyen játékok lehetséges állásait játékgráffal tudjuk ábrázolni^[5], amely elemei:

- lehetséges állapotok halmaza (azaz legális játékállások)
- egy kezdőállapot
- lehetséges lépések halmaza és egy állapotátmenet függvény, amely a lépésekhez állapotokat rendel
- végállapotok halmaza, ami a lehetséges állapotok részhalmaza
- hasznosságfüggvény, amely minden lehetséges végállapothoz hasznosságot rendel

Két játékos felváltva lép ezen a játékgráfon, az egyik maximalizálni szeretné a lépés értékét, míg ellenfele minimalizálni. A minimax algoritmus ezeket számolja ki minden n csúcsra:

$$\text{minimax}(n) = \begin{cases} \text{hasznosság}(n) & \text{ha végállapot} \\ \max_{a \text{ szomszédja}} \text{minimax}(a) & \text{ha MAX jön} \\ \min_{a \text{ szomszédja}} \text{minimax}(a) & \text{ha MIN jön} \end{cases}$$

A teljes gráf értékét a kezdőcsúcsból indított maximalizáló hívással tudjuk kiszámítani. Ha a játékgráfban van kör, akkor nem terminál, de a gyakorlatban csak adott mélységig futtatjuk és egy viszonylag könnyen számítható heurisztikát rendelünk a levél állapotokhoz, mivel az

algoritmus exponenciális futási idővel rendelkezik. Ezen a futási időn különböző optimalizálásokkal lehet javítani, de nem ismerünk polinom időben futó algoritmust rá.

Az egyik optimalizálási módszer az alfa-béta vágás. Eszerint ha a játékos már ismer egy x értékű stratégiát egy n csúcsban, akkor, ha az n egyik szomszéd csúcsában az ellenfél egy x -nél rosszabb értéket tud kikényszeríteni, akkor azt a csúcsot nem vizsgáljuk tovább, hiszen ekkor az n csúcsot fogja választani. Ehhez a minimax algoritmus hívásakor felveszünk egy alfa (maximalizáló játékos legjobb értéke) és egy béta (minimalizáló játékos legjobb értéke) értéket. Ha az alfánál rosszabb értéket tud kikényszeríteni egy csúcsban az ellenfél, vagy a bétánál rosszabbat engedünk meg neki, akkor a csúcs leszármazottait nem vizsgáljuk tovább. Az alfa-béta vágás algoritmus hatékonysága nagyban függ a csúcsok rendezésétől, legrosszabb esetben futási ideje megegyezik a minimax algoritmusével, legjobb esetben kétszer mélyebb fát is be tud járni.

Az iteratív mélységi keresés algoritmus alapötlete az, hogy a mélységi keresést többször, mindig egy szinttel tovább futtatja. Ha ezt megfeleltetjük az alfa-béta vágásnak és minden körben értékük szerint rendezzük a csúcsokat (maximalizáló kör esetén csökkenő, minimalizáló esetén növekvő sorrendbe) akkor az alfa-béta algoritmus valószínűleg több részfat tud levágni. Előnye az alfa-béta vágás optimalizálásán felül az, hogy az n -edik ($n > 1$) futáskor már van egy $n - 1$ mélységre optimális megoldásunk, hátránya hogy a csúcsokat többször is ki kell értékelnünk.

A játékhoz két heurisztikát készítettem, az elsőt Wallhugger-nek neveztem el, alapjául a Google AI Challenge azonos nevű algoritmusát szolgált^[3]. Az alap ötlet az, hogy mindig próbáljon fal mellett haladni, így jól ki tudja majd tölteni a rendelkezésére álló helyet. Először csak a szomszédos falak számát állítottam be az állás értékének, de így mindig visszafordult magába és gyorsan meghalt, ezért bővítettem. Ha végzetes a lépés, akkor értéke -1 , különben pedig a pozícióról elérhető padlók számának és a szomszédos falak számának az összege.

A játékhoz használt második heurisztikát Separator-nek neveztem el és a Pablo de Oliveira Castro „A simple Tron bot” cikkében ismertetett módszeren alapul^[8], kisebb módosításokkal. A végállapotokhoz döntetlen esetén 0 , vereség esetén a padlók száma $\cdot -1$, nyereség esetén pedig a padlók száma értéket rendeli a csúcshoz. Nem végállapot esetén először megvizsgáljuk, hogy a két játékos el van-e szeparálva egymástól. Ha nem, akkor közelítenünk kell az ellenfélhez, így a csúcs értéke $1 / \text{kliensek távolsága}$. Ha el vagyunk szeparálva, akkor azok az erős pozíciók, ahol több padlót érünk el, ezért a csúcs értéke a játékos által elérhető és

nem elérhető pozíciók különbsége. A végállapotok értékelése magától értetődő, a többi eset már kevésbé. Nem szeparált állások esetén azért a távolságot vesszük a pozíció értékének alapjául, mert a játék megnyerésének legegyszerűbb módja az, ha az elején nagyobb szeletet tudunk levágni magunknak, mint az ellenfél, tehát érdemes közelíteni hozzá. Miután közel értünk, le kell választanunk ezt a minél nagyobb szeletet és elzárni az ellenfél elől. Ekkor már szeparált állapotot kapunk, ennek kiértékelése következik, amelyben a játékos által elérhető padlók számának arányát vesszük az összes padlóéhoz képest (azért nem a két játékos által elérhető padlók számának különbségét, mert azt túl erőforrásigényes lenne kiszámítani).

4. Pályageneráló algoritmus

A pálya alaphelyzetben csak padlókból áll és az akadályok száma 0. Az erre helyezhető akadályoknak van magassága, és egy pályaelemen egyszerre csak egyet emelhetünk.

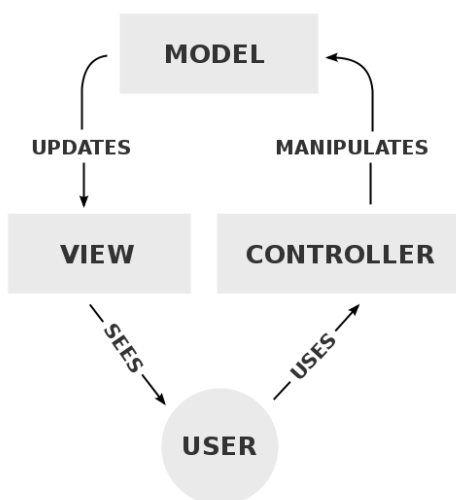
```
void function RAISE_CELL(point, height = INT_MAX)
    if (!grid.contains(point) || grid.heightAt(point) >= height) then RETURN;
    minNeighborHeight = grid.heightAt(point);
    grid.heightAt(point)++;
    loop direction in directions do
        RAISE_CELL(direction.translate(point), minNeighborHeight);
    end loop
end function
```

Kiválasztunk egy véletlenszerű pontot a pályán, és megpróbáljuk megemelni a RAISE_CELL függvénnyel. Ha a pont lelóg a pályáról vagy legalább olyan magas, mint amire emelni szeretnénk, akkor megszakítja a futást, különben emel egyet a magasságán, és minden szomszédjára meghívódik a pont előző magasságával. Mindaddig választunk új véletlenszerű pontot, amíg az akadályok aránya el nem éri a kívánt arányt. Ez után megvizsgáljuk, hogy a generált pálya szabályos-e, azaz elérhető-e minden padló mező. Megkeressük az első padló mezőt, és innen indulva meglátogatjuk a szomszédos mezőket. Ha a mező lelóg a pályáról, nem padló, vagy már meglátogattuk, akkor visszatérünk, különben felvesszük a már meglátogatott listára és meglátogatjuk a szomszédjait is. Ha a meglátogatott pontok listájának mérete pontosan a pálya elemeinek száma – akadályok száma, akkor a pálya szabályos, különben generálunk egy újat. Az eredeti ötlet szerint minden pont emelése után tesztelte volna az algoritmus, hogy szabályos-e a pálya és csak akkor véglegesítette volna az akadályokat, ha igen, de a szabályosság vizsgálat túlságosan erőforrás igényesnek bizonyult. Mindemellett alacsony akadály számnál és azzal a kikötéssel, hogy a pálya szélein 1 mező széles területeken nem lehetnek akadályok, a szabálytalan pálya generálásának valószínűsége viszonylag alacsony (30*20-as pályán 5%-os akadály aránnyal 15 futásból 14-szer elsőre szabályosat generált, egyszer pedig másodjára).

5. Implementációhoz használt technológiák

5.1. MVC

Az MVC egy tervezési minta, amely 3 részre osztja az alkalmazás elemeit: modell (model), nézet (view) és vezérlő (controller)^[10]. A modellek az alkalmazásban használt adatok reprezentációi, semmilyen információjuk nincs a környezetükről. A nézetek szolgálnak a modellek megjelenítésére, ezeken keresztül tud a felhasználó hozzájuk férni. A vezérlők valósítják meg az alkalmazás logikáját, kapcsolódnak modellekhez és nézetekhez is.



5.1.1. ábra. A felhasználó és az MVC elemeinek kapcsolata

Előnye, hogy szeparálja az információ belső reprezentálását a felhasználóval való interakciótól. Azért választottam ezt a tervezési mintát, mert az alkalmazásomban van egy szimulációs és egy grafikus mód, amelyek ugyanazokkal a modellekkel és logikával dolgoznak, ezt kihasználva elég különböző nézeteket megvalósítani hozzájuk.















5.2. Sun Java kódolási konvenció

A kódolási konvenciók ajánlások kódolási stílusra. Érdeemes egy jól bevált konvenciót követni, amivel biztosíthatjuk, hogy ha más (vagy akár saját magunk hosszabb idő után) értelmezni próbálja a kódunkat, akkor könnyen olvasható lesz számára. Ezekben ajánlást adnak a kódunk formázására, például a sorok maximális hossza, hogyan törjük meg, ha túl hosszú lenne. Fájl, osztály, metódus és változó elnevezések, a kódunkban használt sortörések és szóközök helye és száma, a kommentek formázása, tartalmuk vázlata és még sok más ajánlás is megtalálható ezekben a konvenciókban. Az implementációmhoz az 1999. április 20-

án átdolgozott, Sun által publikált kódolási konvenciót választottam^[9], ez az egyik legismertebb Java kódolási konvenció.

5.3. Verziókezelő

A verziókezelők fájlok több verzióban való tárolását kezelik, ez rengeteg előnyt hordoz magában^[2]. Könnyen visszaállíthatóak a fájlok egy régebbi verzióra, több ember munkája egy fájlban szinkronizálható, a változások követhetőek kommentek segítségével (5.3.1. ábra), minden módosításhoz rendel felhasználót (így felelőst is), valamint elágazásokat is kezel a kód különböző verziói között, így például egy új ág létrehozásával kipróbálható egy új implementáció az eredeti verzió elvesztésének kockázata nélkül. Az alkalmazásomhoz én a Git verziókezelő szoftvert választottam, amelyhez a GitHub egy kiváló kezelőfelületet biztosít. A repository a következő linken érhető el: <https://github.com/Bajzathd/TRON>.

	Bugfixek és refactor Bajzathd committed 5 days ago	 2704e10	
	Client lépés bugfix ... Bajzathd committed 5 days ago Eddig csak az épp beállított iránnyal hasonlított össze az új irányt hogy egymással ellentétesek-e, de ha pl. a játékos egy kör alatt 2-szer váltott irányt akkor ellentétes irányba tudott lépni ami azonnal megölte, ez került javításra.	 bcabccc	
	MVC megszegésének javítása ... Bajzathd committed 5 days ago A StartScreen view eddig hozzáfért a Client modellekhez, ez került javításra.	 5090b44	
	AI szint módosítása ... Bajzathd committed 5 days ago Mostantól a 0-s szint jelöli a random algoritmusú AI-t, és a StartScreen Spinnerjén páros számú szinteket tudunk választani.	 64b630e	
	MinimaxAI szeparált állás átértékelése ... Bajzathd committed 5 days ago Szeparált állás értéke eddig a kliens által elérhető padlók és az ellenfél által elérhetőek különbsége volt, ehhez adtuk most hozzá a döntetlen állás értékét.	 887b410	

5.3.1. ábra. Részlet a commit logból

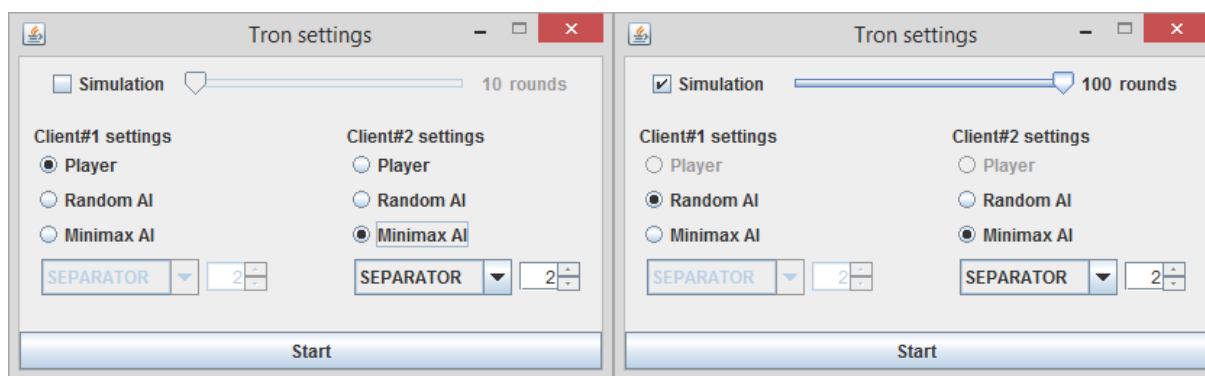
5.4. Java Swing

A Swing egy GUI-k készítéséhez alapot nyújtó API, amely az Oracle Java Foundation Classes (JFC) egyik eleme^[11]. Az Abstract Window Toolkit (AWT) leváltására tervezték, platform független ablakokat hozhatunk vele létre, amelyek a futtató operációs rendszer kinézetét veszik fel. Sokak által használt, jól bevált API GUI-k készítéséhez és rengeteg tutorial érhető el hozzá online. Ezen okok miatt választottam a projektemhez.

6. Implementáció

6.1. Indító képernyő

Az alkalmazás belépési pontját a `StartScreen` osztály tartalmazza, amely egyetlen példányosítást tartalmaz, még hozzá a `StartScreen` osztályét. Ez az osztály a Swing API segítségével valósítja meg az indító képernyő szerepét, amelyen a felhasználó megadhatja a játék indításához szükséges beállításokat. Kezelő felülete három fő részre osztható: szimulációs beállítások, kliensek beállításai, valamint a beállítások jóváhagyására szolgáló start gomb (6.1.1. ábra).



6.1.1. ábra. Indító képernyő: bal oldalt grafikus módra, jobb oldalt pedig szimulációs módra állítva

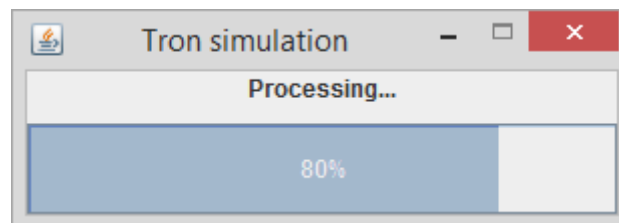
A szimulációs beállítások panel két elemet tartalmaz. Egyrészt az `isSimulation` `JCheckBox`-ot, amivel eldönthetjük, hogy a játékot grafikusan is meg szeretnénk-e jeleníteni, vagy pedig csak szimulált játékok naplózott eredményeit mentse el, másrészt a `StartScreen`-en belüli `SimulationRoundsPanel` privát osztály egy példányát, amely megjeleníti és kezeli a körök beállítását. Ez az osztály a `JPanel` osztályból lett származtatva, és megvalósítja a `ChangeListener` interfészt, így képes a slider-en történt változtatások észlelésére és tud rájuk reagálni. Szimulációs mód esetén módosíthatóvá válik a rajta található `JSlider`, amelyen beállíthatjuk, hány kört szeretnénk szimulálni egy 10-től 100-ig terjedő skálán, amely értékét a mellette található `JLabel` jeleníti meg.

A szimulációs beállítások alatt találhatóak a kliensek beállításai. Egy kliens beállításainak megjelenítéséért és kezeléséért felelős a `StartScreen` osztályban található `ClientSettings` privát osztály. Ez az osztály a `JPanel` leszármazottja, valamint implementálja az `ActionListener` interfészt, amelynek köszönhetően az osztály a rádió gombok közötti váltásra tud reagálni. Ennek segítségével csak akkor módosíthatóak a

Minimax AI beállításai, ha az van kiválasztva kliens típusnak. Ebből az osztályból két példány található a képernyőn, amelyeken az első és a második kliens beállításai találhatóak. Szimulációs mód esetén a játékos kliens típus nem választható, ugyanis ilyenkor a játék nincs grafikusan megjelenítve. Van jelentősége, hogy az első vagy pedig a második klienst választjuk játékosnak, ugyanis az első játékos irányítása a nyilakkal történik, míg a második játékosé a WASD billentyűkkel. Grafikus és szimulált módban is kiválasztható a Random AI és a Minimax AI heurisztikájával és szintjével együtt, amelyekről bővebben az AI-k megvalósításánál lesz szó. Egyelőre elég annyit tudni róla, hogy különböző heurisztikák más algoritmus alapján számítják egy-egy pozíció értékét, valamint hogy minél nagyobb szintű, várhatóan annál jobb döntéseket tud hozni.

A beállításainkat a start gombra kattintva véglegesíthetjük. Ekkor lefut a `startGame()` metódus, amely lekéri a két klienst a `ClientSettings` példányoktól, valamint az `isSimulation` checkbox állásától függően létrehoz egy új objektumot. Ha be van pipálva, akkor egy új `TronSimulation` példányt (szimulációs mód), különben egy új `Tron` példányt (grafikus mód) hoz létre a beállítások panelen beállított értékekkel, valamint az osztályban található konstansokkal.

6.2. Szimulációs mód



6.2.1. ábra. Visszajelzés a szimulációk futásának állapotáról

Szimulációs módban futtatva a játékot kizárólag AI-k versenyezhetnek egymással, amelyek többszöri lefutása után egy naplófájlba íródik a játékok statisztikája. Futtatás közben egy egyszerű képernyőt látunk, amely visszajelzést ad a szimulációk futásának állapotáról (6.2.1. ábra). A szimulációk kezelését a `TronSimulation` osztály végzi, amely megvalósítja a `Runnable` interfészt, így új szálon indítható a futása és egyszerre több szimuláció is futhat. A konstruktor a kapott paraméterek alapján beállítja az osztályváltozókat, létrehozza a saját nézetét (`SimulationView` osztály egy példánya), elindítja a naplózást (`Log` osztály egy példánya), majd a `rounds` változónak megfelelő mennyiségű új `SimulationRound` példányt hoz létre. Ezen objektumok mindegyike a konstruktorban megadott kliensekkel

elindít egy új szimulációt, amely frissíti a nézet progressbar-ját, valamint az eredményét eltárolja az ugyanitt átadott Log példánynak (6.2.2. ábra).

```
private void run() {
    try {
        engine.start();
        do {
            engine.update(log);
        } while (!engine.isOver());
        view.progress();
        log.setResult(engine.getGrid());
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
```

6.2.2. ábra. Szimulációs mód egy játéknak futtatása

Miután minden szimuláció lefutott a körök eredményeinek összesítését egy naplófájlba írjuk. Ennek a módnak a segítségével mértem össze a különböző heurisztikák egymáshoz mért hatékonyságát, ezt egy későbbi fejezetben fogom kifejteni.

6.3. Grafikus mód

Ha szeretnénk követni a játékot egy grafikus felületen, akkor az *isSimulation*-t kikapcsolt állapotban kell hagynunk a *StartScreen*-en a játék indításakor. Ekkor létrehoz egy új Tron példányt, amelynek átadjuk a pálya beállításait (amelyeket a *StartScreen* konstansai tárolnak), valamint a beállításoknak megfelelő két klienst. Ez az objektum lesz felelős a játék futtatásáért. A Tron osztály implementálja a *Runnable* interfészt, így új szálon indítható a futtatása. Konstruktórában kapott változókkal létrehoz egy *GridControllerWithView* példányt *engine* néven (ez lesz a játék motorja, később részletesen is ismertetem), majd elindít egy öt futtató szálát, amely meghívja a *run()* metódust.

```
engine.start();

do {
    lastTime = System.nanoTime();

    engine.update();
    engine.render();

    delta = System.nanoTime() - lastTime;

    if (delta < FRAME_LENGTH) {
        try {
            Thread.sleep((FRAME_LENGTH - delta) / 1000000L);
        } catch (InterruptedException ex) {}
    }
} while (!engine.isOver());

engine.showResults();
```

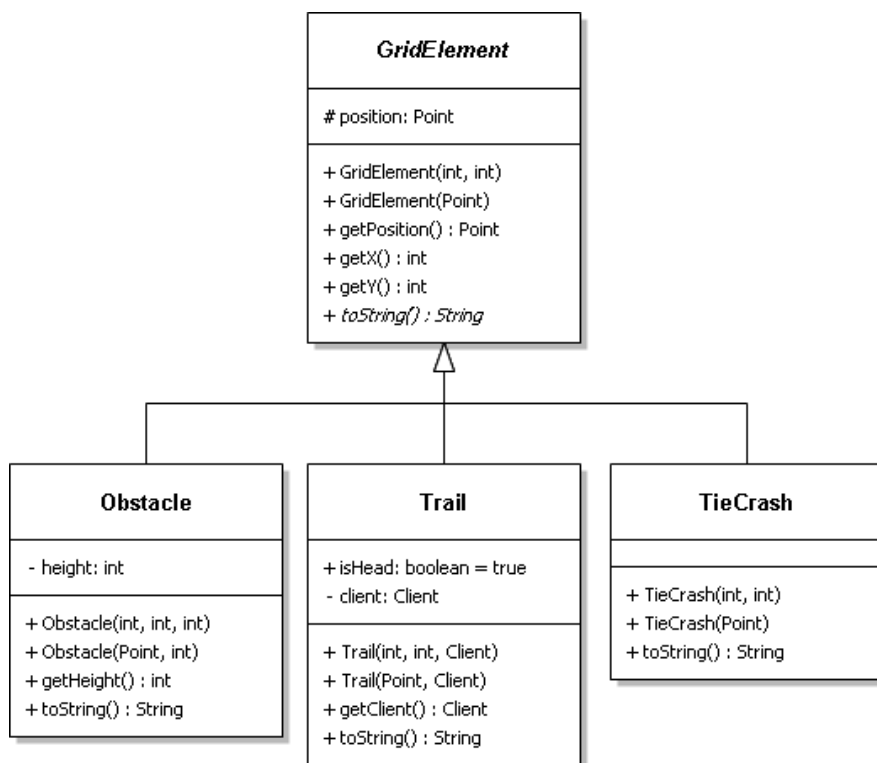
6.3.1. ábra. Játék futtatása grafikus módban (részlet)

Ez a `Runnable` interfész egyetlen megvalósítandó metódusa, esetünkben a játék futtatását végzi (6.3.1. ábra). Deklarálunk két `long` típusú változót a futási idő nyilvántartásához, ezekkel fogjuk maximalizálni a framerate-t, így követni lehet az eseményeket. Elindítjuk a játékot, amely kivételt dobhat, ha nincs beállítva mindkét kliens. Kivétel esetén kiírjuk az adatait és kilépünk az alkalmazásból 1-es hibakóddal. Ha sikeresen elindította a játékot, akkor következik a játék ciklus, ami addig lépteti a játékmotort és rajzolja ki az állást, amíg véget nem ér a játék. Ha véget ért, akkor megjelenítjük az eredményeket.

6.4. Játéktér

A `Grid` a játéktér reprezentációjáért felelős osztály. Fő feladata az elemek tárolása és az élő kliensek nyilvántartása. Ezeken felül a kezeléséhez hasznos metódusokat is tartalmaz, ilyen a `getValidDirections()`, amely visszaadja a megadott ponttal szomszédos padlók koordinátáit, valamint a `getAccessibleFloors()`, amely pedig az adott kliens által elérhető padlók listáját adja vissza.

6.5. Játéktérre elhelyezhető elemek



6.5.1. ábra. Játéktérre helyezhető elemek osztályainak hierarchiája (NClass UML diagramszerkesztő programmal készítve)

A **GridElement** absztrakt osztály a játéktéren elhelyezhető elemek őssztálya (6.5.1. ábra). Minden elem tárolja a saját pozícióját, és kötelező megvalósítania a `toString()` metódust. Az **Obstacle** osztály az akadályokat reprezentálja, pozícióján kívül egy magasság attribútummal is rendelkezik. A **Trail** osztály a kliensek által húzott csík egy elemének reprezentálása, pozícióján kívül két attribútuma van: az *isHead* tárolja, hogy ez az elem-e a csík utolsó eleme, ami alapértelmezetten igaz (hiszen ha most hoztuk létre, akkor értelemszerűen ez az utolsó), a *client* pedig azt a klienst tárolja, akihez tartozik az adott csík. A **TieCrash** osztály azt az esetet reprezentálja a pályán, amikor a két kliens ugyanarra a pontra próbál egyszerre lépni, így egymásnak ütköznek és döntetlen lesz a játék. Ezekből az elemekből, és a `null`-al reprezentált padlókból (amelyeken szabadon haladhatnak a kliensek) épülhet fel a játéktér.

6.6. Játéktér generálása

A pályageneráló algoritmust a **GridCreator** osztály valósítja meg. Konstruktórában a pálya dimenzióit és a minimális akadály arányt várja. A generált pálya magassága és szélessége is 2-vel kisebb, mint a végleges pálya, így fogja biztosítani, hogy a pálya szélén 1 elem

vastagságban végig padló van. Egy példány a megadott feltételeknek megfelelően több pályát is tud generálni, a `getGrid()` metódus hívásával. Ez generál egy `int[][]` tömböt, amelyben a 0-k reprezentálják a padlót (kezdetben minden elem ilyen), a $h > 0$ egész számok pedig a h magas akadályokat. Az akadályok emelését a `raiseCell()` metódus valósítja meg az algoritmusban specifikált módon (6.6.1. ábra).

```
private void raiseCell(Point p, int h) {
    if (!obstaclesBound.contains(p) || grid[p.y][p.x] >= h) {
        return;
    }
    int minNeighborHeight = grid[p.y][p.x];

    if (grid[p.y][p.x] == 0) {
        numObstacles++;
    }

    grid[p.y][p.x]++;

    for (Direction direction : Direction.values()) {
        raiseCell(direction.getTranslatedPoint(p), minNeighborHeight);
    }
}
```

6.6.1. ábra. Egy elem megemeléséért felelős metódus

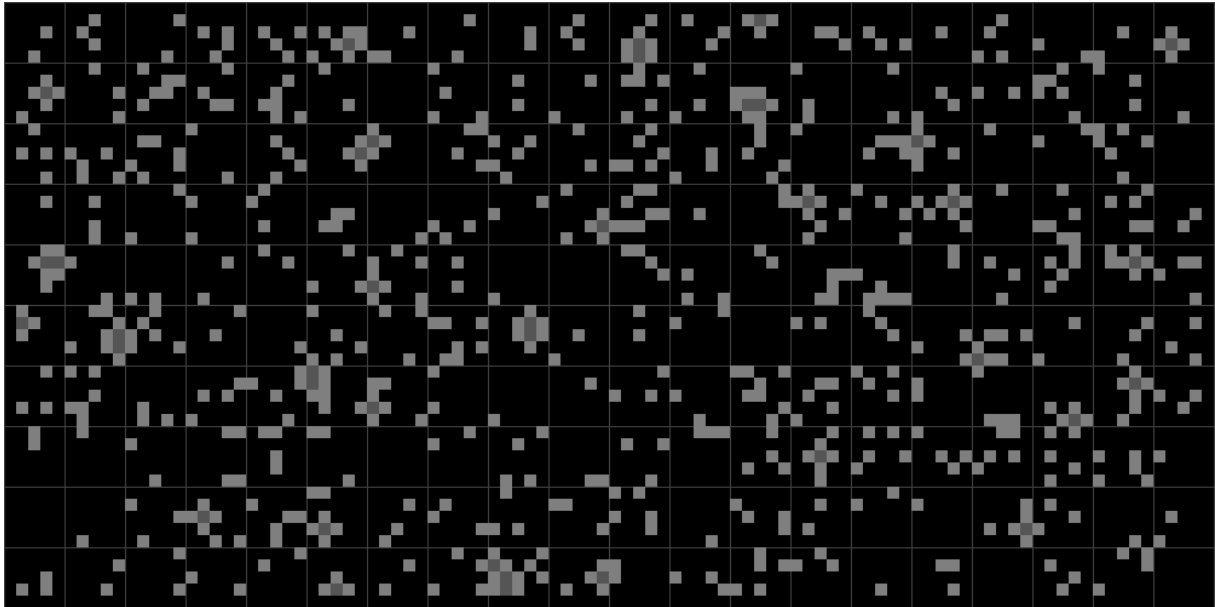
A `numObstacles` változóban tároljuk az akadályok számát, amely a generált játéktér érvényességének ellenőrzéséhez fontos információ, hiszen csak akkor érvényes, ha a bejárható padlók száma egyenlő az összes pont – akadályok számával (6.6.2. ábra).

```
private boolean isValidGrid() {
    try {
        visitPoint(getFirstFloor());
        return visitedPoints.size() == ((width * height) - numObstacles);
    } catch (NotFound ex) {
        return false;
    }
}
```

6.6.2. ábra. A generált pálya érvényességét eldöntő metódus

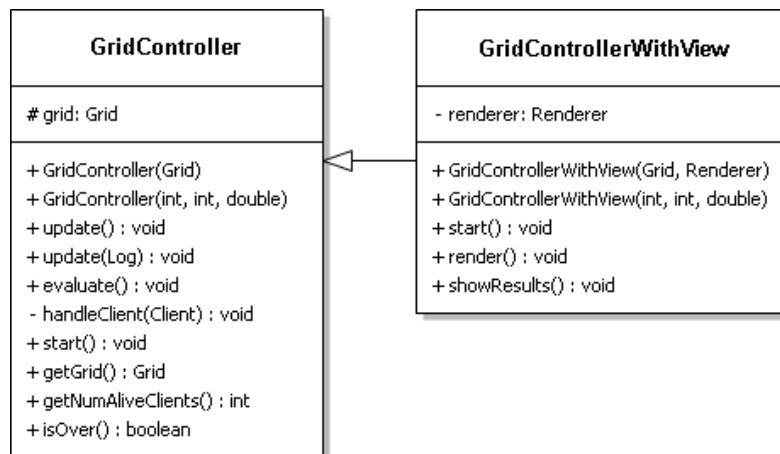
A pontok bejárásához megkeressük az első padló pontot a `getFirstFloor()` metódus hívásával (ha nincs ilyen, akkor `NotFound` kivételt dobunk), majd innen indulva rekurzívan meglátogatjuk a pontokat (ha még nem voltak, azaz nem tartalmazza őket a `visitedPoints`), és felvesszük őket a `visitedPoints` listára. Ha a játéktér nem érvényes, akkor újat generálunk. Érvényes esetén elkészítjük az `int[][]` tömbnek megfelelő `Grid`-et, amely a tömbben nem nulla értékű mezők mindegyikéhez készít egy új `Obstacle` példányt a

koordinátákkal (amik a tömb indexei) és a magassággal (tömb elem értéke), felveszi őket egy listára, valamint az *elements* `Object[][]` típusú tömb azonos indexére is elhelyezi. Az *elements* tömbbel és az akadályok listájával példányosítjuk a `Grid` osztályt, és visszatérünk a kapott objektummal.



6.6.3. ábra. A program által generált 100*50-es játéktér 15%-os akadály aránnyal

6.7. Játéktér kezelése

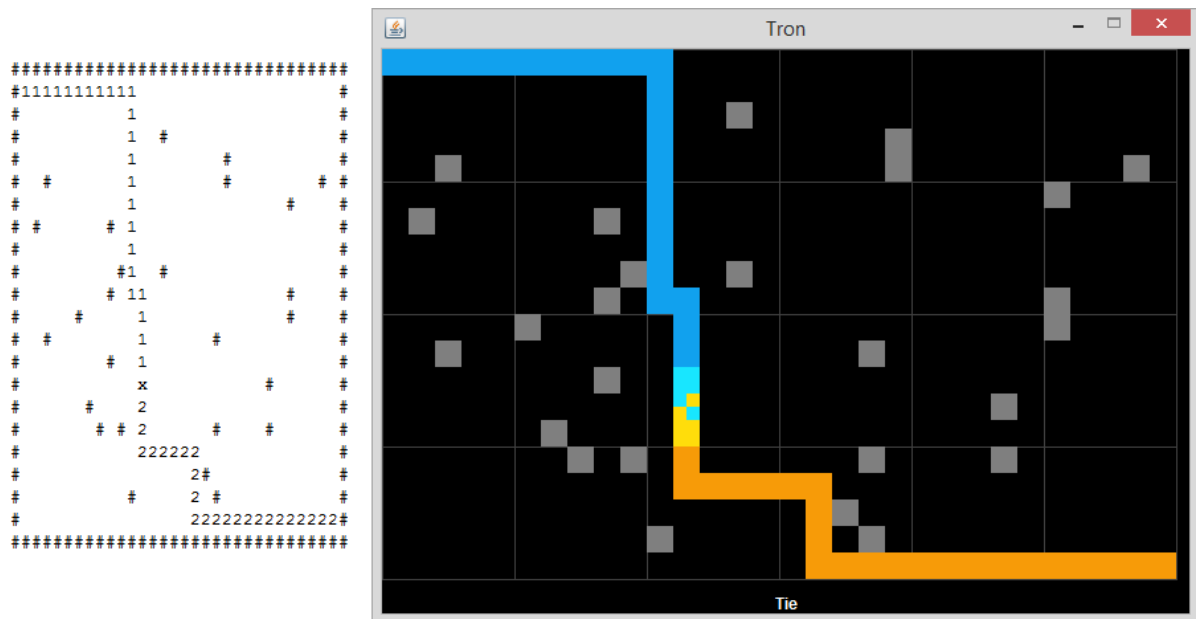


6.7.1. ábra. Játéktér kezelését végző osztályok hierarchiája

A `GridControllerWithView` a `GridController` osztály leszármazottja, amely a `Grid` típusú játéktér kezelésén felül egy `Renderer` interfészt megvalósító nézetet is kirajzol a játék állásának megfelelően (6.7.1. ábra). Példányosításakor vagy egy konkrét `Grid` és egy `Renderer` példányt adunk át neki, vagy pedig a létrehozandó `Grid` paramétereit. A második esetben a `GridController` konstruktorát hívja meg, amely létrehoz egy `GridCreator`

példányt és generáltat vele egy új játékkeret, amelyet a példány *grid* adattagjában eltárol. Ez után pedig példányosítja a *SwingRenderer* osztályt, amely a *Renderer* interfész *Swing* API-t használó megvalósítása. A *render()* és a *showResults()* metódusok a *renderer* azonos nevű metódusait hívják meg a kellő adatokkal, a *start()* metódus pedig meghívja a *Grid* azonos nevű metódusát, amely elhelyezi és elindítja a klienseket a pályán, majd a *renderer*-hez hozzáadja az emberi játékosok *KeyListener*-jeit.

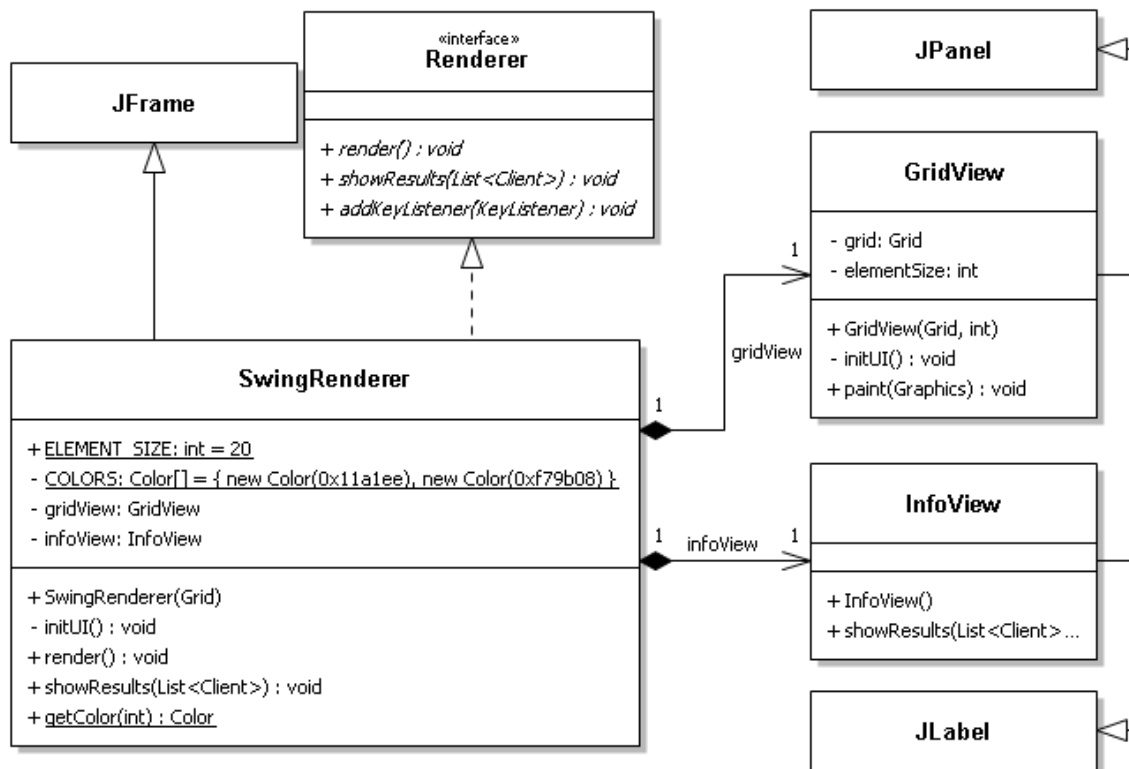
6.8. Játéktér megjelenítése



6.8.1. ábra. Játéktér megjelenítése: bal oldalt konzolos, jobb oldalt grafikus módon

A játéktér legegyszerűbb megjelenítését a *Grid* osztály *toString()* metódusa valósítja meg, amely minden padlót szóközzel, egyéb elemeket pedig a saját *toString()* metódusának visszatérési értékével reprezentál (ezek az egyszerűség kedvéért egyetlen karakter hosszú sztringgel térnek vissza). Az akadályokat „#”-el, a játékosokat az *id*-jukkal, az egymásnak ütközést pedig „x”-el jelöljük (6.8.1. ábra, bal oldal).

Ennél bonyolultabb megjelenítés a *Renderer* interfészen keresztül érhető el, amelyet a *SwingRenderer* osztály valósít meg (6.8.1. ábra, jobb oldal). Két konstanst is tartalmaz: az egyik ilyen az *ELEMENT_SIZE*, amely a blokkok magasságát és szélességét szabja meg pixelben, a másik a *COLORS* tömb, amely a kliensek színeit tartalmazza (*id* – 1 indexen érhető el a sajátja, ezt a *getColor()* metódus kezeli). A képernyő két részből áll: a játéktér, és egy információs sáv. Előbbit a *GridView*, utóbbit az *InfoView* valósítja meg, és mindkét osztályt csak a *SwingRenderer* használja (6.8.2. ábra).



6.8.2. ábra. A Swing-et használó játéktér megjelenítés felépítése

Az információs sáv a `JLabel` leszármazottja, alapértelmezetten fehér „Game started” felirattal indul, majd a játék végén az eredmény jelzésére szolgál: döntetlen esetén „Tie”, különben a nyertes színével írva a neve és egy „won” szuffix. A `GridView` a `JPanel` leszármazottja. Példányosításkor megragadja a fókuszot, így nem kell az ablakba kattintani mikor elkezdődik a játék hogy tudja irányítani az emberi játékos a saját vonalát. Ezek a `KeyListener`-ek a játék indításakor rendelődnek hozzá az ablakhoz a `GridControllerWithView` `start()` metódusában. A `paint()` a `JPanel` azonos nevű metódusának felülírása, itt történik a tartalom kirajzolása. Először meghívja az őosztály azonos nevű függvényét, majd kirajzolja a háttér rácsot, amely 5 blokkonként vízszintes és függőleges irányban is sötétszürke csíkokat húz keresztül a pályán. Ezután az akadályok megjelenítése következik, amelyek szürke színt kapnak, minden szín komponens értéke 255 osztva az akadály magassága + 1-el. Koordinátaikat az elemek méretével szorozva kapjuk meg a rajzvászonon található helyük bal felső pontját, innen indítva töltünk ki egy elemméret széles és magas négyszöget (ezt használjuk a többi típusú elemnél is). Ezt a kliensek által húzott csíkok kirajzolása követi, amelyek színe a hozzájuk tartozó kliens *id*-ja alapján kérhető le a `SwingRenderer` `getColor()` metódusával. Ha az épp rajzolt csík elem a

kliens feje, akkor világosítunk a színen majd kirajzoljuk. Végül, ha összeütközéses döntetlen lett az eredmény, akkor annak a pozíciónak a blokkját kiszínezzük az első kliens fejének színével, majd a jobb felső és bal alsó negyedét a blokknak feltöltjük a második kliens fejszínével. Ez a `paint()` folyamat minden kör után lefut, ezt hívja meg a `Renderer` interfész egyik metódusát megvalósító `render()` a `repaint()`-en keresztül a `SwingRenderer` osztályban.

6.9. Kliensek



6.9.1. ábra. Kliens osztályok hierarchiája, és az irányváltást kezelő metódus

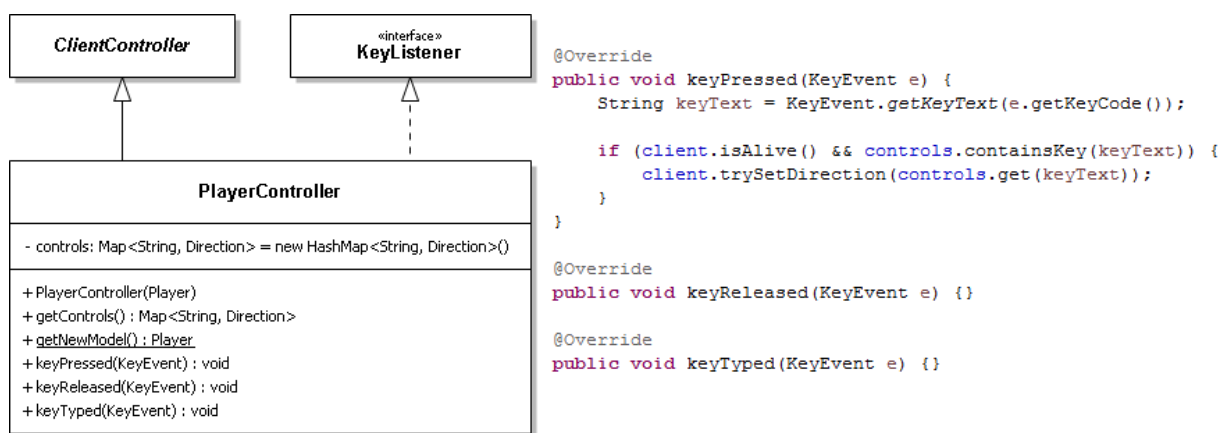
A játékban két fajta kliens lehet: **AI** (gépi játékos) és **Player** (emberi játékos). A **Client** absztrakt osztály ezeknek a típusoknak az őse, amely összegyűjti közös tulajdonságaikat (6.9.1. ábra). A *lastDirection* és *nextDirection* változókkal tudjuk elkerülni, hogy egy kliens véletlenül megölje magát azzal, hogy az eddigi haladási irányával ellentétes irányba próbál fordulni, ezt az ellenőrzést a `trySetDirection()` metódus valósítja meg. A **Player** osztály látszólag nem látja el a **Client**-et újabb funkciókkal, azonban a kliens típusok kezeléseinek különbségeiből adódóan határozottan meg kell hogy tudjuk különböztetni az emberi játékost a gépitől, ez a fő szerepe. Az **AI** osztályt a **RandomAIController**-el irányítjuk, míg a **MinimaxAI**-t a **MinimaxAIController**-rel. Utóbbi *level* adattagja a minimax játékfa maximális mélységét tárolja, minél nagyobb várhatóan annál jobb döntést tud hozni (azonban a számítási ideje is nő vele), *heuristic* adattagja pedig a minimax

algoritmushoz használt heurisztikát. Részletesebben a gépi játékosok kezelésénél ismertetem ezeket.

6.10. Kliensek kezelésének alapja

A `ClientController` absztrakt osztály adja meg a kliensek kezelésének alapjait. A `nextId` statikus adattagja tárolja a soron következő `id`-t, mindig ezt használjuk új kliens példányosításánál. Legfontosabb metódusa a `step()`, amely lépteti a példánynak megadott klienst.

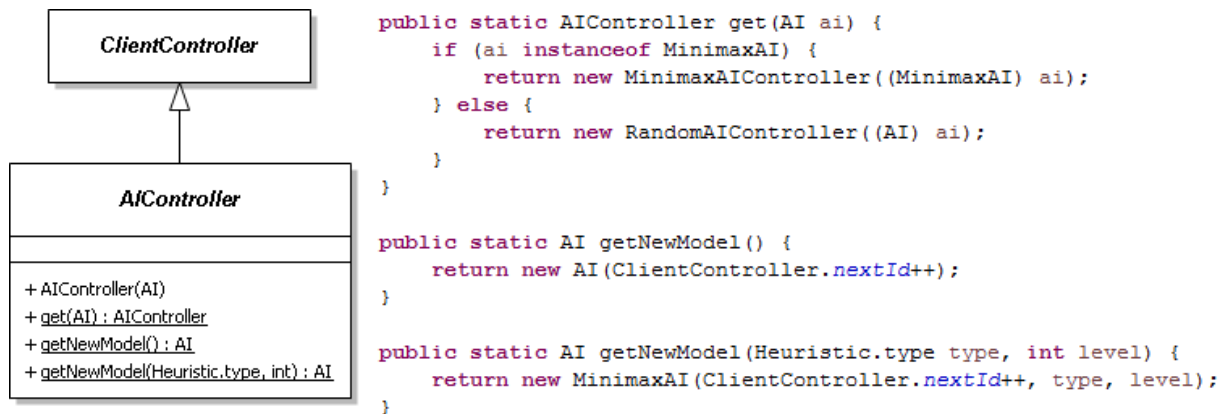
6.11. Emberi játékosok kezelése



6.11.1. ábra. A `PlayerController` osztálydiagramja és a `KeyListener` metódusainak megvalósításai

Az emberi játékosok irányítását a `PlayerController` osztály kezeli (6.11.1. ábra). Ezek a kliensek billentyűzettel irányíthatóak, amely megvalósításához a `KeyListener` interfészt használja a Swing. Három megvalósítandó metódust tartalmaz: `keyPressed()` (egy billentyű le lett nyomva), `keyReleased()` (egy billentyű fel lett engedve) és `keyTyped()` (be lett írva egy karakter). Ezek közül a `PlayerController`-ben egyedül a `keyPressed()` metódus tartalmaz kódot. Ez lekéri a `keyText` változóba a leütött billentyű `String` reprezentációját (pl. „W”, „Up”), és ha a kliens életben van, valamint az irányítását tartalmazó `controls` `HashMap<String, Direction>` adattag tartalmazza a `keyText` indexet, akkor az adott indexszel jelölt irányba próbálja meg fordítani a játékos által irányított klienst (6.11.1. ábra, jobb oldal). A `controls`-t a konstruktor tölti fel értékekkel, amely 1-es játékos esetén a nyilakat felelteti meg irányoknak, 2-es esetén pedig a WASD billentyűket értelemszerűen.

6.12. Gépi játékosok kezelésének alapja



6.12.1. ábra. AIController osztálydiagramja és metódusainak megvalósítása

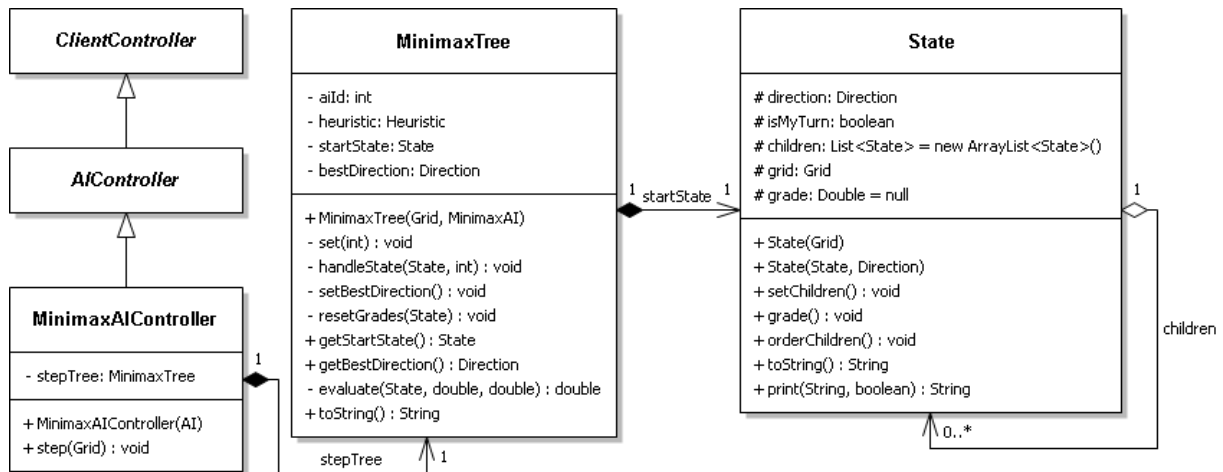
A gépi játékosok őssztálya az absztrakt AIController, amely a ClientController leszármazottja (6.12.1. ábra). Két statikus metódusa van: az egyik a `get()`, amely visszaadja az adott AI modellhez tartozó vezérlő osztályt. Ez `MinimaxAI` példány esetén `MinimaxAIController` lesz, különben `RandomAIController`. A másik statikus metódus a `getNewModel()`, amely két paraméterlistával is elérhető: paraméterek nélkül AI példányt ad vissza, típus és szint paraméterekkel pedig `MinimaxAI`-t. Az `id` (ugyanúgy ahogy az emberi játékos modellje esetében is) a `ClientController` `nextId` adattagjának értéke lesz, ezután léptetjük.

6.13. Random gépi játékos kezelése

A random gépi játékos a már ismertetett Random AI megvalósítása. Kezelését a `RandomAIController` osztály végzi, amely az `AIController` leszármazottja, egyetlen új adattagja egy Random generátor. A léptetést végző `step()` metódusában lekéri a jelenlegi pozícióból léphető irányokat, ha ez nem üres, akkor választ közülük véletlenszerűen egyet, különben nem változtat a haladási irányán (azaz egy zsákutca végét elérte, nincs jó lépés).

6.14. Minimax algoritmust használó gépi játékos kezelése

A már korábban ismertetett Minimax AI kezelését a `MinimaxAIController` osztály végzi, a minimax algoritmus fáját pedig a `State` típusú csúcsokat tartalmazó `MinimaxTree` valósítja meg (6.14.1. ábra).



6.14.1. ábra. Minimax algoritmuson alapuló gépi játékos kezeléséhez használt osztályok diagramja

A **MinimaxAIController** **step()** metódusa minden körben újrapéldányosítja a **MinimaxTree**-t a jelenlegi játéktérrel és a klienssel, majd a fa által kiválasztott legjobb irányba állítja a klienst. A **MinimaxTree** példányosításakor létrehozuk a kezdőállapotot (**startState** adattag), és elkészítjük a belőle kiinduló fát a megadott mélységig. Egy csúcs részfájának regenerálását a **handleState()** metódus végzi rekurzívan, amely két paramétert vár: a részfa gyökércsúcsát és a maximális mélységet (6.14.2. ábra, bal oldal). Első lépésként megnézi, hogy elértük-e a megadott mélységet: ha igen, akkor osztályozzuk a gyökércsúcsot, és megszakítjuk a futást. Ellenkező esetben beállítjuk az innen elérhető állapotokat, majd megvizsgáljuk, hogy létezik-e ilyen. Ha nem (végállapotba értünk), akkor osztályozza a gyökércsúcsot (hiszen levél lett a teljes fában), különben pedig az összes gyermekcsúcsra is meghívjuk a metódust egyel kisebb mélységgel. Ha elkészült a teljes fa, akkor kiértékeljük az **evaluate()** rekurzív metódussal, amely az alfa-béta vágás algoritmus megvalósítása egy kis bővítéssel, ugyanis nem csak visszatér a talált értékkel, hanem be is állítja azt az épp vizsgált csúcson (6.14.2. ábra, jobb oldal). Ezután beállítjuk a **bestDirection** adattagot abba az irányba, amely felé haladva a gyökér gyermekcsúcsai közül a maximális értékű állapotot találjuk (ugyanis a minimax fa egyezményesen mindig maximalizálással kezd).

```

private void handleState(State state, int limit) {
    if (limit <= 0) {
        state.grade();
        return;
    }

    state.setChildren();

    if (state.children.isEmpty()) {
        state.grade();
    } else {
        Iterator<State> it = state.children.iterator();
        while (it.hasNext()) {
            handleState(it.next(), limit - 1);
        }
    }
}

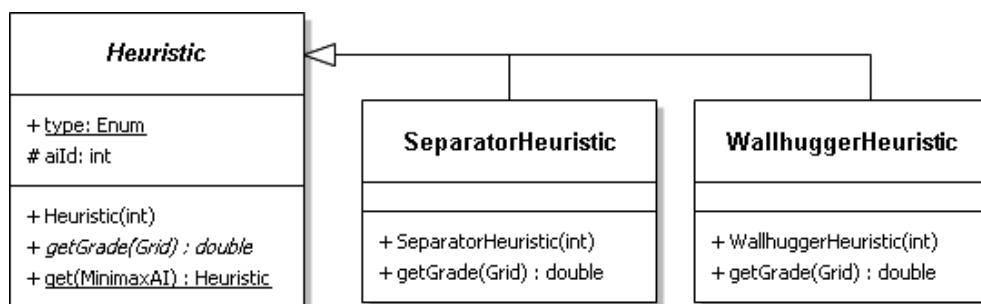
private double evaluate(State state, double alpha, double beta) {
    if (state.grade != null) {
        return state.grade;
    }
    if (state.isMyTurn) {
        for (State child : state.children) {
            alpha = Math.max(alpha, evaluate(child, alpha, beta));
            if (beta <= alpha) {
                state.grade = beta;
                return beta;
            }
        }
        state.grade = alpha;
        return alpha;
    } else {
        for (State child : state.children) {
            beta = Math.min(beta, evaluate(child, alpha, beta));
            if (beta <= alpha) {
                state.grade = alpha;
                return alpha;
            }
        }
        state.grade = beta;
        return beta;
    }
}

```

6.14.2. ábra. Adott csúcsból kiinduló részfa elkészítése (bal oldal) és a fa kiértékelése (jobb oldal)

A nem levél csúcsok értékelésére több algoritmust is kipróbáltam, végül az alfa-béta vágás mellett döntöttem. A minimax és az alfa-béta vágás algoritmusok összehasonlításánál azt tapasztaltam, hogy nem ért el jelentős javulást a futási idő, majd amikor kipróbáltam az iteratív mélyülő mélységi kereséssel javítani az alfa-béta vágás hatékonyságát, a futási idő jelentős mértékben romlott. Ezt később oda tudtam visszavezetni, hogy a fa felépítése és értékelése során a legtöbb idő (Separator heurisztikával alfa-béta vágásnál általánosan ~80%) a levelek értékelésekor, a kliens pozíciójából elérhető padlók megkeresésére megy el, amelyet mélységi gráf bejárás algoritmussal valósítottam meg. Mivel az iteratív mélységi keresésnél többször is ki kell értékelni a leveleket, így emiatt romlott a futási idő.

6.15. Heurisztikák



6.15.1. ábra. Heurisztikák osztályhierarchiája

A heurisztikák absztrakt őssztálya a *Heuristic*, amely *type* felsorolása tartalmazza a lehetséges heurisztika típusokat, és a *get()* statikus metódusa példányosítja egy adott MinimaxAI-hoz (6.15.1. ábra). Példányainak egyetlen metódusa a *getGrade()*, amelyet a

gyermekosztályok valósítanak meg, és egy játéktér állás értékét adja meg (ezt használja a `State` osztály `grade()` metódusa). Mindkét gyermekosztály a nevükben található heurisztikát valósítja meg, amelyeket a 3.3 Minimax AI fejezetben ismertettem.

7. Teszteredmények

7.1. Tesztbeállítások

Minden tesztet 30*20-as játéktéren futtattam 5%-os akadály aránnyal, amelyet körönként újragenerált a program. Minden naplófájl tartalmazza a végeredmények statisztikáját, az átlagos időt, amely egy lépés kiszámításához volt szükséges milliszekundumban, valamint a körök átlagos hosszát (7.1.1. ábra).

```

1 Ties: 3
2 AI#1 won 53 times
3 AI#2 won 44 times
4
5 Average step times:
6 AI#1: 0 ms
7 AI#2: 0 ms
8
9 Average number of steps per round: 24

```

7.1.1. ábra. Naplófájl két random gépi játékos 100 játszmájáról

7.2. Szimulációs módban futtatott tesztek

Az első tesztekben a random játékost játszottam maga, és a két minimax AI 2-es szintje ellen. Azonos gépi játékos ellen 56-42 (2) eredményt értek el 100 játék alatt, átlagosan 23 lépés hosszú köröket játszottak, a következő lépés kiszámítása átlagban pedig kevesebb, mint 1 ms időt vett igénybe. Ez után a 2-es szintű minimax játékosok következtek, először a Separator heurisztikával, amely 91-0 (9)-re nyerte a 100 kört, majd a Wallhugger-rel, amely 97-2 (1)-re győzte le a random játékost. Ezekből az eredményekből jól látszik, hogy a random AI gyorsan dönt, de nem túl erős ellenfél.

Ezek után a minimax gépi játékosok heurisztikáit játszottam azonos típusú ellenfelekkel, 6-os szintig 100 körben, felette csak 10-ben (ugyanis itt már nagyon sokáig futottak a tesztek, éppen ezért nem is vettem őket figyelembe az elemzésnél). A Separator heurisztika statisztikáit a következő táblázatban foglaltam össze:

Szint	2	4	6	8	10
Eredmény	9-14 (77)	43-26 (31)	35-25 (40)	6-2 (2)	3-0 (7)
Átlagos számítási idő (ms)	2	18,5	103	622,5	4121
Átlagos lépésszám	51	104	91	102	102

Jól látszik, hogy a szintekkel együtt nő a számítási idő, az átlagos lépésszám viszont 2-es szint felett stagnál. Érdekes megfigyelni, hogy nem mindig maradt 50% körüli a nyerési arány, ez valószínűleg a random generált pályáknak és a viszonylag kis futási számnak köszönhető. Az átlagos lépésszámból nem látszik, hogy jobb döntést tudnának hozni a nagyobb szintű gépi játékosok, de lefuttattam a teszteket 2-6 szintűekkel. Az eredményeiket egy táblázatban foglaltam össze, ahol a cellák értéke az oszlopban jelölt szintű játékos nyerési aránya a sorban jelölt játékos ellen, százalékban megadva:

Szint	2	4	6
2	-	66	58
4	9	-	20
6	10	38	-

A 2-es szintet nagy fölényrel legyőzték a nagyobb szintűek, de a 4-es és 6-os közti eredmény a várakozásokkal ellentétes lett.

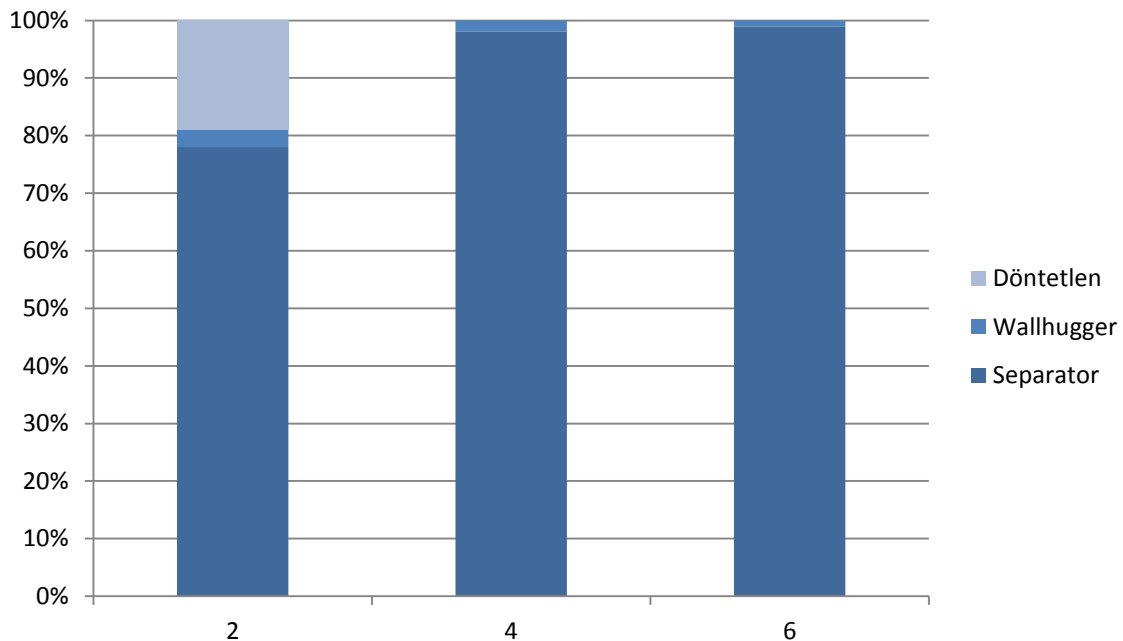
Ezek után elvégeztem ugyanezeket a teszteket a Wallhugger heurisztikával is:

Szint	2	4	6	8	10
Eredmény	40-44 (16)	54-44 (2)	42-58 (0)	7-3 (0)	6-4 (0)
Átlagos számítási idő (ms)	5	25	119,5	1296	2621,5
Átlagos lépésszám	109	159	160	159	157

Szint	2	4	6
2	-	70	70
4	28	-	58
6	28	42	-

A számítási idő és az átlagos lépésszám viselkedése hasonló mintát mutat a Separator heurisztikához, és a különböző szintek összemérésének eredménye is nagyon hasonló hozzá, a szint emelése egyel 2-esről többet javított, mint 4-esről.

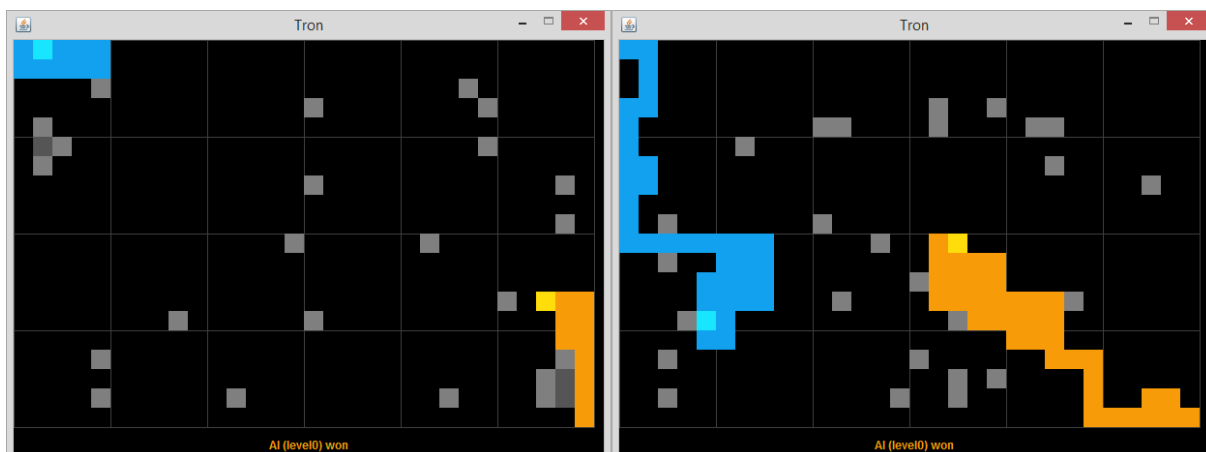
Ezek után összemértem a különböző heurisztikákat azonos szinten:



A szintek változtatásával nem változott jelentősen az eredmény, minden esetben a Separator jóval magasabb győzelmi aránnyal rendelkezett, a Wallhugger-ek alig-alig tudtak győzni, döntetlent pedig csak 2-es szint ellen tudtak elérni. Érdekességgépp futtattam 2 tesztet, ahol a 2-es Separator ellen 4-es és 6-os Wallhugger-t állítottam be, az eredmény 82-13 (5) és 84-12 (4) lett, tehát a szint emelése után sem tudja megközelíteni a Separator-t.

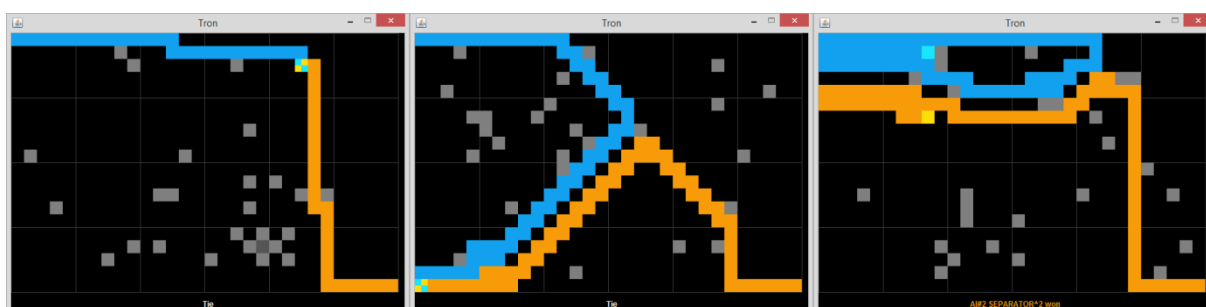
7.3. Grafikus módban futtatott tesztek

Grafikus módban is játszottam gépi játékosokat azonos szintű ellenféllel, majd emberi játékos ellen is, ezek a tesztek megfigyeléseit gyűjtöttem itt össze. A random gépi játékos általában saját magát csalja zsákutcába, ez magyarázza az alacsony átlagos lépésszámát (7.3.1. ábra).



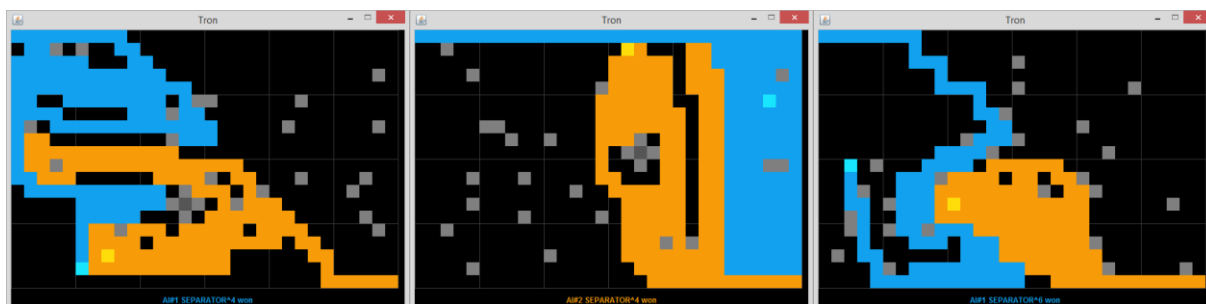
7.3.1. ábra. A random gépi játékos zsákutcába fordulása

A Separator heurisztika 2-es szinten általában a döntetlent látja a legjobb megoldásnak, de mint a jobb oldali képen is látszik, vannak helyzetek, amikor jól el tudja különíteni magát (7.3.2. ábra). A döntetlenek magas száma egyrészt annak köszönhető, hogy csak 1-1 lépéssel tud előre „gondolkodni”, másrészt pedig annak, hogy a cikk-cakkban kettéválasztott területen számolt általa elérhető – el nem érhető területek számához képest a döntetlen jobb döntésnek tűnik.



7.3.2. ábra. Separator 2-es szinten

Nagyobb szinteken már jóval ritkábbak a döntetlenek, és a szeparálás is jobban látszódik a kliensek mozgásán, van, hogy látványosan üldözi az ellenfelet, hogy minél kevesebb területet hagyjon neki (7.3.3. ábra).



7.3.3. ábra. Separator 2-es szint felett (bal és középső képen 4-es, jobb oldalin 6-os szint)

A Wallhugger egy sokkal konfliktuskerülőbb heurisztika, ez magyarázza a magasabb átlagos lépésszámot. Érdekes megfigyelni, hogy egy szabad területet nem választ le ha nincs rákényszerítve, így később vissza tud oda térni, ez az elérhető szabad padlók számának figyelembe vételének köszönhető (7.3.4. ábra).



7.3.4. ábra. Wallhugger heurisztika 2-es, 4-es és 6-os szinten

A két heurisztika egymás elleni játékán jól látható a különböző taktikájuk: a Separator igyekszik minél nagyobb területet leválasztani magának, a Wallhugger pedig a fal mellett halad, ebből kifolyólag a Separator szorosan mellette haladva el tudja választani a játéktér nagyobb részét, megnyerve ezzel a játékot (7.3.5. ábra).



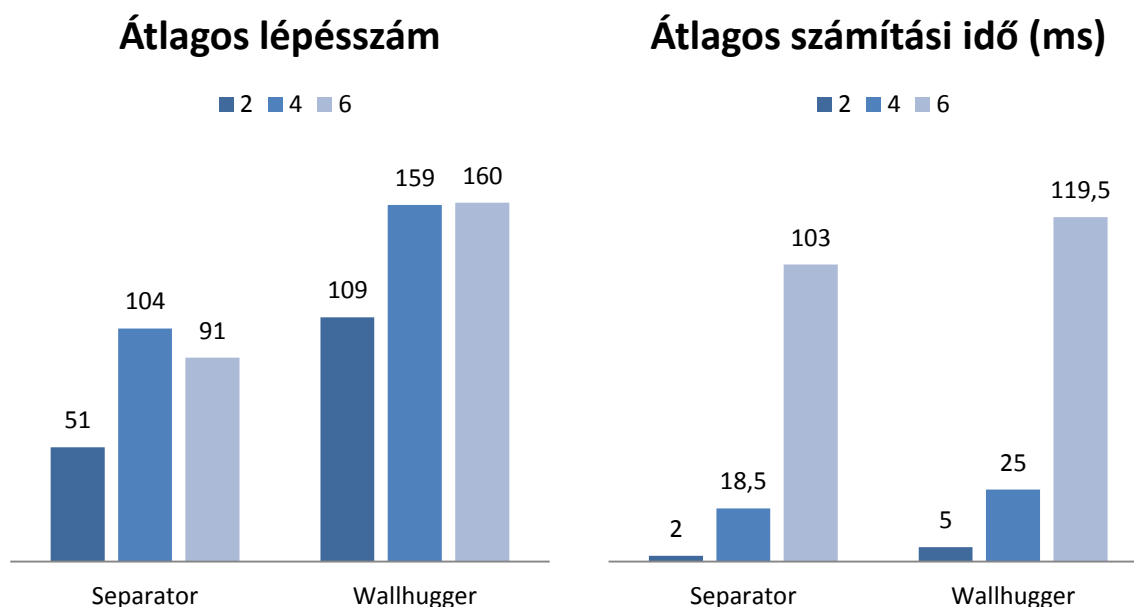
7.3.5. ábra. A két heurisztika egymás ellen (2-es, 4-es és 6-os szinten)

Ezek után következtek az emberi játékos ellen a különböző gépi játékosok. A random itt sem okozott meglepetést, általában nem kihívás legyőzni. A Separator AI-k ellen játszott meccsek alatt megfigyeltem, hogy a végeredmény azon múlik, hogy a játék elején ki szerzi meg a nagyobb területet. Ha ez sikerült az AI-nak, akkor nem hibázott és megnyerte a játszmát, de amikor nekem sikerült, én többször is vétettem hibát (később vagy hamarabb próbáltam kanyarodni, mint kellett volna). A Wallhugger AI-k nem jelentenek nagy kihívást emberi játékos ellen, ugyanis csak a pálya szélén elkezdik feltölteni a területet, közben könnyen elzárható ellene a pálya nagyobbik fele, amelyen ha nem hibázik a játékos, akkor könnyű győzelem a játék vége.

8. Konklúzió

Több érdekes konklúzió is levonható a szakdolgozatból, az első számomra az volt, hogy egyes algoritmusok hiába tűnnek elméletben jónak, a gyakorlatban ezt sok tényező is keresztülhúzhatja. Erre az első példa a generált pálya érvényességének ellenőrzése volt, amelyre azt gondoltam, hogy gyorsabb lesz, ha minden ledobott akadály után teszteljük (így mindig csak egy pályát kell generálni), mint ha a végén, és hibás esetén újra generáljuk. Miután teszteltem a 30*20-as, 5%-os akadályaránnal rendelkező pályák generálását, a fordítottja igazolódott be. Ezen felül a minimax algoritmus optimalizálására használt alfa-béta vágásról is kiderült, hogy a jelenlegi helyzetben nem hozott jelentős javulást, mélységi kereséssel és a csúcsok rendezésével kombinálva pedig egyenesen romlott a futási idő az erőforrásigényes csúcs kiértékelés miatt.

A másik eredmény, amire jutottam az, hogy a minimax algoritmuson alapuló gépi játékosok teljesítménye a mélység növelésével nem növekszik drasztikusan, a futási idejük viszont igen. Ez az azonos szintű és heurisztikával rendelkező ellenfelekkel futtatott tesztek eredményeiből készített diagramokról könnyen leolvasható:



Emberi játékosok ellen futtatott tesztek alatt kiderült, hogy pár játék után kiismerhetővé váltak (és így legyőzhetővé) a minimax algoritmusra épülő gépi játékosok, de az emberi hibalehetőségek miatt így is jó ellenfélnek bizonyult a Separator heurisztikával. A random AI nevének megfelelően kiszámíthatatlanul viselkedett, de a gyakori zsákutcába fordulása miatt gyenge ellenfél.

9. Fejlesztési lehetőségek

A játék több szempontból is tovább fejleszthető. GUI szempontjából egy felhasználóbarát menü és szebb textúrák sokat javítanának a megjelenésén. A gépi játékosok közül a random AI eredményei javíthatóak lennének, ha minden fordulás előtt megszámolná hány elérhető padló marad előtte, de ez egy erőforrásigényes művelet, és szerettem volna egy egyszerű és gyors algoritmussal összehasonlítani a jóval kifinomultabb minimax AI-t.

Irodalomjegyzék

- [1] Alon Itai, Christos H. Papadimitrou and Jayme Luiz Szwarcifter: Hamilton paths in Grid Graphs, Siam Journal on Computing Vol 11. No 4, November 1982, pp 676-686
- [2] Farkas Gábor - Amit tudnod kell fejlesztőként, IV. rész: Verziókezelés
http://ithub.hu/blog/post/Amit_tudnod_kell_fejlesztokent_IV_resz_Verziokezeles/
- [3] Google AI Challenge (Tron)
<http://tron.aichallenge.org/>
- [4] Házy Attila, Nagy Ferenc – Adatstruktúrák és algoritmusok (2009)
http://www.tankonyvtar.hu/hu/tartalom/tamop425/0046_adatstrukturak_es_algoritmusok/ch08.html
- [5] Jelasity Márk - Mesterséges Intelligencia I. előadásjegyzet (vázlat), 2011. január 23.
<http://www.inf.u-szeged.hu/~jelasity/mi1/2010/jegyzet.pdf>
- [6] M. R. Garey, D. S. Johnson and R. Endre Tarjani: The planar Hamiltonian-circuit problem is NP-complete, Siam Journal on Computing Vol 5. No. 4, December 1976, pp 704-714
- [7] Michael R. Garey and David S. Johnson (1979), Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman, ISBN 0-7167-1045-5 A1.3: GT37–39, pp. 199–200.
- [8] Pablo de Oliveira Castro- A simple Tron bot, 2013
<http://www.sifflez.org/misc/tronbot/index.html>
- [9] Sun Microsystems - Code Conventions for the Java TM Programming Language, April 20, 1999
<http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html>
- [10] Wikipédia - Model-view-controller
<https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller>
- [11] Wikipédia - Swing (Java)
[https://en.wikipedia.org/wiki/Swing_\(Java\)](https://en.wikipedia.org/wiki/Swing_(Java))

Nyilatkozat

Alulírott Bajzáth Dávid, programtervező informatikus szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Tanszékcsoport Számítástudományi Tanszékén készítettem, programtervező informatikus BSc diploma megszerzése érdekében. Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel. Tudomásul veszem, hogy szakdolgozatomat a Szegedi Tudományegyetem Informatikai Tanszékcsoport könyvtárában, a helyben olvasható könyvek között helyezik el.

2015. 12. 01.

Aláírás