Szakdolgozat

Miskolci Egyetem

Nim játék nyerő stratégiájának bemutatása példa programmal

Készítette:

Pozsgay Máté

Programtervező Informatikus

Témavezető:

Körei Attila

Orova Péter

Miskolc, 2017

Miskolci Egyetem
Gépészmérnöki és Informatikai Kar
Alkalmazott Matematikai Tanszék **Szám:**

Szakdolgozat Feladat

Pozsgay Máté (PW72KS) programtervező informatikus jelölt részére.

A szakdolgozat tárgyköre: Logikai játékok nyerő stratégiáinak keresése

A szakdolgozat címe: Nim játék nyerő stratégiájának bemutatása példaprogrammal

A feladat részletezése:
Kétszemélyes, teljes információjú játékok vizsgálata. A játékfa bejárására, elemzésére, levágására alkalmas módszerek bemutatása (minimax tétel, alfabéta algoritmus). A nyerő stratégia létezésének kérdése. Northcott-sakkot (Nim játék egy variánsa) játszó program elkészítése, tesztelése.

Témavezető(k): Dr. Körei Attila egyetemi docens

Konzulens(ek): Orova Péter szoftvermérnök

A feladat kiadásának ideje: 2017. február 15.

. . . .
szakfelelős

Eredetiségi Nyilatkozat

Alulírott **Pozsgay Máté**; Neptun-kód: **PW72KS** a Miskolci Egyetem Gépészmérnöki és Informatikai Karának végzős **programtervező informatikus** szakos hallgatója ezennel büntetőjogi és fegyelmi felelősségem tudatában nyilatkozom és aláírásommal igazolom, hogy **Nim játék nyerő stratégiájának bemutatása példa programmal** című szakdolgozatom/diplomatervem saját, önálló munkám; az abban hivatkozott szakirodalom felhasználása a forráskezelés szabályai szerint történt.

Tudomásul veszem, hogy szakdolgozat esetén plágiumnak számít:

- szószerinti idézet közlése idézőjel és hivatkozás megjelölése nélkül;
- tartalmi idézet hivatkozás megjelölése nélkül;
- más publikált gondolatainak saját gondolatként való feltüntetése.

Alulírott kijelentem, hogy a plágium fogalmát megismertem, és tudomásul veszem, hogy plágium esetén szakdolgozatom visszautasításra kerül.

Miskolc, ...év . . . . .hó . . . . .nap

. . . .

Hallgató

1.

szükséges (módosítás külön lapon)

A szakdolgozat feladat módosítása

nem szükséges

....

dátum témavezető(k)

2. A feladat kidolgozását ellenőriztem:

témavezető (dátum, aláírás): konzulens (dátum, aláírás):

....

....

....

3. A szakdolgozat beadható:

....

dátum témavezető(k)

4. A szakdolgozat .... szövegoldalt

.... program protokollt (listát, felhasználói leírást)

.... elektronikus adathordozót (részletezve)

....

.... egyéb mellékletet (részletezve)

....

tartalmaz.

....

dátum témavezető(k)

5.

bocsátható

A szakdolgozat bírálatra

nem bocsátható

A bíráló neve: ....

....

dátum szakfelelős

6. A szakdolgozat osztályzata

a témavezető javaslata: ....

a bíráló javaslata: ....

a szakdolgozat végleges eredménye: ....

Miskolc, ...

a Záróvizsga Bizottság Elnöke

Tartalomjegyzék

1. fejezet

Bevezetés

Az emberiséget már jó ideje foglalkoztatja, hogy megértse saját gondolkodásának a működését, hogy megismerje hogyan gondolkodik, miként rendszerezi, és használja fel a megszerzett tudást. Bár a mesterséges intelligencia - mint különálló tudományág - viszonylag fiatal, mégis az utóbbi idők technológiai fejlődése tette igazán lehetővé ennek a tudománynak a gyakorlati felhasználását.

A számítógép megjelenése volt az, ami életre hívta ezt a tudományágat, hiszen lehetővé tette az embernek, hogy önmagától elvonatkoztatva, egy különálló entitáson vizsgálja a gondolkodás tudományát. Az utóbbi idők robbanásszerű fejlődése - olyan tudományterületeken, mint például biológia, elektronika, matematika - nem csak intelligens programok megírását tette lehetővé, hanem a gépek tároló, és feldolgozó kapacitásának ugrásszerű növekedése elérhető közelségbe hozta az egyre valósághűbb, intelligens ágensek elkészítését, és azok futtatását.

A Mesterséges Intelligencia egy olyan tudományág amivel nem csak érdemes foglalkozni, hanem szükségszerű is. Legújabb korunkat megfigyelve észrevehető az a tendencia, hogy az idő múlásával egyre több helyen, és egyre nagyobb mértékben hagyatkozunk a gépek segítségére, a gépek által elvégzett munkára. Jelenleg nincs olyan jel, ami arra utalna, hogy ez a jelenség megváltozik a belátható jövőben. Amennyiben ez igaz, és továbbra is ebbe az irányba haladnak a dolgok, akkor valószínű, hogy egy idő után az ember nem lesz képes a számítógépes rendszerek vezérlésére, mikromenedzselésére, és ezt a feladatkört sem ember fogja betölteni, ezt is rá kell bízni a gépekre, mégpedig egy intelligens szoftverre.

Erre a jelenségre már ma is sok példa áll rendelkezésre. Vegyük például a ma egyre inkább népszerű elképzelést, az IoT-t, vagyis a dolgok internetjét. Ezen elképzelés szerint a (közeli) jövőben a hétköznapi használati tárgyaink jelentős része (óra, mérleg, telefon, ruhaszekrény, gépjárművek, lámpák) átalakul "okos" eszközzé, amelyek egymással kapcsolatban állnak, kommunikálnak. Ekkora mennyiségű kommunikációra viszont nagyon is nehéz felkészíteni a telekommunikációs infrastruktúrát, illetve sok esetben nem is lehet. Gondoljunk csak abba bele, hogy az emberek (okos eszközeikkel együtt) folyamatos mozogásban vannak. Ingáznak munkába, rendezvényekre mennek, utaznak, egyszóval élnek, ami az infrastruktúrát nem egyenletesen terheli. Sokszor bizonyos részeire hirtelen nagy mértékű terhelést ad, amire egy ember képtelen megfelelő gyorsasággal és hatékonysággal reagálni. Erre a problémára fog megoldást nyújtani a

SDN (Software Defined Network), amely egy olyan komplex hálózati megoldás, ami folyamatosan monitorozza a telekommunikációs infrastruktúrát, az esetlegesen bekövetkező váratlan eseményre reagál, és megfelelően helyes döntést hozva a rendszer túlterhelt részeit további erőforrásokkal megtámogatva tehermentesíti.

A mesterséges intelligenciáknak gyakorlati felhasználásának csak a képzelet szab határt, és talán - gyakorlatias ember lévén - éppen ezért foglalkoztat engem is a téma. Régebben már ugyan készítettem kétszemélyes játékot, amibe gépi játékost is terveztem, de akkor még nem jutottam el odáig, hogy ezt meg is valósítsam. Most egy igen egyszerű játékkal, a Nim-mel ezen régi adósságomat - önmagammal szemben is - törlesztem.

2. fejezet

Kétszemélyes logikai játékok és a Nim játék

2.1. A mesterséges intelligencia

A mesterséges intelligencia egy viszonylag új, ám hatalmas lehetőségeket rejtő tudományág. A "Mesterséges Intelligencia - Modern megközelítésben" ([4]) könyv négy kategóriába sorolja a mesterséges intelligencia mibenlétét, ezen belül is kettő csoportba, amely teljesen más szemszögből vizsgálja a problémát.

A megközelítések egyik csoportja az emberközpontú viselkedés, ahol az mesterséges intelligenciától azt várjuk, hogy a lehető legjobban hasonlítson az emberre. Utánozza az emberi gondolkodást az elképzelhető legjobb módon, és cselekvése - amennyire lehet - emberi viselkedésre hasonlítson. A másik megközelítés a racionalitást helyezi középpontba, azaz a mesterséges intelligenciának a rendelkezésre álló információk alapján a lehető legjobb (helyes) döntéseket kell hoznia, és a lehetőségeihez képest a legjobb módon is kell azt végrehajtania.

Ez a két nézőpont teljesen másképpen írja körül a mesterséges intelligencia mibenlétét, ennek megfelelően más módszereket is használ a vizsgálatukra. Matematikai és mérnöki megközelítésben a racionális irányzatok azok amelyek megfelelően jól vizsgálhatóak ezen tudományok eszköztárával.

2.1.1. Emberi módon gondolkodni

Ez az irányzat azt kutatja, hogy hogyan működik az emberi elme, miképpen gondolkodik az ember. Ennek kutatására leginkább két módszert használnak: az önelemzést és a pszichológiai kísérleteket.

A mesterséges intelligencia modellezését és a pszichológiai kísérleteket a kognitív tudomány (más nevén megismeréstudomány) kapcsolja össze. Mivel a kognitív tudomány rendkívül sok, egymástól távol eső tudományterületet foglal magába, ezért célszerűbb kognitív szemléletként hivatkozni rá, mintsem önálló tudományterületként. Foglalkozik - többek között - biológiával, nyelvészettel, számítástechnikával, pszichológiával, és minden olyan tudományággal, ami valamilyen módon a megismerési folyamatokat kutatja.

Az utóbbi időkben az orvostudomány fejlődésével jobban megismerhettük az agy, illetve a benne található idegsejtek működését, ez inspirálta a mesterséges intelligencia kutatóit a neurális hálók létrehozására és használatára, ami nagyot lendített a természetes nyelvfeldolgozás területén és a gépi látás fejlődésében.

2.1.2. Emberi módon cselekedni

A mesterséges intelligencia - mint tudományág - kialakulásának kezdetén, 1950-ben Alan Turing brit matematikus tett egy javaslatot, amivel igazolni lehet egy gépről, hogy intelligensen gondolkodik oly módon, hogy egy szintén intelligens lényhez (ember) mérik összehasonlíthatatlanságukat. Ez afféle munkadefiníció volt, nem kísérelte meg pontosan leírni, hogy egy intelligensen gondolkodó gépnek miféle képességekkel kelljen rendelkezni.

A Turing teszt három résztvevőből áll. Egy emberből, egy mesterséges intelligenciából és egy bírálóból, amelyik szintén ember. A résztvevők egymással írásban kommunikálnak, azonban ezen kívül semmilyen más információt nem ismerhetnek a többiekről. Nem láthatják és nem is hallhatják egymást. A megfigyelő kérdéseket tesz fel felváltva minkét résztvevőnek. Amennyiben a bíráló a feltett kérdéseire kapott válaszból nem tudja egyértelműen megállapítani, hogy melyik résztvevő ember, és melyik a gép, akkor a mesterséges intelligencia teljesítette a Turing-tesztet.

Jelenlegi tudásuk szerint egy a Turing-teszt sikeresen teljesítésére képes gép elkészítése nagyon összetett probléma, melynek a komplexitását a következő négy alapkövetelmény adja:

- Természetes nyelv feldolgozása: A bíráló a kérdéseit valamilyen ember által is beszélt természetes nyelven teszi fel. A kérdést a gépnek meg kell érteni, és ugyanezen a nyelven kell válaszolnia, ellenkező esetben azonnal lebuktatná magát.
- Tanulás: A gép képes kell, hogy legyen új információk elsajátítására, például a bíráló is adhat a gép számára új információt, melyre valamely módon visszakérdezhet.
- Tudás tárolása: A megszerzett tudást tárolnia kell oly módon, hogy ahhoz könnyen és gyorsan hozzá tudjon férni szükség esetnén.
- Automatizált következtetés: Kérdések megválaszolásához használja fel a megszerzett tudást, illetve vonjon le következtetéseket.

Látható, hogy a Turing-teszt kerülte a fizikai kontaktust a résztvevők között, hogy az a bíráló döntését ne befolyásolja, az intelligencia méréshez pedig nincs szükség fizikai kontaktusra. Létezik azonban a Turing-teszt egy kibővített változata, amely két további elemmel terjeszti ki a tesztet. Ebben a gépnek videó jelet is fel kell dolgoznia, a látottakra reagálnia kell, továbbá egy nyíláson keresztül tárgyakat átadhat át a bíráló, melyet a gépnek át kell tudni vennie (robotika).

A teljes Turing-teszt lefedi a mesterséges intelligencia mind a hat ágát, ugyanakkor érte néhány megfontolandó kritika: [7]

- Nem mindegyik ember képes teljesíteni a tesztet. Gondoljunk itt a valamilyen fogyatékossággal élő emberekre, vagy kisgyerekekre, holott más szempontok alapján ezek az emberek ugyanúgy intelligensnek tekinthetőek.
- Elképzelhető, hogy a teszten résztvevő ember megtagadja az együttműködést. Egy valóban emberként viselkedő gép szintén dönthet így, ezáltal megbukik a teszten, holott az együttműködés hiánya nem tekinthető az értelem hiányának.
- A beszélgetésfolyamat - jellegénél fogva - számos korláttal rendelkezik, ezáltal egy gép megfelelő nagyságú adatbázissal egyszerű mintaillesztéssel képes válaszolni a kérdésekre anélkül, hogy tényleges intelligenciával rendelkezne.

2.1.3. Racionálisan gondolkodni

A helyes gondolkodás tudományával már az ókorban is foglalkoztak, Arisztotelész görög filozófus elsőként kísérelt meg általános következtetési folyamatokat törvényekbe foglalni. Később ebből alakult ki a logika tudománya. Ezen ág követői azt remélik, hogy egy logikusan gondolkodó gép minden esetben logikus - azaz - helyes következtetésre jut, ezáltal cselekvése szintén logikus.

Az egyik legnagyobb probléma ezzel a megközelítéssel az, hogy az informális tudás a logika formális elemeket használó jelölésrendszerével nehezen kifejezhető, ráadásul sok esetben a tudás helyessége sem garantálható. A másik gond az, hogy egy probléma általában nagyon sok részből, ismeretlenből tevődik össze, amelyek túl nagy számítási igényt jelentenek.

2.1.4. Racionálisan cselekedni

Az intelligens ágens egyik legfontosabb tulajdonsága, hogy cselekszik, mégpedig olyan módon, hogy a rendelkezésére álló információkból a legjobb döntést hozza meg. A 2.1.3 részben tárgyalt helyes következtetés meghozása csupán az érem egyik oldala. Sokszor kell azonban cselekedni akkor is, ha nem lehetséges helyes következtetést hozni. Vagy azért mert nincsen, vagy pedig azért, mert egyszerűen túl sokáig tartana meghozni a helyes következtetést.

2.2. Kétszemélyes teljes információjú játékok

A kétszemélyes teljes információjú játékok a játékelmélet egy speciális ága. Egy olyan többágenses - egészen pontosan kettő - környezetben játszódik, ahol az ágensek egymás ellenségei, azaz versenyhelyzetben vannak, továbbá cselekvéseiket felváltva hajtják végre.

Általában zérus összegű játékokkal foglalkozunk, ami azt jelenti, hogy minden ágens minden cselekedetének hasznossága megegyezik az ellenfélével, csak ellenkező előjellel, tehát arra az előnyre, amit az egyik ágens egy lépésével megtesz, az a másik ágensnek ugyanannyira a hátrányára válik. Fontos megjegyezni, hogy minden kétszemélyes zérus összegű játékban létezik mindkét fél számára optimális stratégia.

Teljes információjúnak akkor hívunk egy játékot, ha a játék folyamán - kivéve esetleg az előkészületet - a játéktér minden tekintetben megismerhető az összes ágens számára. Ismerik a játékszabályokat, és az eddig megtett összes lépést.

Fontos még megemlíteni, hogy a játszma minden állásában véges - és előre ismert - lépés létezik, továbbá a játszma nem lehet végtelen, azaz véges számú lépés megtétele után az egyik játékos nyer, a másik pedig veszít (egyes esetekben a döntetlen végeredmény is elfogadott).

2.3. A játékfa

A kétszemélyes teljes információjú játékok esetében a következő lépés eldöntésének problémájához gyakran alkalmaznak játékfát. A játékfa - mint a neve is mutatja - egy fa adatszerkezet, ami leírja a játék lehetséges állapotait, illetve ezekben az állapotokban megléphető érvényes lépéseket.

A fa csomópontjai a lehetséges állapotok, míg az élek az átmenetet szimbolizálják. A fa szintjei gyakorlatilag egy-egy lépésnek felelnek meg, így a vizsgálatot is ennek megfelelően célszerű végezni.

Legtöbbször egy már megtalált állapotból nem lehet visszajutni ugyanabba az állapotba - pont ez adja a fa jellegét - ellenkező esetben a fa végtelen nagyságú - ebből következően megoldhatatlan - volna, továbbá sérülne a játék végességére vonatkozó kritérium. Ez a probléma főleg az olyan játékokat érinti, ahol a lépések visszavonhatóak, azaz az előzőleg megtett lépés ellentettje a rákövetkező lépésben érvényes lépésnek számít. Amennyiben ilyen előfordul, azt többnyire detektálni lehet, és az ilyen ágakat levágják, a fát ismét véges nagyságúra - ezáltal a problémát megoldhatóra - redukálva.

A megoldás keresése gyakorlatilag a játékfa felépítése és bejárása. Kezdésként tekintsük a kiinduló állapotot gyökérelemnek. Ezután megvizsgáljuk, hogy végállapotban van-e a játék, majd az összes lehetséges lépést véve egy új ágat hozunk létre, melynek a végére a lépés megtétele utáni állapotteret helyezzük. Egy csomópont további ágainak létrehozását kiterjesztésnek nevezzük. Hogy melyik ággal kezdjük a kiterjesztést azt a keresési stratégia határozza meg. Szükségszerűen a játék végállapotai nem kifejtett csomópontok (levélobjektumok) lesznek.

2.3.1. Minimax

Szó esett már arról, hogy miként lehetséges a játékfa felépítése, ám azt még nem részleteztem, hogy miként lehet kiválasztani a soron következő helyes lépést. Ideális esetben egy játékosnak a győzelemhez vezető út a lépések egy sorozata, ám ebbe az ellenfélnek is van beleszólása. Az optimális lépéssorozat kiválasztásában a Minimax (1.) algoritmust tudjuk segítségül hívni.

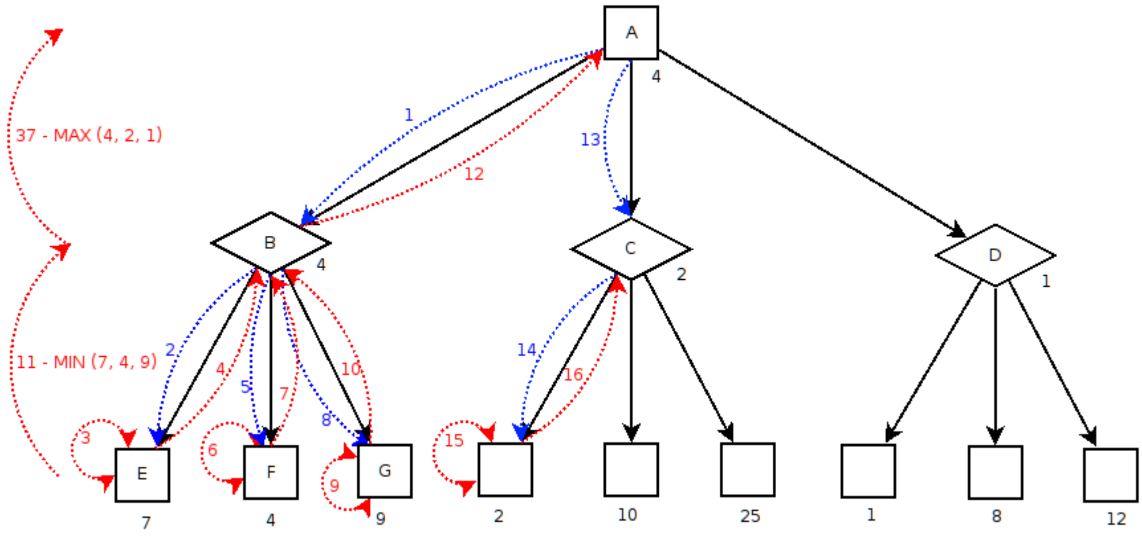
A minimax elv egy kevert stratégia, egy olyan döntési szabály, amely azt célozza meg, hogy minimalizálja a maximális veszteséget, illetve ez meg is fordítható, maximalizálja a minimális nyereséget.

Algorithm 1 Minimax algoritmus pszeudo kódja

function MINIMAX(csomopont, melyseg)
    if a *csomopont* levél, vagy $melyseg = 0$ **then**
        return csomopont heurisztikus értéke
    else
        $\alpha \leftarrow -\infty$
        for all gyerekére a csomopontnak **do**
            $\alpha \leftarrow max(\alpha, -\text{MINIMAX}(gyerek, melyseg - 1))$
            return α
        end for
    end if
end function

Célszerűségből nevezzük el a kezdőjátékost MAX-nak, a második játékost MIN-nek. A minimax elvet a játékfa felépítése során alkalmazzuk. Építsük tehát fel a játékfát az előbb tárgyaltaknak megfelelően, és a végállapotokhoz rendeljünk hozzá egy számértéket. Az olyan játékoknál, ahol csupán azt számít, hogy egy játékos nyert-e elegendő az 1 (MAX nyert), 0 (döntetlen), -1 (MIN nyert) számokkal operálni. Amennyiben a vizsgált játék kifinomultabb pontrendszer rendelkezik, akkor azt az értéket rendeljük hozzá, amennyit az adott játék végállása ér.

A megértést segítendő tekintsük a 2.1. játékfát. A MAX játékos lehetséges lépéseit a négyzetek, míg MIN játékosét a rombuszok jelölik. Hogy szemléletesebb legyen a minimax bemutatása jelen játéknál nem csupán az számít, hogy a játékos nyer-e, hanem az is, hogy mennyivel, azaz a MAX a legnagyobb értékre törekszik, míg MIN arra, hogy a MAX-nak legyen a legkevesebb pontja. Az ezt reprezentáló számérték a négyszögek alatt találhatóak, melyek minden lehetséges lépéshez megmutatják a minimax értéket. A szaggatott nyilak a mellettük található számmal minimax lépéseinek a sorrendjét jelölik. A kék a rekurzióba való belépést, míg a pirosak az abból való visszatérést szimbolizálják. Az átláthatóság érdekében a nyilakat csupán az első baloldali részfára helyeztem el, de a számértékeket kiszámítottam minden lépéshez.



2.1. ábra. Példa a minimax működésére

A minimaxot általában rekurzív algoritmussal szokták implementálni, ami ugyan a gyökérelemtől indul ki, de - a rekurziónak köszönhetően - egyből az első levélobjektumot veszi célba. Mivel mindegyik lépés minimax értékét a rákövetkező lépések minimax értékeiből számoljuk ki, ezért a fa legmélyebb pontján kell kezdenünk. A levélobjektum minimax értékének a kiszámítása triviális, ugyanis megegyezik azzal a számértékkel amit a játék végállapotához rendeltünk hozzá. Ezt az ábrán az önmagába visszaforduló nyíllal jelöltem. A nem levélobjektum minimax érték úgy tudjuk kiszámolni, hogy az összes gyerekobjektum minimax értékének vesszük vagy a minimumát, vagy pedig a maximumát attól függően, hogy az adott lépés a MIN, vagy a MAX játékoshoz tartozik-e. Nem meglepő módon a MAX játékos a legnagyobb minimax értékeket keresi, míg a MIN a lehető legkisebbet. Miután egy adott ág egy szintjén kiszámoltuk az összes minimax értéket feljebb léphetünk egy szinttel, ahol ugyan ezeket a lépéseket kell megtennünk, egészen addig, míg vissza nem jutunk a rekurzió legfelsőbb szintjére, a gyökérelemre.

Az így felépített játékfa segítségével pedig egyszerűen leolvashatjuk, hogy a MAX-nak milyen lépést kell megtennie: mivel a legnagyobb érték a 4, így a B irányába kell elindulnia, tehát meglépi a B lépést. Ekkor a MIN - ha optimálisan játszik - a legkisebb (jelen esetben szintén a 4) értékhez tartozó lépést, az F-et lépi meg.

Látszik, hogy amennyiben a MIN nem játszik optimálisan még rosszabbul jár, ezáltal a minimax algoritmus mindig optimális megoldást ad. A dolog szépséghibája, hogy még az egyszerűbb játékok esetében sem biztos, hogy a MAX-nak lesz elegendő ideje kiszámolni ezt az optimális stratégiát. Ezen a problémán tudnak segíteni a különböző vágási technikák, melyek közül egyet a következő fejezetben be is mutatok.

2.4. A játékfa levágásának módszerei

Az előző részben tárgyaltakat tekintve az olvasóban - jogosan - felmerülhet, hogy a játékfának a használata - még rendkívül egyszerű játékok esetében - is óriási költségekkel (tárhely, számítási igény) jár. A játékfa felépítése minimax algoritmussal $O(b^b)$ idő, és $O(b * m)$ tárigényű, ahol b a csomópontokban létező érvényes lépések m pedig a fa maximális mélysége. Léteznek azonban módszerek, amelyekkel a fa bizonyos részeit le lehet vágni még az előtt, hogy fel kéne építeni, így akár drasztikusan csökkentve ezen költségeket.

2.4.1. Alfa-Béta vágás

Az Alfa-Béta vágás segítségével a gyakorlatban a minimax algoritmust tudjuk felgyorsítani oly módon, hogy a döntésben részt nem vevő ágakat lenyessük, azaz igazából ki sem értékeljük. Ennek megfelelően az alfa-béta vágást a játékfa építése közben kell alkalmazni. A megértés megkönnyítése céljából ezúttal is tekintsünk a már jól ismert 2.1. ábrára. A 2. algoritmus bemutatja az alfa-béta vágás pszeudokódját.

Algorithm 2 Alfa-Béta vágás algoritmusának pszeudo kódja

```
function KIERTEKEL(csomopont, alfa, beta)
    if a csomopont levél then
        return csomópont heurisztikus értéke
    end if
    if a csomopont maximalizálandó then
        for all gyerekére a csomopontnak do
            β ← MIN(β, KIERTEKEL(gyerek, alfa, beta))
            if β <= α then
                return α
            end if
        end for
        return β
    end if
    if a csomopont minimalizálandó then
        for all gyerekére a csomopontnak do
            α ← MAX(α, KIERTEKEL(gyerek, alfa, beta))
            if β <= α then
                return β
            end if
        end for
        return α
    end if
end function
```

Értékeljük ki a minimax értékeket a baloldali részfán az előbb tárgyalt módon, és kezdjük el kiépíteni a második részfát. Láthatjuk, hogy rögtön az első levélobjektumra kiszámított minimax érték kettő, ami már most kisebb, mint az előző részfa utolsó előtti szintjére számított minimális minimax érték. Ez azt jelenti, hogy teljesen felesleges kiszámolnunk jelen részfa további értékeit, hiszen még ha találnánk is nagyobb értéket az utolsó előtti szinten úgyis a minimumot kell vennünk, ami legrosszabb esetben is maga a kettő, ugyanakkor az eggyel feljebb lévő szinten a maximumot keressük, így - mivel a négy nagyobb a kettőnél - a kettő, s ezáltal a teljes C részfa eldobható. Ugyan ez a helyzet a D részfával is. Az alfa-béta vágás használatával jelen esetben csupán az első részfát kellett teljesen kiépíteni, a többi részfa az első levélelem minimax számítása után eldobhatóvá vált, s ezzel jelentősen csökkent mind az időigény, mind pedig a tárigény.

Természetesen nem minden esetben vagyunk ilyen szerencsések. Látszik, hogy nagyon nem lényegtelen, hogy az ágakat milyen sorrendben értékeljük ki. Sokszor alkalmaznak kiegészítő heurisztikus függvényeket, amelyek megpróbálják megbecsülni a kiértékelés ideális sorrendjét, sőt egyes esetekben - a játék sajátosságaiból adódóan - ez a sorrend előre ismert, vagy kiszámítható. Ez az algoritmus kifejezetten hatékony olyan esetekben, amikor a csomópontoknak sok gyermeke van, és jól meg tudjuk becsülni, hogy melyik az az elem, amelyik elbuktathatja az adott ágat. Ha ezt megtehetjük, akkor az algoritmus időigénye $O(b^{m/2})$-re redukálódik, ami szignifikánsan jobb a minimax $O(b^m)$ időigényétől, ami nagyjából megduplázza a minimax sebességét.

2.5. A Nim játék leírása

A Nim játék egy kétszemélyes teljes információjú körökre osztott stratégiai játék. Egyszerűségéből fakadóan számos változata, illetve továbbgondolása is létezik. Néhányat a későbbiekben röviden ismertetni is fogok.

A játék körökre bontott, azaz a játékosok felváltva teszik meg lépéseiket. A játék másik lényeges tulajdonsága, hogy teljes információjú játék, azaz a játék kezdetétől fogva mindkét játékos rendelkezésére áll az összes a játékra vonatkozó ismeret, beleértve a szabályokat, és a teljes játékteret.

Nim játék esetében minden kör egy, és csakis egy lépésből áll, amit az éppen soron következő játékosnak kötelezően meg kell tennie. A játéktér tetszőleges számú halomból állhat, melynek elemeinek darabszáma csakugyan kötetlen (lehet egyforma, és akár mindegyik halom eltérő elemszámú). Hagyományosan ezek az elemek kavicsok, de igazából matematikai szempontból ezen entitások manifesztációja lényegtelen. Mindegyik lépés abból áll, hogy az éppen soron következő játékos az egyik nem üres elemszámú halomból elvesz legalább egy, legfeljebb az adott halom elemszámával megegyező darab (tehát akár az egész halmot) entitást a halomból.

A játék célja az, hogy amikor sorra kerülünk, akkor ne legyen már több halom, azaz az ellenfelet olyan helyzetbe hozzuk, hogy az végső lépést õ teszi meg, az utolsó entitás(okat) ő veszi el. Ez egyébként a leggyakrabban játszott Nim változat, Misère néven is ismert. Mint már említettem a Nim játéknak számos változata létezik, így előfordul, hogy fordítva játsszák, azaz nem az a soron következő játékos nyer, aki nem tud lépni, hanem az, aki a végső elem(eket) elveszi az utolsó halomból.

2.6. A Nim játék története

A Nim játék különböző variációit nagyon régóta játsszák. Pontos információink nincsenek, de egyes források arra engednek következtetni, hogy már az ókori Kínában is játszották ezt a játékot. Ugyancsak erre enged következtetni a 捡石子 (jiǎn-shízi) kínai eredetű játék, amely kísértetiesen hasonlít a Nim játékra, azzal a kivétellel, hogy ott egy halommal játsszák, igaz ennek is sok variánsa létezik, és az érvényes lépéseknek a szabályai bonyolultabbak. Európában először a 16. század kezdetén tesznek róla említést, de igazán a figyelem középpontjába csak a 19. század végén került, amikor Charles L. Bouton tanulmányozta, majd 1901-ben a játék teljes elméletét kidolgozta. Úgy tudni a játékot is ő keresztelte el Nimnek, a német "Nimm" (elvenni) szó alapján. Más források arra hívják fel a figyelmet, hogy a "NIM" szót 180 fokkal elfordítva az angol "WIN" (nyerni) szót kaphatjuk meg. A játék további ismertségre tett szert az 1939-es New York-i világkiállításon, ahol az 1886-ban alapított amerikai Westinghouse Electric Corporation cég bemutatta a Nimatront, egy olyan gépet, amely Nim játékot játszott. A dolog külön érdekessége, hogy ez volt világon az első elektronikus számítógépes játék.

2.7. Ismertebb Nim variációk

2.7.1. Moore-Nim

A Moore-Nim játék nevét kitalálójáról Eliakim Hastings Moore amerikai matematikusról kapta. Ez egyfajta általánosítása a több-halmos Nim játékoknak, ahol a játékosok egyszerre nem csak egy, hanem legalább egy, maximum k halomból vehetnek el elemeket. Az elvehető elemek számát lehet korlátozni is.

2.7.2. Póker-Nim

A póker-Nimet egy előre rögzített számú entitással játsszák. Kezdetben az összes elemet felhasználva létrehoznak egy normál Nim játékot, majd a játékosok hagyományos Nim játékot játszanak azzal a különbséggel, hogy sorra kerülésükkor nem csupán elvehetnek a halomból, hanem már az elvett elemeket újra felhasználva azokat a halomhoz hozzáadhatják. Könnyen belátható, hogy új elemek hozzáadása a kupachoz nem befolyásolja lényegesen a játékmenetet, hiszen a következő játékos tetszőleges számú elemet elvehet egy kupacból beleértve az előző személy által hozzáadott extra elemeket is.

2.7.3. Lasker-Nim

Nevét az német-amerikai sakk- és go-mesterről, Edward Laskerről kapta. Ő javasolta, hogy a hagyományos Nim játékot egészítsék ki egy új érvényes művelettel, a halmok kettébontásával. Ez a kettébontás nem feltétlenül két egyenlő félre való bontást jelent, a két új halom elemszámainak arányára vonatkozóan nincsen megkötés.

2.7.4. End-Nim

End-Nim esetében a halmok sorba vannak rendezve, és bár a játékosok a hagyományos Nim szabályai szerint játszanak, azonban csupán a sor két végén levő halmokból vehetnek el elemeket.

2.7.5. Fibonacci-Nim

Ezt a Nim variánst egyetlen halommal játsszák, de a benne lévő elemek számára nincsen megkötés. A hagyományos Nim játékhoz képest az a különbség, hogy a játékosok legfeljebb mindig az előző játékos által elvett elemek kétszeresét veheti el. A kezdő lépést megtevő játékos tetszőleges (de nem az összes) elemet elvehet a halomból. A kupacból az utolsó entitást elvevő játékos nyer.

2.7.6. Wythoff-Nim

Willem Abraham Wythoff Holland matematikusról kapta a nevét, aki 1907-ben publikálta a játék matematikai analízisét. A játékot két érmehalommal játsszák, minden körben a soron lévő játékosnak el kell vennie valamennyi érmét az egyik kupacból, vagy mindkét halomból egyenlő számú érmét. Az nyer, aki az utolsó érmét, vagy érméket elveszi.

A játék megegyezik a királynőt a sarokba játékkal, ahol egy vezért kell eljuttatni valamelyik (általában bal alsó) sarokba, de egy lépés csak akkor érvényes ha azzal közelebb kerül a célhoz.

2.7.7. End-Wythoff

Az egyik legbonyolultabb Nim játék. Ez ötvözi az End-Nim, és a Wythoff-Nim szabályait. Tetszőleges számú halommal játsszák, de a halmok sorba vannak rendezve, és a soron következő játékos csak a szélén lévő halomból vehet el elemeket, vagy a szélén lévő két kupacból megegyező számú elemet.

2.7.8. Az osztó játék

Egy tetszőleges természetes számról indulva a játékosok minden körben elosztják ezt a számot egy olyan prím szám valamely hatványával, amely osztója az éppen aktuális számnak (kivéve természetesen az 1). Az a játékos, amelyik a végén eljut az 1-re, az nyer, vagy veszít attól függően melyik változatát játsszák.

2.7.9. A kivonó játék

Sokan tévesen ezt a variánst ismerik Nim játékként. Többnyire egy halommal játsszák, azonban az elvehető elemek maximális számára van valamilyen $S(1, 2, ..., k)$ korlát.

2.7.10. A 21 játék

Ezt a játékot Misère játékként játsszák. A kezdő mond egy 20-nál kisebb pozitív egész számot, majd a soron következő játékosok az aktuális számot 1, 2, 3-mal növelhetik, de nem léphetik át a 21-et. Az a játékos, amelyik a 21 kimondására kényszerül elveszíti a játszmát.

2.7.11. A 100 játék

A 21 játékhoz nagyon hasonló. Itt a célszám 100, és az azt elérő játékos nyer. A kezdőszám a 0, és a játékosok körönként 1 és 10 közötti egész számot adhatnak hozzá az aktuális értékhez.

2.7.12. Körkörös Nim (Kayles)

Az elemek ebben a variánsban körben helyezkednek el, és a soron következő játékos legfeljebb k egymást követő elemet vehet el a körből. Játszható normál és Misère változatban is.

2.7.13. Grundy játéka

A Grundy játéka egy halommal indul benne tetszőleges számú elemmel. A hagyományos Nim játéktól eltérően itt nem elemeket vesznek el a halomból, hanem a halmokat bontják két nem egyforma méretű halomra. A játéknak akkor van vége, amikor már nincs olyan halom, amit ketté lehetne bontani két nem egyforma méretű halommá. Ezt a variánst is lehet normál, illetve Misère módon játszani.

2.7.14. Mohó Nim

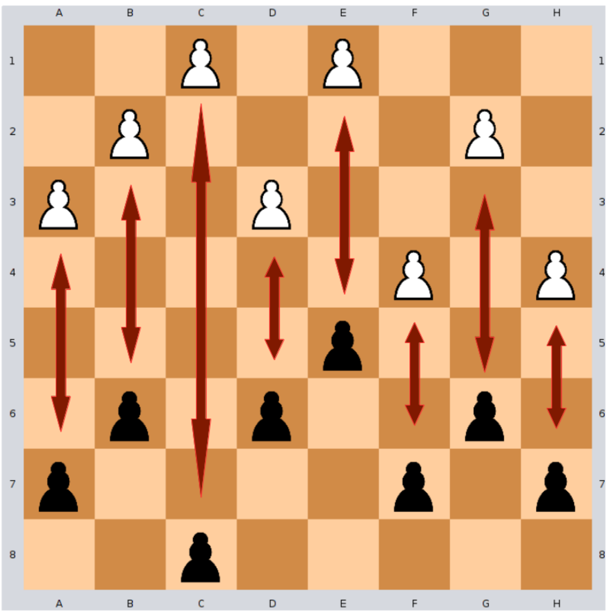
Normális, és Misère módon is játszható a Mohó-Nim, ami csupán annyiban különbözik a hagyományos Nim játéktól, hogy a játékosok csak a legnagyobb elemszámú halomból vehetnek el elemeket.

2.7.15. Építő Nim

Az építő-Nim két részből tevődik össze. Először felépítik Nim játékot az előre megadott számú elemet felhasználva, majd azt lejátsszák. Felépítéskor a játékosok körönként 1-1 elemet raknak (a kezdetben üres) előre meghatározott számú halomba.

2.7.16. Northcott-sakk

A Northcott-sakk egy 8x8-as sakktáblából áll. Kezdetben a játékosok (egymás elől elrejtve) elhelyezik a bábuikat (mindegyik oszlopba csak egyet téve) a saját térfelükön. Miután ezzel végeztek elkezdődik a játék. A játékosok felváltva lépnek, minden körben csak az egyik saját bábujukkal, csak előre, és legfeljebb annyi mezőt, amennyi üres hely van az ő bábuja, és az ellenfél bábuja között, azaz az ellenfél bábuját nem ütheti le, és nem ugorhatja át.



2.2. ábra. Nortcott-sakk táblája. A nyilak a bábuk közti távolságot jelöli, ami a halom méretének feleltethető meg.

Ha jobban belegondolunk ez a játék egy az egyben megfeleltethető a hagyományos Nim játéknak, ebből következően játszható sima, és Misère módon is.Szempontunkból ez a variáns azért is különösen érdekes, mert a szakdolgozatom tartalmazza ennek a játéknak a példaimplementációját, ami ráadásul ténylegesen Nim játékot játszik a háttérben ugyanazt a játékosztályt használva ezzel szemléltetve, hogy mennyire is visszavezethető a Northcott-sakk a standard Nim játékra.

2.8. A Nim játék matematikai háttere

A matematikai háttér elemzését kezdjük néhány definícióval:

2.1. definíció. Egy nem negatív elemekből álló halmaz legkisebb kizártjának (lkkz) a legkisebb olyan egész számot nevezzük, ami nem szerepel a halmazban.

2.2. példa. Például az $A := \{0, 1, 2, 4, 5, 7, 9\}$ halmaz legkisebb kizártja a 3, mert ez az első olyan nem negatív egész szám, ami nem szerepel az A halmazban.

2.3. definíció. Halmazelméletben rendszámnak nevezzük a természetes számok koncepciójának egy olyan általánosítását, ami leírja, hogy objektumok egy gyűjteményét hogyan lehet sorrendbe helyezni, egyiket a másik után.

2.4. definíció. Nimbereknek (vagy más néven Grundy-számoknak) nevezzük azokat a rendszámokat, amelyeket két további művelettel ruházunk fel. A nimber-összeadással, és a nimber-szorzással.Nimber-összeadás művelete: $\alpha \oplus \beta = lkkz(\{\alpha' \oplus \beta : \alpha' < \alpha\} \cup \{\alpha \oplus \beta' : \beta' < \beta\})$Nimber-szorzás művelete: $\alpha\beta = lkkz(\{\alpha'\beta + \alpha\beta' + \alpha'\beta' : \alpha' < \alpha, \beta' < \beta\})$

2.5. megjegyzés. Számunkra lényeges jelentősége a nimber-összeadásnak van, amelyet számítógépen triviálisan egyszerű implementálni, ugyanis véges rendszámok esetében ez nem más, mint a bitenkénti kizáró vagy, azaz a XOR művelet, melyre a legtöbb processzorban létezik utasítás.

Egyszerűség kedvéért vegyünk egy egy halomból álló Nim játékot. A játék minden lépéséhez rendeljünk hozzá egy nimbert. Ez a szám legyen 0, ha vesztő pozícióban vagyunk, azaz a halomban nincsen elem. Az egy elemű halmaz állapotát az előbbihez képest könnyen meg tudjuk határozni, hiszen egyből csak a nulla pozícióba léphetünk. A legkisebb kizártja ennek a halmaznak az 1, ennek megfelelően rendeljük az egyet a halomhoz. Látható, hogy a kupachoz rendelt nimber megegyezik a halom elemszámosságával.

Ezt továbbgondolva kijelenthető, hogy a nimber definíciójából adódóan egy Nim játéknak csak akkor van nyerő stratégiája, ha a játék nimbere nem nulla.

2.9. Nyerő stratégia

A [5] internetes oldalon található egy leírás a nyerő stratégiához egy remek bizonyítással egybekötve, amihez úgy érzem nem tudnék érdemben hozzá tenni, ugyanakkor ez a dolgozat bizonyítás nélkül nem lehet teljes, így az alábbiakban közlöm az ezen az oldalon található angol nyelvű nyerő stratégiájának mind a leírását, mind pedig annak bizonyítását magyar nyelvre fordítva. Az eredeti bizonyítás Charles L. Boutontól származik, aki elsőként dolgozta ki a Nim játék teljes matematikai hátterét. [3]

A nyerő stratégia bemutatását kezdjük egy nagyon fontos definícióval:

2.6. definíció. Az a és b nemnegatív egész számokon elvégzett $a \oplus b$ műveletet nim-összegnek nevezzük, amennyiben a következő módon kerül kiszámításra. Jelentse a és b kettő különálló hatványainak összegét. Vessük el kettő olyan hatványait, amelyek többször is szerepelnek, majd a fennmaradó hatványokat adjuk össze.

2.7. megjegyzés. Ez a definíció gyakorlatilag a nimbereken elvégzett nimber-összeadás műveletét írja le kicsit másképpen, de a gyakorlatban ugyan arról van szó. Mint már említettem ez a XOR logikai műveletének felel meg, implementálás során én is ezt használtam ki.

Például $3 \oplus 5$ a következőképpen számítható ki. A $3 = 2^1 + 2^0$, és az $5 = 2^2 + 2^0$. Mivel a 2^0 kétszer szerepel, ezért eldobjuk, a maradékot pedig összeadjuk 2^1+2^2 kiadva az $3 \oplus 5 = 6$ eredményt.

Bizonyítható, hogy $\oplus$ asszociatív, ezáltal több szám nim-összegét $a_1 \oplus a_2 \oplus ... \oplus a_n$ definiálja.

Vegyünk egy normál Nim állást, ahol $a_1, a_2, ...a_n$ jelentik a halmok méreteit. Az éppen soron következő játékos akkor nyer, ha $a_1 \oplus a_2 \oplus ... \oplus a_n \neq 0$; továbbá a nyerő lépést megtalálni az $i \in \{1, 2, .., n\}$ halom és a $b_i \in \{0, 1, ..., a_i - 1\}$ számérték meghatározásával lehet úgy, hogy $a_1 \oplus a_2 \oplus ... \oplus a_{i-1} \oplus b_i \oplus a_{i+1} + a_{i+2} \oplus ... \oplus a_n = 0$, és az i halmazból elvéve valamennyi elemet b_i-t hátrahagyva. Ha $a_1 \oplus a_2 \oplus ... \oplus a_n = 0$, akkor az éppen lépő játékos veszít.

Misère Nim esetében a stratégia majdnem teljesen azonos. Egészen addig, amíg a javasolt lépés elvégzése után marad legalább egy halom 2, vagy több elemmel használjuk, a normál Nim stratégiáját. Amennyiben a javasolt lépés után nem marad legalább egy halom kettő, vagy több elemmel, más lépést kell tennünk:

- Ha a javasolt lépés után 1 elem maradna hátra, akkor vegyük el az egész halmot, vagy
- Ha a javasolt lépés után nem maradna elem a halomban, úgy hagyjunk benne egy elemet.

Más szóval a helyes lépés az, hogy páratlan számú halmokat hagyjunk meg 1 elemmérettel (Normál esetben páros számú 1 méretű halmokra törekszünk, ezzel a nim-összeget zérussá téve).

2.9.1. Nyerő stratégia bizonyítása

2.8. tétel. *A soron következő játékos akkor, és csak akkor nyeri meg a normál Nim játékot, ha a halmok nim-összege nem zérus.*

Bizonyítás. Kezdésnek vegyük az egyszerű alapesetet: ha mindegyik halom elemszáma zérus, akkor a soron következő játékos veszít, és a nim-összeg is zérus. Ettől fogva tegyük fel, hogy nem mindegyik halom üres.

Először is vegyük észre, hogy a nim-összeg számos fontos tulajdonsággal rendelkezik. Minden nem negatív a, b, c egész számra igaz, hogy:

- Asszociatív: $(a \oplus b) \oplus c = a \oplus (b \oplus c)$
- Kommutatív: $a \oplus b = b \oplus a$
- Létezik semleges eleme: $0 \oplus a = a$
- Öninverz: $a \oplus a = 0$
- Lehetséges egyszerre több számnak a Nim-összegét meghatározni oly módon, hogy felírjuk az összes számot 2 különálló hatványaira, majd megkeressük 2 összes olyan hatványát, mely páratlanszor szerepel, végül összeadjuk ezeket 2 hatványokat úgy, hogy mindegyiket csak egyszer vesszük. Például: $1 \oplus 3 \oplus 7 = (2^0) \oplus (2^0 + 2^1) \oplus (2^0 + 2^1 + 2^2) = 2^0 + 2^2 = 5$

Tegyük fel, hogy a halmok elemszáma minden lépés előtt $a_1, a_2, ..., a_n$, illetve $b_1, b_2, ..., b_n$ minden lépés végén. Feltételezzük továbbá, hogy amennyiben k halmon végzünk el egy lépést, akkor minden $i \neq k$ -ra $a_i = b_i$. Legyen $s = a_1 \oplus a_2 \oplus ... \oplus a_n$ és $t_n = b_1 \oplus b_2 \oplus ... \oplus b_n$. Ekkor a következőket kapjuk:

$$
\begin{aligned}
t &= 0 \oplus t \\
&= (s \oplus s) \oplus t \\
&= s \oplus (s \oplus t) \\
&= s \oplus ((a_1 \oplus a_2 \oplus ... \oplus a_n) \oplus (b_1 \oplus b_2 \oplus ... \oplus b_n)) \\
&= s \oplus ((a_1 \oplus b_1) \oplus (a2_{\oplus} b_2) \oplus ... \oplus (a_n \oplus b_n)) \\
&= s \oplus (0 \oplus 0 \oplus ... \oplus 0 \oplus (a_k \oplus b_k) \oplus 0 \oplus ... \oplus 0) \\
&= s \oplus (a_k \oplus b_k).
\end{aligned}
$$

Most két eset bizonyítása következik.

2.9. lemma. *Első eset: ha $s = 0$, akkor $t \neq 0$. Amennyiben az eredeti méretek nim-összege zérus, úgy a lépést végző játékos vesztésre áll (ebből kifolyólag a nim-összeget nem-zérussá kell alakítania)*

Azt állítjuk, hogy $a_k \oplus b_k \neq 0$. Tegyük fel, hogy így is van, ekkor:

$$
\begin{aligned}
a_k &= a_k \oplus 0 \\
&= a_k \oplus (a_k \oplus b_k) \\
&= (a_k \oplus a_k) \oplus b_k \\
&= b_k.
\end{aligned}
$$

Tehát $a_k = b_k$. De ez ellent mond annak a ténynek, hogy a lépést végrehajtó játékos a b_k halmon végezte el a lépést, s így nem is csökkentette a halom méretét. Tehát mivel $a_k \oplus b_k \neq 0$, így:

$$
\begin{aligned}
t &= a \oplus (a_k \oplus b_k) \\
&= 0 \oplus (a_k \oplus b_k) \\
&= a_k \oplus b_k \\
&\neq 0.
\end{aligned}
$$

2.10. lemma. *Második eset: Ha $s \neq 0$, akkor lehetséges, hogy $t = 0$. Amennyiben az eredeti halmok nim-összege nem zérus, abban az esetben az éppen lépő játékos nyertes helyzetben van. (hiszen a nim-összeget zérussá tudja alakítani)*

Vegyük számításba, hogy 2 legmagasabb hatványa 2^k nem nagyobb, mint s. Léteznie kell legalább egy olyan a_i-nak, ami szintén tartalmazza 2^k-t, különben 2^k nem szerepelhetne s-ben. Most vegyük $b_i = s \oplus a_i$-t. A b_i értéke 2^k-nal csökken, és legfeljebb $2^{k-1} + 2^{k-2} + ... + 2^0 = 2^k - 1$-gyel nő. (2 minden visszamaradt hatványa kiadja s-t, hozzáadódva az értékhez; például $s = s^2 + 2^1 + 2^0$ és $a_i = 2^3 + 2^2$ kiadja $b_i = 2^3 + 2^1 + 2^0$-t), tehát $b_i < a_i$. Továbbá:

$$
\begin{aligned}
t &= s \oplus (a_i \oplus b_i) \\
&= s \oplus (a_i \oplus (s \oplus a_i)) \\
&= (s \oplus s) \oplus (a_i \oplus a_i) \\
&= 0.
\end{aligned}
$$

Ezzel a tétel bizonyítva van. □

3. fejezet

Nim játék és Northcott-sakk példaprogram

3.1. Felhasználói dokumentáció

3.1.1. Rendszerkövetelmények

A program Java-ban íródott, éppen ezért futtatásához Java SE 1.8 futtatókörnyezet szükséges. A program nem támaszt különösebb rendszerkövetelményt a futtató géppel szemben, egy mai asztali számítógépen probléma nélkül el kell, hogy fusson. Legalább egy 1024x768-as felbontású monitor szükséges az ablak megjelenítéséhez, továbbá a programot egérrel, és billentyűzettel (vagy ezt kiváltó perifériákkal) lehet vezérelni.

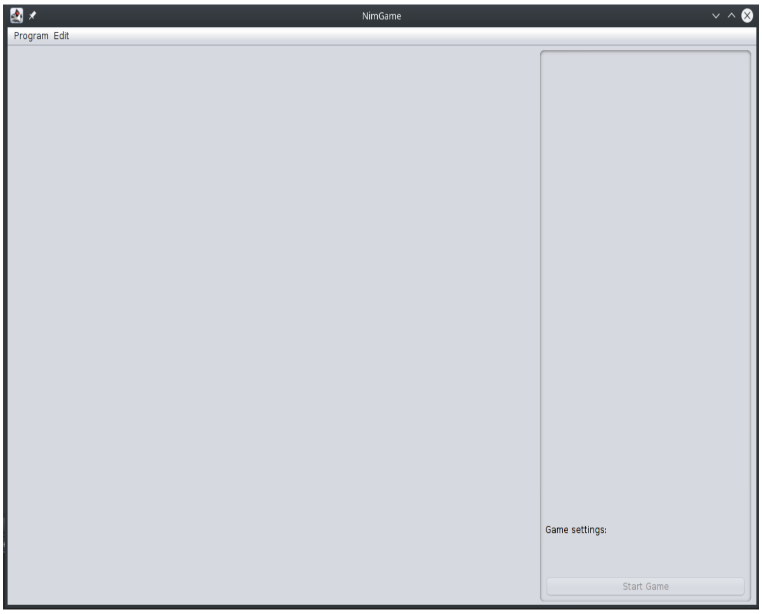
3.1.2. Telepítési útmutató

A programot nem szükséges telepíteni, a Java archívumot (.jar) egyszerűen a Java futtatókörnyezetével elindítva a program futtatható.

3.1.3. A program használata

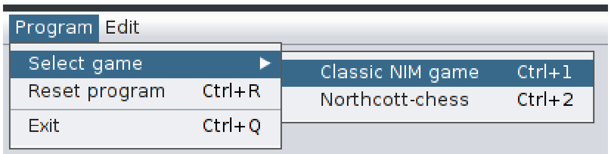
A programot elindítva az 3.1. főképernyő fogad minket:

A szoftver úgy lett megtervezve, hogy több különböző (nem feltétlenül csak Nim) játék futtatására legyen alkalmas, de alapvetően mindegyik játék három főbb elemmel rendelkezik. A beállításpanellel, az állapotpanellel, és a főpanellel. Az első kettő a jobb oldalon található oldalsó panelen jelenik meg, míg a főpanel az ablak nagyobb részét kitöltő üres helyen helyezkedik el, amint a megfelelő játékot betöltjük.



3.1. ábra. A példaprogram főképernyője

Játékot betölteni a "Program" menü (3.2.) "Select game" almenü segítségével lehet megtenni. Itt ki kell választani a kívánt játékot.



3.2. ábra. A főképernyő program menüje

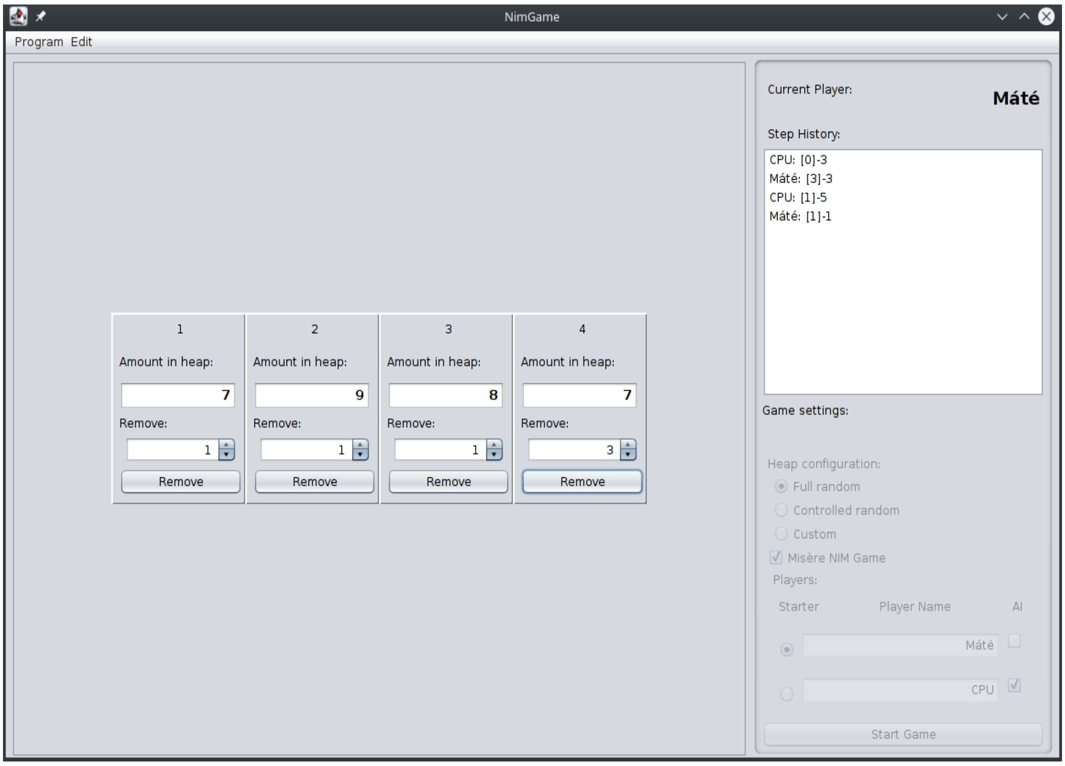
Jelenleg két játék közül lehet választani, ezeknek a leírására külön fejezetet szentelek. Az aktuális játékot bezárni a "Reset program" menüelemmel lehetséges, továbbá a programból kilépni az "Exit" menüelem használatával tud a felhasználó.

Ezek a menüelemek a program futása során bármikor elérhetőek, azonban fontos megjegyezni, hogy amennyiben futó játék alatt egy másik játéktípus kerül kiválasztásra, abban az esetben az éppen játszott játék bezárásra kerül csak úgy, mintha a "Reset program" menüelemet választottuk volna.

A legtöbb menüelemet gyorsbillentyű segítségével is aktiválni lehet. Hogy melyik elemhez milyen gyorsbillentyű tartozik arról a menüfelirat mellett elhelyezkedő billentyűparancs ad tájékoztatást.

3.1.4. Klasszikus Nim Játék

A klasszikus Nim - mint arra a neve is utal - a hagyományos Nim játék megvalósítása. Az 3.3. ábra éppen azt mutatja, amint a gép ellen játszok. Jól látszik, hogy a játékban (éppen) 4 halom van, amelyekben a kavicsok darabszáma rendre 7, 9, 8, 7. Az állapotablakban jól látszik, hogy négy körön már túl is vagyunk.



3.3. ábra. Klasszikus Nim játék a gép ellen

A játék kezdete előtt lehetőségünk van részletekbe menően beállítani, hogy pontosan hogyan szeretnénk játszani a játékot. Ehhez be kell állítanunk a halom konfigurációt és a játékosokat. Miután a beállításokkal végeztünk az oldalsó panel alján található "Start Game" gombbal kezdhetjük a játékot. Ezután a beállítások módosítása már nem lehetséges, de bármikor kezdhető új játék a már említett menüelemek használatával.

Játékos beállítások

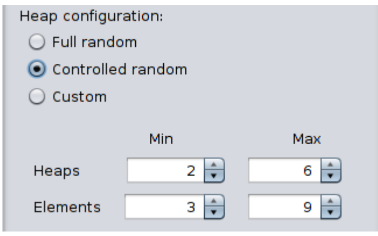
A Nim játékot természetéből fakadóan két "személy" játssza. A szövegbeviteli mezőkbe a játékosok neveit kell megadni (eltérőeknek kell lenniük). A bal oldalon található rádiógomb segítségével azt választhatjuk ki, hogy melyik játékos kezdjen, még a jobb oldalon elhelyezkedő jelölőnégyzet arra szolgál, hogy tudassuk a programmal, hogy azt a játékost ő vezérli.A beállítások adta szabadságból látszik, hogy a programmal lehet ember-ember, gép-ember, és gép-gép játékot is játszani.

Halom beállítások

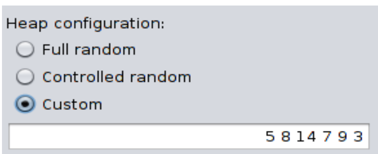
A halombeállítások kezdetén azt kell eldönteni, hogy milyen mélységben szeretnénk beleszólni a kezdeti játéktér felépítésébe. Ezt vezérelni a "Heap Configuration" szöveg alatti rádiógomb-csoporttal lehetséges. Ezekre kattintva a felület dinamikusan átalakul további funkciókat felfedve.

- Full random: Teljes egészében a programra bízzuk, hogy hány halmot generál milyen elemszámmal
- Controlled random: Továbbra is a gépre bízzuk, hogy összeállítsa a játékteret, azonban a generálás szabályokat ezzel vezérelni tudjuk. Erre kattintva a 3.4. panel megjelenik, ahol az első sorban a halom számát, a második sorban a halmok elemszámára tudunk egyéni korlátot megadni. Az első oszlop az alsó korlátot, a második pedig a felső korlátot adja meg.
- Custom: Itt teljesen megszabjuk, hogy hány halmot szeretnénk milyen elemszámmal. Erre a gombra kattintva egy új panel (3.5.) jelenik meg, ami egy szövegbeviteli mezőből áll. A mezőbe szóközzel elválasztva kell megadni, hogy a halmokban hány elem legyen. Mindegyik szám egy halmot reprezentál, és a halmok az itt megadott sorrendben fognak létrejönni.

Amennyiben Misère Nim játék helyett normál módon akarunk játszani, akkor a "Misère NIM Game" jelölőnégyzetetek szüntessük meg a bejelöltségét.



3.4. ábra. Irányított generálás



3.5. ábra. Egyedi halomkonfiguráció

A játék menete

Miután végeztünk a beállításokkal kattintsunk a "Start Game" gombra, és a játék elindul. Ekkor a jobb felső sarokban található állapotablak tájékoztat minket arról, hogy ki az éppen soron lévő játékos.

A főpanelon látszódnak a halmok 1-1 panel formájában. Mindegyik panel tetején megtalálható a panel sorszáma, alatta egy szövegbeviteli mezőben a halomban található elemek száma, az alatt egy pörgettyűben beállítható az elvenni kívánt elemek darabszáma, majd legalul az elvesz gomb.

Az éppen soron következő játékosnak meg kell hoznia megfelelő döntést, és lépnie kell oly módon, hogy a kiválasztott halomhoz tartozó pörgettyűbe beállítja mennyivel szeretné csökkenteni a halom elemszámát, és megnyomja a halomhoz tartozó "Remove" gombot. Ekkor a halom elemszáma csökken, a játékos köre véget ér, és a következő játékos kerül sorra. Amennyiben egy halom elfogy abban az esetben az azt reprezentáló panel eltűnik.

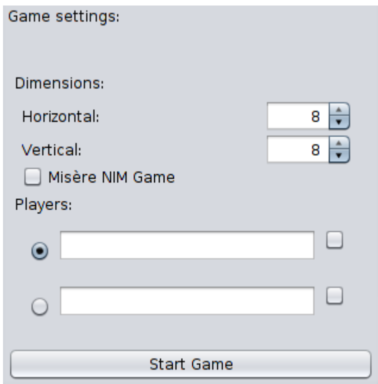
A játék véget ér, amennyiben mindegyik halom elfogy, és a program egy előugró üzenetben ad tájékoztatást arról, hogy melyik játékos nyerte meg a játékot.

3.1.5. Northcott-sakk

A játék beállítása

A Northcott-sakk sok tekintetben hasonlít a hagyományos Nim játékra, éppen ezért beállítása is hasonlóan történik. A szövegbeviteli mezőkbe a játékosnevet kell írni, a bal oldali rádiógomb kiválasztja a kezdőjátékost, míg a jobb kéz felől található jelölőnégyzet a játékost gépinek jelöli.

Ami viszont lényeges eltérés, hogy itt a halom beállításai mások, mint a Nim esetében. A 3.6. ábrát megfigyelve lényegesen leszűkültek a beállítási lehetőségek. Itt gyakorlatilag csak a sakktábla dimenzióját adhatjuk meg. A "Horizontal" (Vízszintes) az oszlopok számát, míg a "Vertical" (függőleges) a sorok számát adja meg. Ez nim játékra levetítve oszlop darab halom, és minden halomban maximálisan sor darab elem.



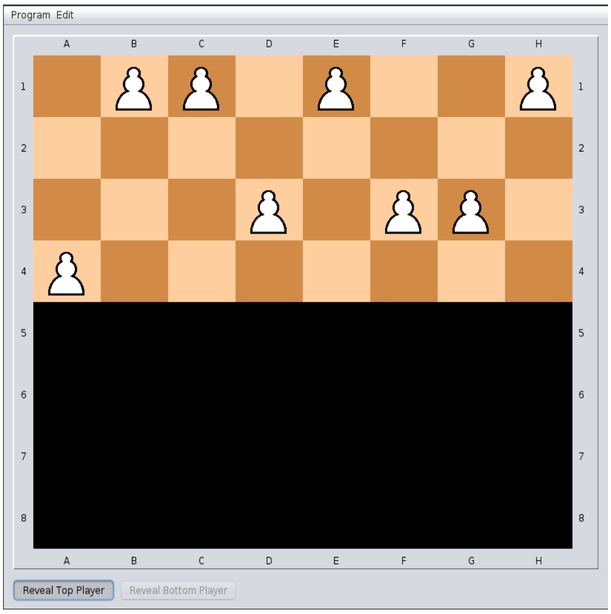
3.6. ábra. Northcott-sakk beállítópanelje

Ez a játék is játszható Misère módon, a hozzá tartozó "Misère NIM Game" jelölőnégyzettel lehet állítani, hogy Misère, vagy normál módon kívánunk játszani.

Látható, hogy a véletlen halomgenerálás teljesen eltűnt. Ennek az az oka, hogy a játék kezdése előtt van egy extra művelet, ami nem teszi lehetségessé a véletlen generálást.

A játék előkészítése

Ha végeztünk a játékbeállításokkal, akkor még ne indítsuk el a játékot, ugyanis a játékosoknak lehetőségük van beállítani a bábujuk kezdő pozícióját. A játéktér kezdetben mindkét játékos térfelét elfedi. A térfeleket az főpanel alján lévő két vezérlőgombbal lehet fel, illetve elfedni. A gép által vezérelt játékos bábuinak kezdőpozíciója nem állítható, így a hozzá tartozó gomb eltávolításra kerül, ha jelölőnégyzetet bejelöltük.



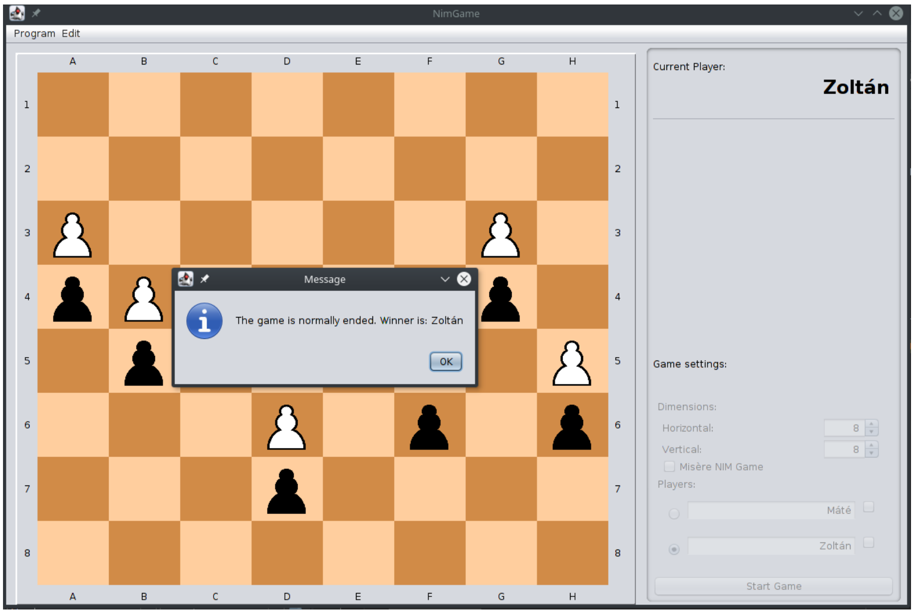
3.7. ábra. Northcott-sakk kezdő pozíciók beállítása

A felső játékos - miután meggyőződött arról, hogy ellenfele nem figyeli - a "Reveal Top Player" gombra kattintva felfedi saját játékterét. Felfedett állapotba a gomb "beragad", és a másik gomb letiltódik, hogy véletlenül ne lehessen az ellenfél térfelére átváltani. A felső játékos a kívánt mezőre kattintva áthelyezheti az adott oszlopban lévő bábuját arra a mezőre, amelyikre kattintott. Ha a felső játékos késznek érzi a kezdőállapotot, akkor a beragadt gombra újból rákattintva elfedi a felső játékteret, és átadja a helyét az alsó játékosnak, aki ugyanezt a műveletsort végigjátssza.

Miután mindketten végeztek a beállítással, a játék a "Start Game" gombra kattintva indítható. Ezután a játék beállításai már nem módosíthatóak, a beállításhoz tartozó vezérlőelemek letiltásra, a játéktér elrejtéséért/felfedéséért felelős vezérlőgombok eltávolításra kerülnek.

A játék menete

A játék indulásakor a teljes játéktér felfedésre kerül, és a program a jobb felső sarokban található állapotpanelen tájékoztat a soron következő játékos kilétéről. Lépni úgy lehet, hogy a kiválasztott oszlopban arra a mezőre kattintunk, ahova a saját bábunkkal lépni szeretnénk. A mezőre kattintva a lépés bekövetkezik, és a játékos köre véget ér, helyére a következő játékos lép.



3.8. ábra. Northcott-sakk játék vége; nyertes játékos Zoltán

A játék addig tart, amíg van olyan oszlop, ahol a bábuk között lévő távolság nagyobb, mint 0, azaz nem közvetlenül egymás mellett áll mindegyik bábu. A játék végén a program egy előugró ablakban ad tájékoztatást arról, hogy a játékot melyik játékos nyerte meg.

3.2. Fejlesztői dokumentáció

Általánosságban elmondható, hogy amikor az ember találkozik egy konkrét problémával, az gyakran egy nagyobb, összetettebb problémakör része. Éppen emiatt amikor egy megoldandó problémával kerülök szembe, első körben sosem az adott problémára koncentrálok, hanem próbálom megtalálni mi az a kiinduló pont, ahonnan a probléma valójában eredeztethető. Ennek meg van az a hátránya, hogy sokszor lényegesen több munkát kell belefektetni abba, hogy a konkrét problémát megoldjuk, ugyanakkor a megoldás sokkal több problémára nyújt megoldást, s később nagyságrendekkel könnyebben lehet adaptálódni a változó igényekhez. A saját tapasztalataim is azt mutatják, hogy egy adott problémára adandó megoldás követelményei kezdetben egészen mások, mint amire végül szükség van.

Nem volt ez most sem másképp. Eredeti megközelítésemben egy univerzális játékmotort képzeltem el, ahol a játékok maguk csupán kiegészítései a szoftvernek, amiket a felhasználók - akár futásidőben is - készíthetnek, vagy módosíthatnak. Ehhez a játékmotorhoz tartozott volna egy univerzális gépi játékos is, ami a játékleíróban található szabályok szerint játszott volna. Ez az elképzelés nagyon szép, ugyanakkor rengeteg időt vett volna igénybe, ami nem állt a rendelkezésemre, így ezt tervet végül végletekig leegyszerűsítettem. Egy dologból azonban nem engedtem. A játékok most is különálló modulként viselkednek (csak bele vannak "égetve" magába a játékmotorba), és erre nagyon jó okom volt.

Nem célom a teljes forráskód beillesztése a dolgozatba, mert egyrészt viszonylag nagy méretű, (több ezer sor) sok rész triviális, vagy csak technikai, azonban bizonyos osztályokat, vagy kódrészleteket behivatkozok, amikben úgy érzem, hogy említésre méltó dolgok találhatóak.

3.3. Program felépítése, tervezési szempontok

Kezdetektől fogva a Northcott-sakkot szerettem volna leimplementálni, ugyanis a Nim játékok variánsai közül látványosabbak közé tartozik, és nagyon tetszett az a tulajdonsága, hogy egy-az-egyben visszavezethető hagyományos Nim játékra. Ebből következően az implementálást nem is a Northcott-sakkal kezdtem, hanem a hagyományos Nim játékkal. Miután ezzel végeztem, akkor tértem át magára a szakdolgozatom gyakorlati feladatára, a Nortcott-sakkra. A visszavezethetőség szemléltetésére implementálás során különös figyelmet szenteltem, olyan szinten, hogy a Nortcott-sakk nem is különálló játékmodulként szerepel (ezzel kissé megszegve a program koncepcióját), hanem csupán egy új megjelenítési réteget ad a Nim játékhoz. Más szóval, amikor a felhasználó Northcott-sakkot játszik, akkor valójában a gép azt Nim játékként fogja fel. Konkrétan ugyanazt az osztályt (NimGameCore) használja felel a Nim játék, mint a Nortcott-sakk. Szándékosan még csak le sem származtattam, ezzel is kidomborítva a a Nortcott-sakk eme érdekes tulajdonságát.

A programot Java nyelven írtam, és használok benne olyan nyelvi elemeket, amik megkövetelik a 8-as verziót, azonban a SE (Standard Edition) eszközkészletén túlmenően semmilyen egyéb külső programkönyvtárat nem használtam fel.

A program belépési pontja a MainWindow osztályban található, ami gyakorlatilag létrehozza az alkalmazás főképernyőjét. A főképernyő áll egy menüsorból, egy oldalsó panelből, és egy főpanelből. A főpanelen helyezkedik el a létrehozott játék játéktere, míg az oldalsó panelen a létrehozott játék állapotpanelje, és a beállításpanelje. Ezek a panelok nem feltétlenül kell, hogy a java.swing.JPanel osztályból származzanak. Minden olyan komponenst be tud illeszteni ezekre a helyekre, amelyek a java.awt.Component osztályból származnak.

Minden játék két részből tevődik össze. Egyrészt magából a játékból, amely a GameCore osztályból kell, hogy származzon, másrészt egy olyan osztályból, amely megvalósítja a GameEntityProvider interfészt. Ez az interfész nyújt lehetőséget arra, hogy az adott játék beállításait lekérdezzük, továbbá ez az interfész kiterjeszti a GameUIProvider osztályt ami segítségével a már említett játék-specifikus komponenseket lehet lekérdezni. A játékot, és a GameEntityProvider-t a GameController osztály fogja össze, és egységesen egy játékként kezeli. Amennyiben egy futó játékot szeretnénk terminálni ezt rajta keresztül kell megtenni, ez gondoskodik a játék leállításáról, és a grafikus vezérlőelemek helyreállításáról.

A játékok felépítése hierarchikus. Mindegyik játék abból az osztályból származtat le, amelyik leginkább megfelel neki. Az alaposztály a GameCore, ami általános koncepciókat tartalmaz egy játékról mindenféle specializáció nélkül. A következő szint a

TurnBasedGame, ami a körökre osztott játékok használhatnak, és jelenleg erre épül a NimGameCore. Mindegyik hierarchia hozhatja magával az adott szinthez tartozó kiegészítő (esemény, kivétel) osztályokat.

3.4. Objektumok közti kommunikáció

Fontos rögtön az elején tisztázni, hogy a program jelenleg egy szálra van felkészítve, azaz sem a metódusok, sem pedig az algoritmusok nem szál-biztosak, nincsenek felkészítve a többszálú futásra, a hívások szinkron módon történnek. Ez jelenleg - a program egyszerűségéből fakadóan - nem okoz nagy problémát, egyetlen kellemetlenség van csupán, abban az esetben, ha két gépi játékost játszatunk egymás ellen, kellően nagy játékteret biztosítva számukra, akkor amíg a játékot játsszák a felhasználói felület az eseményekre nem reagál.

Kommunikációra kétféle módszert használunk. Általában ha egy esemény több objektumból érkezhet, de a célja mindig egy konkrét objektum, akkor ott interfészt definiálunk. Egyes esetben - ha a kommunikáció két irányú - nem is egyet, hanem mindjárt kettőt a vissza-irányba is. Amennyiben egy eseményt potenciálisan több helyre szeretnék eljuttatni, akkor az EventManager szolgáltatásait vesszük igénybe.

3.4.1. A program eseménykezelője

A programban egy nagyon pici, egyszerű eseménykezelőt implementáltam, a továbbiakban részletezem a felépítését, és használatát.

EventManager

Az EventManager egy nagyon pici statikus osztály, aminek az a feladata, hogy - igény esetén - EventChanneleket létrehozzon, a létrehozott EventChanneleket nyilvántartja, és igény esetén ezeket az objektumokat kérésre átadja, illetve ha egy EventChannel tulajdonosa úgy dönt, hogy megszünteti az EventChannelt, akkor ezen keresztül teheti meg ő, és csakis ő.

EventChannel

Az EventChannel feladata, hogy egyfajta csatornaként működjön. Erre a csatornára bárki feliratkozhat a subscribeForChannel(GameEventListener eventListener) metódus használatával. Paraméterben a feliratkozónak meg kell adnia önmagát, továbbá implementálnia kell a GameEventListener interfészt, ami gyakorlatilag egy visszahívó metódust tartalmaz. A visszahíváskor két paraméter adódik át. A csatorna azonosítója, illetve az eseményobjektum, ami a GameEventből kell, hogy származzon.

GameEvent

A GameEvent a játék eseményeinek az alaposztálya. Gyakorlatilag egy fontos dolgot tárol, hogy az esemény honnan származik. Minden kiegészítő információ tárolása a leszármaztatott objektumra van bízva. A GameEvent jelen dokumentáció írásakor még tartalmaz egy típus mezőt is, azonban azt fontolgatom, hogy ezt koncepciót kiveszem az eseményből, és az esemény típusát maga az objektum osztálya határozza meg.

Mint azt már korábban említettem a játékok követnek egyfajta hierarchiát, ami az eseményobjektumokon is megmutatkozik. Például a Nim játékok eseményeinek az alaposztálya a NimGameEvent, ami a TurnbasedGameEvent osztályból származik (ez adja hozzá a következő kör eseményt), ami pedig végül a GameEvent osztályból származik le.

3.5. Az absztrakt játék (GameEngine) szintje

Ez a program rétegződésének a legfelső szintje, gyakorlatilag minden osztálya absztrakt. Az itt található osztályok teljesen általánosan fogják fel a játék mibenlétét, ebből következően ezekből példányosítani értelmetlen, emiatt absztraktként vannak megjelölve. Mindazonáltal már ezen a szinten is bizonyos dolgokat ismerünk. Tudjuk például, hogy a játékot játékos(ok) játsszák, azok lépéseket tesznek meg, a játéknak van eleje, vége, és beállításai.

3.5.1. StepObject

Minden játékban léteznek valamiféle átmenetek, amik a játékteret a kezdőállapotból további állapotba, végül pedig a végállapotba viszik. Ezeket az állapotátmeneteket a StepObject objektumok reprezentálják. Az alap StepObject absztrakt osztály önmagában csak azt az információt tartalmazza, hogy ezt az átmenetet melyik játékos vitte véghez. Értelemszerűen a konkrét játéknak ki kell egészíteni ezt az osztályt a játék állapotátmeneteire vonatkozó információkkal.

3.5.2. Player

Egy tetszőleges játékost ír le ez az osztály. Ezen a szinten amit biztosan tudunk, hogy a játékosnak van egy neve, egy belső azonosítója, és egy PlayerControllerje, akitől szükség esetén lekérdezhetjük a játék beállításait, illetve a kontrolleren keresztül a játékot utasíthatjuk a következő lépés megtételére.

3.5.3. GameCore

Egy általános célú játék absztrakt megvalósítása. Nyilvántartja a játékosokat, eltárolja a lépéselőzményeket, a játék beállításait fogadni tudja, továbbá fenntart egy EventChannelt amire a program többi komponense feliratkozhat. A játék eseményeit ezen keresztül szórja szét.

3.5.4. AbstractGameSettings

Egy teljesen üres osztály, a játék beállításait tartalmazza, minden játéknak implementálnia kell a saját beállításait, melynek ebből az osztályból kell származnia.

3.6. Nim játék leírása

A Nim játékot megvalósító osztály a már említett NimGameCore. Az osztály, és a hozzá tartozó egyéb osztályok a gameplayer.nimgame.standard csomagban találhatóak.

3.6.1. NimPlayer

NimHumanPlayer

A NimHumanPlayer az emberi játékos implementációja. Gyakorlatilag egy üres implementáció, a cselekvés a felhasználói felületre van rákötve.

NimAIPlayer

Sokkal érdekesebb a NimAIPlayer osztály. Ez a gépi játékot megvalósító játékos implementációja. Látszik, hogy már egyből a konstruktorba létrehozom a mesterséges intelligenciát az adott játékospéldányhoz. Ez persze azt is jelenti, hogy több gépi játékos esetén több mesterséges intelligencia jön léte, ami magában hordozza annak a lehetőségét, hogy különböző MI implementációkat versenyeztessünk egymással.

A notifyYourTurn() visszahívó metódusban értelemszerűen meghívom a mesterséges intelligencia getNextStep() metódusát, ami egy NimAISolution objektumot tartalmaz. Ezután már nincs más dolgom mint ezt az objektumot felhasználva készítsek egy NimStepObject objektumot, és a kontrolleren keresztül léptessem a gépi játékost. A mesterséges intelligenciát megvalósító osztályok a gameplayer.nimgame.standard.AI csomagban találhatóak, de erről részletesebben a 3.8 részben írok.

3.6.2. NimGameSettings

A Nim játék lehetséges beállításait tartalmazza. A 3.1.4 és a 3.1.4 pontokban részletesen leírtam a programomban lévő Nim játék nyújtotta lehetőségeket. Ezeknek a lehetőségeknek a megvalósítása szerepel ebben az osztályban.

3.6.3. NimStepObject

A Nim játék leszármaztat az alap StepObject osztályból hozzáadva azt az információt, hogy az adott lépésből melyik halomból mennyi elemet vett el az állapotátmenetet előidéző játékos.

3.6.4. A Nim játék grafikus felhasználói felületelemei

A Nim játék grafikus elemeit a gameplayer.nimgame.standard.UI csomagban találjuk. A 3.3 szekcióban már leírtam, hogy legalább három grafikus komponenst kell a játékoknak implementálni, és itt ez a három komponens a NimMainPanel, NimSettingsPanel, és a NimStatusPanel.

NimMainPanel

Egy javax.swing.JPanel típusú grafikus komponens. Ez valósítja meg a játékteret, de egy összefogó szerepe is van. Implementálja a GameEntityProvider interfészt, és a segédpaneleket is lepéldányosítja, továbbá feliratkozik a Nim játék EventRegistry osztályban található EVENT_GAMEENGINE eseménycsatornára. Ennek köszönhetően a játéktérben bekövetkezett változásról azonnal értesül, és frissíteni tudja a játékteret.

3.7. Northcott játék leírása

A Northcott-sakk megvalósításának a szépsége, hogy ehhez a játékhoz nem tartozik játék, azaz nincsen hozzá olyan osztály, ami a GameCoreból származik le. Az egész játék csupán felhasználói felületelemekből épül fel - mint korábban írtam - a Nim játéknak ténylegesen csak egy másfajta megjelenítése.

A három panel itt is ugyan úgy megtalálható: NorthcottMainPanel, NorthCottSettingsPanel, és NorthcottStatusPanel. Itt is a főpanel látja el az összefogó szerepét. Sok szempontból analóg a viselkedése a Nim játék főpaneljével, így részletesen ezt nem is tárgyalom.

Akad azonban egy érdekes probléma. Az igaz ugyan, hogy a Northcott-sakkot vissza lehet vezetni a Nim játékra, ez fordítva azonban nem teljesen igaz. A Nim játékban a halmok elemszáma megfeleltethető ugyan a Northcott-sakkban a bábuk közt lévő távolságnak, azonban a bábuk pozíciójára vonatkozóan a Nim játék nem hordoz információt. Ezt a problémát áthidalandó az oszlopok nyilvántartanak egy eltolás értéket, ami azt mutatja meg, hogy a felső játékos bábuja a tábla tetejéhez képest mennyivel van lejjebb. Ehhez az eltolási értékhez hozzáadva a halom elemszámát megkapjuk az alsó játékos bábujának a helyét.

3.8. Nim nyerő stratégia gépi implementációja

A gépi játékost stratégiáját a AINimWinningStrategy osztály valósítja meg, amely implementálja a NimAI interfészt:

source/nimgame/standard/AI/NimAIStrategy.java

```java
/*
 * To change this license header, choose License Headers in
   Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package gameplayer.nimgame.standard.AI;

import java.util.List;
import gameplayer.nimgame.standard.exceptions.AIException;

/**
 *
 * @author Máté Pozsgay
 */
public interface NimAIStrategy {

    public NimAISolution getNextStep(List<Integer>
   heapConfiguration) throws AIException;
}
```

A visszaadott NimAISolution objektum gyakorlatilag egy tárolóobjektum amely a kiválasztott halmot, és az abból elvett elemszámot tartalmazza.

Maga a stratégia implementációja jól követi az elméleti részben tárgyalt stratégiát, de néhány helyen magyarázatot igényel. A belépési pont a getNextStep() metódus, ami paraméterben megkapja a halomlistát. Ezután ezt átalakítja tömbbé, és meghívja a getBestMove() metódust. Az átalakítás nem szükséges, de az adatstruktúrán igen sok műveletet hajt végre az algoritmus, így optimálisabb ezzel az átalakítással.A getBestMove() a gerince a stratégiának, itt dől el, hogy a mesterséges intelligencia melyik lépést választja a következőnek. Először is megkeresi az első nem üres halmazt, (ha mindegyik halmaz üres, akkor dobunk egy kivételt, az MI nem futtatható üres játéktéren) majd egyszerű próbálkozással nekiáll megkeresni az első olyan állapotot, amit a DecisionMaker elfogad. Minden halomra (ha nem üres) megpróbálja elfogadtatni az lépést úgy, hogy először elvesz az aktuális halomból egyet, majd kettőt, és így tovább amíg el nem fogy a halom. Ha az egész halmot elvette, de még mindig nem fogadta el a lépést a DecisionMaker, akkor visszaállítja a halom elemszámát a kísérletezés előtti állapotra, és lép a következő halomra.A DecisionMaker egy belső interfész, melyet két belső osztály implementál. Feladata az, hogy eldöntse, hogy az éppen vizsgált lépés elfogadható-e legjobb lépésnek. Az a két belső osztály ami implementálja az pont a NimStandardDecisionMaker, és a NimMisereDecisionMaker. Mint a nevükből kiderül az egyik a hagyományos Nim játéknál hoz döntést, míg a másik a Misère módhoz. Első megközelítésben nem sok értelme látszik ennek a fajta megoldásnak, hiszen ezt a vizsgálatot beleépíthettem volna a kereső algoritmusba, azonban ha így tettem volna, akkor minden ciklusfutáskor kellett volna tennem egy feltételvizsgálatot, hogy éppen melyik módban van a játék. Így azonban a konstruktorban elvégzem ezt a vizsgálatot, és a mezőhöz a megfelelő DecisionMaker objektumot rendelem hozzá, így a ciklusban az összes ilyen vizsgálat elhagyható tovább gyorsítva az algoritmust.Ez az algoritmus majdnem minden esetben talál nyerő állapotot, kivéve, ha az ellenfél kezdett, és hibátlanul játszik. Ebben az esetben nem tud mit csinálni, így - mivel nincs jó megoldás, de lépni kell - elvesz az első nem üres halomból halomból egyet.

source/nimgame/standard/AI/AINimWinningStrategy.java

```java
/*
 * To change this license header, choose License Headers in
   Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package gameplayer.nimgame.standard.AI;

import java.util.Arrays;
import java.util.List;
import gameplayer.nimgame.standard.exceptions.AIException;

/**
 *
 * @author Máté Pozsgay
 */
public class AINimWinningStrategy implements NimAIStrategy {

    private final DecisionMaker decisionMaker;

    public AINimWinningStrategy(boolean misereNim) {
        decisionMaker = misereNim ? new
   NimMisereDecisionMaker() : new NimStandardDecisionMaker();
    }

    private interface DecisionMaker {

        public boolean acceptSolution(int nimSum);
```

```java
    }

    private class NimStandardDecisionMaker implements
  DecisionMaker {

        @Override
        public boolean acceptSolution(int nimSum) {
            return nimSum == 0;
        }

    }

    private class NimMisereDecisionMaker implements
  DecisionMaker {

        @Override
        public boolean acceptSolution(int nimSum) {
            return nimSum == 1;
        }

    }

    private int getNimSum(int[] heapConfiguration) {
        int s = 0;
        for (Integer i : heapConfiguration) {
            s ^= i;
        }

        return s;
    }

    private int getFirstNonemptyHeapID(int[] heapConfiguration)
  {
        int testID = 0;
        while (testID < heapConfiguration.length &&
  heapConfiguration[testID] <= 0) {  // Look for the first
  non-empty heap
            testID++;
        }
        if (testID >= heapConfiguration.length) {
            return -1;
        }
        return testID;
    }

    private NimAISolution getBestMove(int[] heapConfiguration)
  throws AIException {
        int testID, testMove = 1, originalValue;
        boolean solutionFound = false;
        testID = getFirstNonemptyHeapID(heapConfiguration);
        if (testID < 0) {
```

```java
            throw new AIException("Attempt to execute on an
    empty gamespace!");
        }
        testID--;
        while ((!solutionFound) && (testID + 1 <
    heapConfiguration.length)) {
            testID++;
            if (heapConfiguration[testID] > 0) {
                originalValue = heapConfiguration[testID];
                heapConfiguration[testID]--;
                while (heapConfiguration[testID] >= 0 &&
    !decisionMaker.acceptSolution(getNimSum(heapConfiguration))) {
                    heapConfiguration[testID]--;
                    System.out.println("Testing: " +
    Arrays.toString(heapConfiguration) + "=" +
    getNimSum(heapConfiguration));
                }
                if (heapConfiguration[testID] >= 0) {
                    testMove = originalValue -
    heapConfiguration[testID];
                    System.out.println("Solution found.");
                    solutionFound = true;
                }
                heapConfiguration[testID] = originalValue;
            }
        }

        if (!solutionFound) { // Currently there is no winning
    move :(
            System.out.println("Faking solution.");
            testID = getFirstNonemptyHeapID(heapConfiguration);
            testMove = 1;
        }

        return new NimAISolution(testID, testMove);
    }

    @Override
    public NimAISolution getNextStep(List<Integer>
    heapConfiguration) throws AIException {
        return
    getBestMove(heapConfiguration.stream().mapToInt(i ->
    i).toArray());
    }

}
```

4. fejezet

Összefoglalás

A Nim játékok a játékelmélet egy specializált, s egyben a leggyakrabban vizsgált ágába a kétszemélyes teljes információjú stratégiai játékok családjába tartoznak. Bár számos változata létezik, jelen dolgozatban kimondottan a hagyományos Nim játékkal és a Northcott-sakkal foglalkoztam, azon belül is az utóbbinak azzal az érdekes tulajdonságával, hogy egy az egyben visszavezethető a hagyományos Nim játékra.

A játék implementálása során az elsődleges célomat sikerült elérnem, a Northcott-sakk Nim játékra való visszavezethetőségét a gyakorlatban is sikeresen igazoltam, méghozzá nem is akárhogyan. Sikerült elérnem, hogy a Northcott-sakk ténylegesen csupán egy másfajta vizualizációja legyen a Nim játéknak, ezzel még szemléletesebbé téve ezt a tulajdonságát.

A Nim játékhoz készített mesterséges intelligencia - ami gyakorlatilag a nyerő stratégia egy implementációja - szinte legyőzhetetlen nehézségű ellenfelet jelent bármilyen emberi játékos számára.

A másodlagos célomat is sikerült részben elérnem. A megírt program hierarchikus és moduláris felépítése lehetővé teszi, hogy a szoftvert bármilyen egyéb - nem feltétlenül csupán Nim - játékkal kényelmesen ki lehessen bővíteni. Ez gyakorlatilag egy felhívás arra, hogy a többi Nim variánst (a hozzájuk tartozó mesterséges intelligenciával) is hozzáadjuk a programhoz létrehozva egy komplett gyűjteményt így tisztelegve ennek a minden tekintetben ősi játéknak, és az ezekből kifejlődő egyéb más játékoknak.

A szakdolgozat témájához ugyan nem kapcsolódik, de a program megírása során született néhány "melléktermék", melyek kellően általánosak ahhoz, hogy máshol is fel lehessen ezeket használhatni. Ilyen például a DisableableJPanel, amelyet egy adott panel, és a benne található összes gyerekkomponens letilthatóságának az igénye hívott életre. Sajnos készen ilyen komponenssel nem találkoztam, ezért készítettem el a DisableableJPanel-t, ami egy olyan JPanel, melynek a benne foglalt komponensei rekurzívan letilthatóak, ráadásul azt is intelligens módon teszi, azaz letiltáskor csak azokat az elemeket tiltja le, melyek nem voltak korábban letiltva, és feloldáskor pedig visszaállítja az eredeti állapotot, azaz a már korábban (panel tiltása előtt) letiltott vezérlőelemek a panel feloldása után is letiltva maradnak.

Irodalomjegyzék

[1] Thomas Fisher: *Simulating the Pick-up Stones game: A dynamic approach*, Department of Computer Science and Electrical Engineering, University of Maryland Baltimore Country, http://www.users.miamioh.edu/fishert4/docs/fisher-algo.pdf

[2] Flesch, Rudolf (1951). *The Art of Clear Thinking.* New York: Harper and Brothers Publishers. 3. oldal

[3] Charles L. Bouton: *Nim, A Game with a Complete Mathematical Theory*, Annals of Mathematics (http://www.jstor.org/stable/1967631)

[4] Russell Stuart és Norvig Peter: *ARTIFICIAL INTELLIGENCE. A MODERN APPROACH. 2nd Edition*, Pearson Education, Inc (Magyar nyelvű fordítás: *https://mialmanach.mit.bme.hu/aima/index*)

[5] Nyerő stratégia és bizonyítása: *https://brilliant.org/wiki/nim/*

[6] Nim játék variánsai: *https://mialmanach.mit.bme.hu/erdekessegek/nim_jatek*

[7] Turing-teszt: *https://hu.wikipedia.org/wiki/Turing-teszt*

[8] Nimberek: *https://en.wikipedia.org/wiki/Nimber*,
https://en.wikipedia.org/wiki/Ordinal_number,
https://hu.wikipedia.org/wiki/Rendsz%C3%A1m_(halmazelm%C3%A9let)

[9] Minimax: *https://hu.wikipedia.org/wiki/Minimax_elv és [4]*

[10] Alfa-Béta vágás: *https://hu.wikipedia.org/wiki/Alfa-b%C3%A9ta_v%C3%A1g%C3%A1s és [4]*

Adathordozó használati útmutató

Ebben a fejezetben kell megadnunk, hogy a szakdolgozathoz mellékelt adathordozót (pl. CD) hogyan lehet elérni, milyen strukturát követ. Minimum 1 maximum 4 oldal a terjedelem. Lehet benne több alszakasz is. A fejezet címe nem módosítható, hasonlóan a következő részhez (Irodalomjegyzék).