

Szakdolgozat

Majorics Tamás

Debrecen

2008

Debreceni Egyetem
Informatikai Kar

Játékprogram írása: Go

Témavezető:
Dr. Halász Gábor
Egyetemi docens

Készítette:
Majorics Tamás
Programozó Matematikus

Debrecen
2008

Tartalomjegyzék

Bevezetés	5
Témaválasztás indoklása	5
A Go-ról	5
Célkitűzések.....	6
A Go.....	7
Történelem.....	7
Játékszabályok	7
Fogalmak	7
A játék menete.....	8
Végeredmény megállapítása.....	9
Stratégia.....	9
Játékosok rangsorolása	10
A program	11
Az alkalmazott programozási nyelv	11
A program szerkezete, felépítése	11
A program logikai felépítése	12
Grafikai rész	12
Algoritmikus rész	12
A reprezentáció	16
A feladat állapotter-reprezentációja	16
A reprezentáció konkrét megvalósítása	19
Hasznosság.....	24
Elméletben	24
Gyakorlatban	24
Kinézet	26
A menü	26
A fejlesztés során felmerült problémák.....	29
Programozás technikai problémák	29
Set	29
Grafika.....	30
Szemantikai hibák.....	30
Összefoglalás.....	33
Irodalomjegyzék.....	35
Írott anyagok	35
Elektronikus anyagok.....	35
Függelék.....	36
Kódok.....	36

A <i>Jatek run</i> metódusa	36
A GoAllapot alkalmaz metódusa	38
A MinimaxAlgoritmus konstruktora	43

Bevezetés

Témaválasztás indoklása

Amikor megkezdtem tanulmányaimat a Debreceni Egyetemen, mint elsőéves programozó matematikus hallgató, egy sor új tantárggyal találtam szemben magam. Ami elsőre megtetszett, az a „Matematikai logika alapjai” című tárgy volt. Én amúgy is egy logikus gondolkodású személynek tartom magam, ezért a matematikai logika egyből megfogott. Későbbi tanulmányaim során igyekeztem logikai tárgyakat felvenni („Nem klasszikus logikák”, „Haladó logika”), és mivel kötelező tantárgy, ezért megismerkedtem a Mesterséges intelligencia alapjaival is. Mivel ez alapvetően a matematikai logikán alapul, ezért nem volt meglepő, hogy ez a tantárgy is a kedvenceim közé tartozott. Aztán elérkezett az idő, hogy szakdolgozat témát válasszak. A meghirdetett témák között láttam meg a „Játékprogram írása: Go” című témát. Gondoltam, na ez az a téma, ami nekem való. Ez lenne az oka a témaválasztásomnak.

A Go-ról

Alapvetően a mesterséges intelligencia miatt választottam a témát, a Go-ról azelőtt nem nagyon hallottam. A Go egy ősi kínai táblajáték. Ott nagyobb tradíciója és népszerűsége van, mint Európában a sakknak. Legjobban a Sakkhhoz hasonlítható, de itt minden bábú egyenértékű. Éppen ezért ezeket nem bábuknak, hanem köveknek nevezzük. Amíg a sakkban egy kitüntetett bábú, a király leütése a cél, addig a Go-ban a legfőbb cél a területszerzés. Ebben is nagy szerepe van a stratégiának, és a tapasztalatnak. A Go egyszerűbbnek tűnik első ránézésre, mint a sakk, hiszen egyszerűbbek a lépések, az azonos színű bábúk nem különböznek egymástól sem erősségben, sem viselkedésben. Ennek ellenére mégis összetettebb, bonyolultabb gondolkodásmódot igényel. Programozhatósága is nehezebb, hiszen ha csak a lehetséges lépéseket nézzük, a sakktáblán 64 pozíció van a Go 361 rácspontjával szemben. A mesterséges intelligenciában alkalmazott algoritmusok sorra veszik az egy adott állásban léphető összes lépést, és ezekből kiválasztanak egyet valami alapján. Hogy mi alapján, az a kulcsa az egésznek. Tehát az egyik ok, amiért nehéz leprogramozni, hogy nagyon nagy sok számolás szükséges ahhoz, hogy elegendően sok lépést előre kiszámoljunk. Az elegendő alatt azt értem, amennyi elég mondjuk egy profi Go játékos legyőzéséhez.

Célkitűzések

Mivel a hagyományos Go leprogramozása igencsak nehéz feladat, ezért ebben a dolgozatban egy kisebb, 9x9-es táblát használunk. Ezáltal az egy adott állásnál megtehető lépések száma lecsökken, hiszen csak 81 rácspont lesz a táblán. Ez nagyban lerövidíti az egyes lépések kiszámításához szükséges időt. A játék az első leütésig tart, szemben a hagyományos Go-val, ahol a játék végét a játékosok közös megegyezéssel határozzák meg. Hogy pontosan mi is egy leütés, és mit értünk „közös megegyezésen”, azt a későbbiekben, a játékszabályoknál tárgyaljuk. Alapvető célunk egy működő játékprogram megírása. Erre a Java programnyelvet használjuk. A program egy grafikus felületen fogja ábrázolni a táblát, és a rajta lévő köveket fekete, illetve fehér korongokkal. A játékos ezen a grafikus felületen teheti meg a lépést az egér segítségével. Azért választottam a Java-t, mert ennek ismerem legjobban a grafikai megjelenéshez szükséges részét.

A Go

Történelem

A Go kínai eredetű, körülbelül négyezer éves táblás játék. Pontosán nem tudjuk, hogy ki, és hogyan talált ki. Különböző legendák léteznek. Az egyik kínai legenda szerint Yao császár találta ki a fia fegyelemre és koncentrációra tanítása céljából, míg egy másik szerint viszont Shun császár találta ki ugyanezen okból. Létezik egy japán változat is, miszerint két testvér találta ki, hogy az örökölt földjüket elosszák. Egy biztos, ez a legrégebbi ma is játszott táblás játék, ugyanis már időszámításunk előtti 6. századból vannak feljegyzések a Go-ról, melyek Konfuciusz-tól származnak. A 17. század elején megalakult a Haninbo-iskola, melynek vezetője, és névadója Haninbo Sansa, a kor legerősebb Go játékosa volt. Ekkor nagy fejlődésnek indult a játék, és kialakult a játékosok értékelésének rendszere is, mely hasonló a harcművészetekéhez. A népszerűsége egyre nőtt, ma már többen játsszák a világon, mint a sakkot.

Magyarországon körülbelül 30 éve vált ismertté, és az utóbb időben egyre nő a népszerűsége. Ma már több olyan magyar játékos is van, akiket a nemzetközi Go-körökben is elismernek.

Játékszabályok

Fogalmak

Egy kő *életeinek* száma, a négy szomszédos mezői közül az üreseknek a száma.

Egy *csoport* vagy *sztring* olyan azonos színű kövekből áll, melyek négyszomszédosan kapcsolódnak egymáshoz.

Egy *csoport életei*, a csoportba tartozó kövek életeinek halmaza.

Fogoly kőről vagy *fogoly csoportról* akkor beszélünk, ha az adott követ vagy csoportot az ellenfél négyszomszédosan körbezárta, azaz az adott kő vagy csoport életeinek száma nulla. Ez esetben az adott kő vagy kövek lekerülnek a tábláról.

Az olyan helyzetet, melyben a játékosok felváltva újra meg újra leüthetik egymás köveit *ko*-nak nevezzük. Az a játékos, akinek követ egy ko-ban leütöttek, a következő lépésekor nem üthet vissza ebben a ko-ban.

Szem-pontok-nak nevezzük azokat az üres pontokat, melyeket kizárólag az egyik játékos élő kövei határolnak.

Az egyéb üres pontok neve *dame*.

Azok az élő csoportok, melyek *dame* pontokat tartalmaznak *szeke*-ben vannak.

Azon szem-pontokat, melyeket élő, de nem *szeke*-ben lévő csoportok vesznek körül, *terület*-nek nevezzük, melynek minden egyes szem-pontja egy terület-pontnak számít.

Azok a kövek *élő*-nek tekintendők, amelyeket az ellenfél nem tud leütni, vagy ha leütik, ez egy olyan kőnek a táblára helyezését teszi lehetővé, melyet az ellenfél nem tud leütni.

A nem élő köveket *halott*-nak nevezzük.

A játék menete

A tábla 19 vízszintes és 19 függőleges vonalból álló négyzetrácsos felület. Két játékos játszik egymás ellen fekete, illetve fehér kövekkel. Egy lépés abból áll, hogy egy saját színű követ helyezünk a tábla egy üres rácspontjára. A játékot a fekete kezdi, ezután felváltva lépnek. A már lerakott köveket nem lehet elmozdítani, kivéve persze, ha leütik őket. Lépéskényszer nincs, tehát a játékosok passzolhatnak. Nem lehet lerakni olyan helyre követ, ahol annak nem lenne élete, vagyis azonnal fogoly kő válna belőle. Kivétel, ha a lerakott kő elfog egy vagy több követ az ellenfél kövei közül, de ez esetben az elfogott kövek lekerülnek a tábláról, így az általuk hagyott szabad helyek között biztos, hogy lesz olyan, mely a lerakott kő életévé válik. Tulajdonképpen ez az eset sem tekinthető igazi kivételnek, hiszen lerakáskor ugyan nincs élete a kőnek, de az általa foglyul ejtett kövek levétele után már van.

A fogalmaknál már említettük a *ko*-t, ez esetben a pontos megfogalmazás úgy szól, hogy nem állhat elő a két lépéssel előbbi helyzet. Ha egymás után mindkét játékos passzol, akkor a játék megáll. Ezután a két játékos megegyezik egymással az élő, illetve halott kövekről, és a területekről. Ha egyikük nem ért egyet a másikkal, és folytatni kívánja a játékot, akkor a másiknak kötelessége folytatni azt, és őt illeti a következő lépés joga. Erre utal a Bevezetőben említett „közös megegyezés” kifejezés. Tehát akkor beszélhetünk a játék végéről, amikor a felek megegyeznek. Bármelyik játékos bármikor befejezheti a játékot, ha feladja azt, és ezzel elismeri a vereségét. Ekkor azt mondjuk, hogy az ellenfele feladással győzött. Befejeződhet a játék egy speciális esetben is, ha ugyan az az állás ismétlődik a teljes táblán, ekkor a játék végeredmény nélkül fejeződik be.

Végeredmény megállapítása

Japán módszer:

Ez azzal kezdődik, hogy mindegyik játékos leveszi a saját területéről az ellenfél halott köveit, és a saját foglyaihoz teszi azokat. Ezután a foglyokat az ellenfél területeire teszik be, majd megszámlálják a területeket, és az így kapott eredmények lesznek az egyes játékosok pontjai. Az a győztes, akinek több pontja van.

Kínai módszer:

Itt is először a halott köveket távolítják el. Ezután összeadják a fennmaradt élő kövek számát a területpontok számával, és ez szolgáltatja a játékosok pontjait. Szintén az a győztes, akinek több pontja van.

A két módszer tehát abban különbözik, hogy a japán esetében a területpontokat csökkentik az ellenfél által foglyul ejtett kövek számával, a kínai módszernél pedig növelik a saját élő kövek számával. A két módszer a legtöbb esetben hasonló eredményt szolgáltat, csak néhány speciális esetben tér el a végeredmény.

Stratégia

A játék három részre bontható úgymint, *megnyitás, középjáték és végjáték*.

A *megnyitás* során a tapasztaltabb játékosok először stratégiai pontokat foglalnak el szétszórtnan, nem próbálnak meg egy részt gyorsan elfoglalni. Ezek a pontok a sarkok közelében vannak. Általában ide helyezkednek először, majd tovább a széleken. A tábla közepének inkább a középjátékban van szerepe.

A *középjátékban* kerül át a hangsúly a tábla belsejére. A játékosok igyekeznek a már meglévő területek mellé még több területet elfoglalni, az ellenfél köveit foglyul ejteni.

A *végjátékra* sok esetben csak akkor kerül sor, ha a felek között kicsi a pontszámkülönbség. Nagy különbség esetén gyakoriak a feladások.

Egy játékos előnye lehet terület vagy úgynevezett potenciál, vagyis az alakzatainak erősségéből adódó előny. Ezek egyike a másikba átvihető. A játékos dolga, hogy megtalálja a helyes egyensúlyt. Az helyes egyensúly megtalálása fontos az egész játékban, terület-potenciál, védekezés-támadás. A játékban vannak úgynevezett közmondások, amik majdnem minden esetben hasznosnak bizonyulnak, mint például: „Az ellenfeled lépésére válaszolj”. Azért majdnem minden esetben, mert van mikor egy kő feláldozása nagyobb előnyt jelent,

mint a kő értéke volt. Akkor válik igazán jó játékosná valaki, amikor megérzi, hogy mikor kell a „jól bevált” lépés helyett egy nem annyira jónak látszót lépni, mert a későbbiekben előnyhöz juttathatja.

A következő szabályokat Ji Xin írta körülbelül 1300 éve.

1. Tan Bu De Sheng - A kapzsiság nem vezet győzelemhez.
2. Ru Jie Yi Huan - Ne siess betörni az ellenfél területére!
3. Gong Bi Gu Wo - Figyelj az egyikre, amíg támadod a másikat!
4. Qi Zi Zheng Xian - Áldozz fel egy követ a sente-ért! (sente = kezdeményezés)
5. She Xiao Jiu Da - Hagyd ott a kicsit, hogy megmentsd a nagyot!
6. Feng Wei Xu Qi - Ha veszélyben vagy áldozz fel valamit!
7. Shen Wu Qing Su - Építs sűrű csoportot, tartózkodj a gyors lépésektől!
8. Dong Xu Xiang Ying - Az ellenfeled lépésére válaszolj!
9. Bi Qiang Zi Bao - Erő ellen biztonságosan játssz!
10. Shi Gu Qu He - Keresd a békét, ne harcolj elzárt vagy gyenge szituációban!

Játékosok rangsorolása

Mint már említettük, a játékosoknak rangsorolása hasonlít a küzdősportokban megszokottakhoz. Az alapszint a 30 kju-s játékos, aki éppen a szabályokat ismeri. Ezután a kju-k száma csökken, ahogy a tudás nő. Az 1 kju után következik az 1 dan. A dan már mester fokozat, és ez a tudással nő egészen 7 dan-ig. Ezeket amatőr dan-nak nevezzük. A 7 amatőr dan-os játékos körülbelül egy szinten áll az 1 dan-os profi játékosal. A profik fokozata 9 dan-ig nőhet. Úgy tartják, hogy az egyes rangok között a különbség 1 kő. Tehát például, ha az 1 kju-s játékos 2 kő előnnyel kezd egy 3 kju-s játékosal szemben, akkor kiegyenlítették az esélyek. Ugyan így a dan-os rangoknál is 1 dan kiegyenlíthető 1 kő előnnyel. A versenyeken ezt használják is. Ezt nevezik handicap-nek. A legnagyobb adható előny 9 kő. Az előnyt minden esetben fehér adja feketének, fekete mindig a gyengébb játékos. Van egy olyan elv is, miszerint 1 kő előny a játék elején, 10 pontot ér a játék végén. Ez esetben is a maximálisan adható előny 9 kő, azaz 90 pont.

A program

Az alkalmazott programozási nyelv

A program megvalósításához a Java programozási nyelvet használtam, mégpedig az 1.4-es verzióját. Eredetileg az 1.5-ös verzióban kezdődött a program fejlesztése, de a későbbiekben a különböző számítógépeken lévő különböző verziójú Java futtató rendszerek kompatibilitási gondjaiból adódó problémák elkerülése érdekében az 1.5-ös verzióban megjelenő új dolgok kikerültek a programból, és az 1.4-es verzióban megtalálható eszközökkel helyettesítődtek. (Lásd *Fejlesztési problémák* című alfejezetben)

Mivel a program grafikus, ezért a szóba jöhető programozási nyelvek a Java, C++, illetve C#. Létezik még sok más nyelv is, de nálam csak ezek jöttek számításba. C++-t nem használtam még, csak a C-t, de annak is jó ideje már, így ezt a megoldást elvettem. A C# egy mostanában igen „felkapott”, objektumorientált nyelv. Ezt már kicsit jobban ismertem, mind elméletből (Programozás 2), mind gyakorlatból (Komputergrafika). De túl nagy programozási gyakorlatra nem tettem szert benne, ugyanis csak egy grafikai programot írtam ezen a nyelven. Maradt tehát a Java. Ezen választás előnye, hogy ezt a nyelvet már jobban ismertem. Igaz ugyan, hogy grafikai programot nem sokat készítettem, csak egy egyszerű Windows ablakos kliens programot a *Hálózati architektúrák és protokollok* nevű tantárgy keretében, de mivel a Java nyelvhez egy nagyon jól használható referencia leírás létezik, az API, ezért ez nem okozott akkora problémát. Másik nagy előnye, hogy mesterséges intelligenciából az algoritmusokat szintén ezen a nyelven tanultuk. Mindemellett a Java programok platform függetlenek, azaz hordozhatóak, igaz ugyan, hogy a már említett verziók közötti kompatibilitásra figyelni kell.

A program szerkezete, felépítése

A Java egy objektum orientált nyelv. A program osztályokból épül fel. Ezek az osztályok két csomagban helyezkednek el, idegen szóval package-ben. Az egyik a *go*, a másik a *jatek* package. Nézzük először a *go* package-et. Ebben három osztály található, a *GoAllapot*, a *GoLepes* és a *GoMain*. A *GoAllapot* a mesterséges intelligenciában alkalmazott állapotot tartalmazza, a *GoLepes* pedig az operátort. A *GoAllapot* tartalmaz még egy belső osztályt is, a *Pozicio* osztályt. Ezekről később bővebben lesz szó, mikor a program logikai felépítését

tárgyaljuk. *GoMain*-ben van a főprogram, a *main* metódus. A *GoMain* osztálynak a grafikában is van szerepe, de erről is majd később bővebben esik szó.

A másik package-ben, a *jatek*-ban hat osztály található. A *Jatek* osztálynak kettős szerepe van, egyrészt a grafikai megjelenítésben játszik szerepet, másrészt, maga a játék ezen osztály egyik metódusában zajlik. Tulajdonképpen ez tekinthető a program gerincének, igaz ugyan, hogy nem itt van a főprogram, de itt történik a legtöbb dolog. A *LepesAjanlo* egy absztrakt osztály, mely adattagokat deklarál. Ebből az osztályból származtatjuk a maradék négy osztályt, melyek az algoritmusokat valósítják meg, úgymint minimax, negamax, valamint ezek alfabétavágással működő változatai.

A program logikai felépítése

Ezt két részre bonthatjuk, úgymint grafikai rész, és mesterséges intelligenciához kapcsolódó rész, nevezzük ezt röviden algoritmikus résznek. Igaz ugyan, hogy az egyes osztályokban ez a két funkció keveredik, de logikailag mégis áttekinthetőbb, ha külön tárgyaljuk őket.

Grafikai rész

Alapvetően a játék egy ablakban jelenik meg. Ezt a *GoMain* osztály valósítja meg, ami egy *JFrame*. Tehát a főprogram létrehozza a *GoMain* osztály egy példányát, ezáltal előáll az ablakunk. Minden ablakkal kapcsolatos teendőt a *GoMain* osztály lát el. Implementál három *Listener*-t, úgymint *ActionListener* a menü kezelésére, *MouseListener*, az egér kezelésére, valamint egy *ItemListener*-t, a menüben lévő kiválasztó menüpontok kezelésére (Például nyelv, Angol vagy Magyar). Ebben hozzuk létre a menüt, és itt kezeljük a menü gombjainak hatását. A kirajzolást egy *JPanel*-en végezzük el. Ezt a funkciót a *Jatek* osztály látja el. Ezen a *JPanel*-en kívül csak egy másik komponensünk van, egy *JTextArea*, amelyen a program üzeneteit jelenítjük meg. Magát a kirajzolást a *Jatek* osztály *paint* metódusa végzi, mely megrajzolja az üres táblát, majd az aktuális állapot segítségével a táblán lévő köveket. Tulajdon képen az egész grafikai részt ez a két osztály végzi.

Algoritmikus rész

Ebben a részben tekintjük át a program működését. Amikor elindítjuk a programot, akkor nem kezdődik egyből el egy játszma, hanem azt a Játék menü Új játék menüpontjával tudjuk elindítani. Ennek a menüpontnak a hatására az éppen futó játszma, ha van ilyen, véget ér, és a

Jatek osztály példányának adattagjai a menüpontok beállításainak megfelelő értéket vesznek fel. Tehát ha átállítunk valamilyen beállítást, az akkor fog érvényre lépni, mikor megkezdünk egy új játékot. A Jatek osztály implementálja a *Runnable* interface-t, és a *run* metódusa vezérli magát a játszmat. Ezért egy új játszma kezdetekor csak az éppen futó szálát kell leállítani, és indítani egy másikat, nem pedig a Jatek osztály egy új példányát létrehozni. A *run* tárol egy állapotot, ami az elején felveszi a kezdőállapotot, ami tulajdonképpen az üres tábla.

Ezután indul egy ciklus, ami addig fut, amíg az állapot nem végállapot. Ha az állapot végállapot lett, akkor kilép a ciklusból, megnézi, hogy ki nyert, ezt kiírja, majd a *run*, és ezzel a szál befejezi működését. A cikluson belül van egy elágazó utasítás, mely eldönti, hogy a gép lép, vagy mi.

Amennyiben a gép következik lépni, akkor megvizsgálja a beállításoknak megfelelően, hogy melyik lépésajánlót kell alkalmazni. Itt meg kell említenünk, hogy a programkód tartalmazza ugyan a negamax algoritmust mind alfa-béta vágással, mind nélküle, de a programból ezen beállítási lehetőség kimaradt, mert a heurisztika, attól függően, hogy ki következik lépni, másképpen határozódik meg. Így tehát itt csak azt vizsgálja, hogy kell-e alfa-béta vágást alkalmazni, vagy sem. A megfelelő lépésajánló algoritmus szolgáltat egy operátort, hogy pontosan mi alapján, azt később részletesebben tárgyaljuk. Ez az operátor letárolódik.

Amennyiben mi következünk lépni, akkor itt következik egy újabb feltételvizsgálat, ami szintén egy tervbe vett, de meg nem valósult funkciót vezérel. Nevezetesen, hogy a programtól lehet lépésajánlatot kérni. Ez ugyanazzal a mechanizmussal zajlana, mint mikor a gép következik lépni, és a lépésajánló által szolgáltatott operátort írná ki a program. Ennek inkább csak tesztelés esetén lenne jelentősége, ezért ez a funkció fejlesztés során időhiány miatt kimaradt. Ezt követően a szálát úgymond „elaltatjuk” a *wait* metódus segítségével, és csak akkor „ébresztjük fel”, mikor egy lépést adunk meg az egér segítségével. Mikor felébred, megvizsgálja, hogy a kapott operátor végrehajtható-e. Ha nem, akkor „visszaalszik” egy újabb lépés beérkezéséig. Amennyiben a kapott operátor alkalmazható, úgy ez is letárolódik, és véget ér az elágaztató utasítás.

Ezt követően az operátort alkalmazzuk az állapotra, létrejön egy új állapot, ami felülírja az eredetit, itt történik egy üzenet kiírása arról, hogy ki a soron következő játékos, és következik egy újabb cikluslépés, amely elején megvizsgáljuk a kapott állapotra, hogy végállapot-e.

Ezzel a program futását tulajdon képen meg is tárgyaltuk. Hátra van még a minimax algoritmus ismertetése, hiszen az előbbi leírásban csak érintőlegesen említettük.

Egy kis elmélet:

Az algoritmus célja a támogatott játékosnak, J-nek egy adott állásban „elég jó” lépést ajánlani.

Az algoritmusnak a következőkre van szüksége:

- A játék $\langle A, \text{kezdő}, V, O \rangle$ reprezentációjára
- J azon a állására, ahol lépni következik
- A $h_J: A \rightarrow \mathbb{R}$ heurisztikára, amely az állások „jóságát” J szempontjából becsli
- Valamint egy mélységi korlátra

Az algoritmus megadása pszeudó kóddal:

```
1:  function MINIMAX-LÉPÉS( $\langle A, \text{kezdő}, V, O \rangle$ , állapot, korlát,  $h_J$ )
2:     $\text{max} \leftarrow -\infty$ 
3:    operátor  $\leftarrow \text{NIL}$ 
4:    for all  $o \in O$  do
5:      if ELŐFELTÉTEL(állapot,  $o$ ) then
6:        új-állapot  $\leftarrow \text{ALKALMAZ}$ (állapot,  $o$ )
7:         $v \leftarrow \text{MINIMAX-ÉRTÉK}$ ( $\langle A, \text{kezdő}, V, O \rangle$ , új-állapot, korlát-1,  $h_J$ )
8:        if  $v > \text{max}$  then
9:           $\text{max} \leftarrow v$ 
10:         operátor  $\leftarrow o$ 
11:       end if
12:     end if
13:   end for
14:   return operátor
15: end function
16: function MINIMAX-ÉRTÉK( $\langle A, \text{kezdő}, V, O \rangle$ , állapot, mélység,  $h_J$ )
17:   if állapot  $\in V$  or mélység = 0 then
18:     return  $h_J(\text{állapot})$ 
19:   else if JÁTÉKOS[állapot] = J then
20:      $\text{max} \leftarrow -\infty$ 
21:     for all  $o \in O$  do
```

```

22:      if ELŐFELTÉTEL(állapot, o) then
23:          új-állapot  $\leftarrow$  ALKALMAZ(állapot, o)
24:          v  $\leftarrow$  MINIMAX-ÉRTÉK(<A, kezdő, V, O >, új-állapot, mélység-1, hJ)
25:          if v > max then
26:              max  $\leftarrow$  v
27:          end if
28:      end if
29:  end for
30:  return max
31: else
32:     min  $\leftarrow$   $\infty$ 
33:     for all o  $\in$  O do
34:         if ELŐFELTÉTEL(állapot, o) then
35:             új-állapot  $\leftarrow$  ALKALMAZ(állapot, o)
36:             v  $\leftarrow$  MINIMAX-ÉRTÉK(<A, kezdő, V, O >, új-állapot, mélység-1, hJ)
37:             if v < min then
38:                 min  $\leftarrow$  v
39:             end if
40:         end if
41:     end for
42:     return min
43: end if
44: end function

```

A minimax algoritmust a *MinimaxAlgoritmus* nevű osztály valósítja meg, ami a *LepesAjanlo* osztály egy leszármazottja, mint ahogy azt már korábban említettük. Az osztálytól csak adattagokat örököl, nevezetesen két konstanst, egy plusz és egy mínusz végtelent. Természetesen ezeknek az értéke nem végtelen, hanem a konstans típusának megfelelő maximum, illetve minimum érték. Egy *allapot*-ot, egy *operator*-t, valamint három int típusú adattagot, melyek *melyseg*, *hasznossag* és *allasokSzama*. Ez utóbbi csak egy statisztikai adat, mely tesztelés esetén hordozhat bizonyos információkat. Magában a *MinimaxAlgoritmus* osztályban sincs metódus, csak egy *toString*, az esetleges kiírásokhoz. Az algoritmust az

osztály konstruktora tartalmazza. Láthatjuk, hogy az algoritmusban a MINIMAX-ÉRTÉK függvény rekurzív. A mi esetünkben a rekurziót az osztály konstruktora végzi úgy, hogy példányosítja saját osztályát, azaz saját magát hívja meg. A mi esetünkben a reprezentációt és a h_j hasznosságot nem kell külön átadnunk a konstruktornak, hiszen ezeket az állapot tartalmazza. Elegendő tehát csak az állapotot, és a mélységet megadnunk. A kód többi része megfelel a pszeudó kódú algoritmusnak.

A reprezentáció

A reprezentáció elég nagy fejtörést okozott. Mérlegelni kellett ugyanis, hogy egy állapot „jóságának” megállapításakor töltsünk el sok időt arra, hogy kiszámoljuk a hasznosságot, vagy az állapotban tároljunk olyan plusz információkat, melyek megkönnyítik a számolást. Az első esetben nem tárolunk plusz információkat, így takarékoskodunk a memóriával, viszont megnő a számolási idő, hiszen mindig mindent előlről ki kell számolni. A második esetben több memória szükséges a plusz információk miatt, viszont a számolási idő rövidül, hiszen nem kell mindent újra kiszámolni, csak a változásoknak megfelelően frissíteni kell az adatokat. Mivel a memóriaigényt a felhasználó nem igazán veszi észre, annyira ugyanis nem nagy, hogy kihasson az operációs rendszerre, de az idő igényt viszont annál inkább, ezért a második megoldás mellett döntöttem. Emiatt a GoAllapot osztály elég sok adattaggal bővült ki, és az *alkalmaz* függvény is eléggé bonyolult sikerült, hiszen ez végzi az adatok karbantartását.

A feladat állapotter-reprezentációja

A probléma állapottere ($\mathcal{A}$):

Minden pozícióhoz tárolunk egy számot (0, 1, 3).

- Ha üres az adott pozíció, akkor 0-t.
- Ha fekete kő van rajta, akkor 1-et.
- Ha fehér, akkor 3-ast.

Ennek megfelelően:

$$H_{11}=H_{12}=H_{13}=\dots=H_{19}=$$

$$H_{21}=H_{22}=H_{23}=\dots=H_{29}=$$

.

$$H_{91}=H_{92}=H_{93}=\dots=H_{99}=\{0, 1, 3\}$$

Tároljuk még, hogy melyik játékos következik:

$$J=\{1, 3\}$$

A probléma állapottere a $H_{11} \dots H_{99}$ és J halmazok Descartes-szorzataként előálló halmaz lesz. Ezen halmaz minden eleme valódi állapot lesz, ezért nem szükséges kényszerfeltételeket megadni.

Egy állapotunk tárol még egyéb információkat is, de ezek az állapotteret nem bővítik ki, mivel ezeket az információkat a már letárolt értékekből számoljuk.

A négyszomszédosan egymáshoz kapcsolódó azonos színű kövek sztringeket alkotnak. Egy sztringhez négyszomszédosan kapcsolódó üres pozíciók halmazát a sztring életeinek nevezzük.

Tárolunk két tömböt, mindegyik színhez egyet, melyek halmazokat tartalmaznak. $T_b[]$, a feketékhez és $T_w[]$, a fehérekhez.

$$T_b = T_w = \{S_0, S_1, S_2 \dots\}$$

Egy-egy tömbpozíción egy-egy sztringhez tartozó életek halmaza található, melyek jelölése: $S_0, S_1, S_2 \dots$

Azt, hogy melyik kő melyik sztringhez tartozik, külön mátrixban tároljuk.

$$I_{11}=I_{12}=I_{13}=\dots=I_{19}=$$

$$I_{21}=I_{22}=I_{23}=\dots=I_{29}=$$

$$I_{91}=I_{92}=I_{93}=\dots=I_{99}=\{0, 1, 2 \dots\}$$

Az I-k azt jelölik, hogy az adott pozíción lévő kő, a saját színű (T_b vagy T_w) tömbjének hányadik indexű eleméhez tartozik.

Állapottér:

$$\mathcal{A} = \{h = (h_{11}, \dots, h_{99}, i_{11}, \dots, i_{99}, T_b, T_w, j) \in H_{11} \times H_{12} \times \dots \times H_{98} \times H_{99} \times J\}$$

Kezdőállapot:

$$kezdő = (0, \dots, 0, -1, \dots, -1, T_b, T_w, 1) \in \mathcal{A}$$

A két tömb kezdetben üres.

Végállapotok: $\mathcal{C}$

$$\begin{aligned} \mathcal{C} = \{ & (h_{11}, \dots, h_{99}, i_{11}, \dots, i_{99}, T_b, T_w, j) \in \mathcal{A} \mid \\ & (\exists i(\text{hossz}(T_b[i]) = 0), \text{ és } j \text{ veszít, ha } j = 1) \vee \\ & (\exists i(\text{hossz}(T_w[i]) = 0), \text{ és } j \text{ veszít, ha } j = 3)\} \end{aligned}$$

$\text{hossz}(T_b[i])$ megadja a T_b tömb i . pozícióján lévő halmaz elemeinek a számát.

Operátor:

$$\mathcal{O} = \{\text{Lép}(\text{mit}, \text{hova})\}$$

Egy $\text{Lép}(\text{mit}, \text{hova})$ operátor alkalmazható a $h = (h_{11}, \dots, h_{99}, i_{11}, \dots, i_{99}, T_b, T_w, j) \in \mathcal{A}$ állapotra, ha teljesülnek a következő operator alkalmazási előfeltételek:

1. Csak oda tehetek követ, ahol üres hely van.

$$h_{\text{hova}} = 0$$

2. Csak olyan színű követ tehetünk, amelyikkel mi játszunk.

$$\text{mit} = j$$

$$\text{előfeltétel}_{\text{Lép}(\text{mit}, \text{hova})}(h) = (h_{\text{hova}} = 0) \wedge (\text{mit} = j)$$

Egy $Lép(mit, hova)$ operátort a $h = (h_{11}, \dots, h_{99}, i_{11}, \dots, i_{99}, T_b, T_w, j) \in \mathcal{A}$ állapotra alkalmazva a következőképpen definiált $h' = (h'_{11}, \dots, h'_{99}, i'_{11}, \dots, i'_{99}, T'_b, T'_w, j') \in \mathcal{A}$ állapotot kapjuk:

$$h_{kl}' = \begin{cases} 1, & \text{ha } kl = hova, \text{ és } mit = 1 \\ 3, & \text{ha } kl = hova, \text{ és } mit = 3 \\ 0, & \text{egyébként} \end{cases}$$

$$j' = \begin{cases} 1, & \text{ha } j = 3 \\ 3, & \text{ha } j = 1 \end{cases}$$

$$i_{kl}' = \begin{cases} tömböt_frissít(T_b, mit, hova), & \text{ha } kl = hova, \text{ és } mit = 1 \\ tömböt_frissít(T_w, mit, hova), & \text{ha } kl = hova, \text{ és } mit = 3 \end{cases}$$

A $tömböt_frissít(T_b, mit, hova)$ függvény az I mátrix segítségével eldönti, hogy a *hova* elhelyezett kő a T_b tömbben melyik sztringhez tartozik, és az életeit ezen sztring életeivel összeuniózza, illetve létrehoz egy új sztringet, ha nem kapcsolódik egyik már létezőhöz sem. A függvény a már létező, vagy az újonnan létrehozott sztring tömbbeli indexével tér vissza.

Az $\mathcal{A}$ állapottérrel, a *kezdő* kezdőállapottal, a $\mathcal{T}$ végállapotok halmazával, az operátorok alkalmazási előfeltételeivel és az $\mathcal{O}$ operátorok halmazával definiáltuk a problémánk egy lehetséges $\langle \mathcal{A}, kezdő, \mathcal{T}, \mathcal{O} \rangle$ állapottér-reprezentációját.

A reprezentáció konkrét megvalósítása

A kövek, és a hozzájuk tartozó sztringek sorszámainak tárolására két kétdimenziós tömböt használunk, a *korongok*, és a *sztringSorszam* tömböket. A sztringek életeinek tárolására pedig két *ArrayList*-et, mivel erre a célra dinamikus adatszerkezetet célszerű használni, hiszen nem tudjuk előre, hogy mennyi sztringünk lesz. A tömblisták elemei *Vector*-ok, szintén ezen okból. Először *Set*-et alkalmaztam, de kiderült, hogy ez nem megfelelő megoldás, így került a helyükre *Vector*, de erről a problémáról később még lesz szó. A *Vector* elemei *Pozicio* objektumok, melyek a *GoAllapot* osztályban található *Pozicio* osztály példányai. Egy *Pozicio*

objektum két int típusú adattaggal rendelkezik, ezek *sor*, és *oszlop*. Értelemszerűen ez a tábla egy rácspontját határozza meg sor és oszlop koordinátákkal.

Nézzünk egy kis példát, hogy jobban átlássuk az ábrázolásmódot!

korongok									
	1	2	3	4	5	6	7	8	9
1	0	1	0	0	0	0	0	0	0
2	3	1	1	0	0	0	0	0	0
3	3	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0
6	0	0	0	0	3	0	0	0	0
7	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0

sztringSorszam									
	1	2	3	4	5	6	7	8	9
1	-1	0	-1	-1	-1	-1	-1	-1	-1
2	1	0	0	-1	-1	-1	-1	-1	-1
3	1	-1	-1	-1	-1	-1	-1	-1	-1
4	-1	-1	-1	-1	-1	-1	-1	-1	-1
5	-1	-1	-1	-1	-1	-1	-1	-1	-1
6	-1	-1	-1	-1	0	-1	-1	-1	-1
7	-1	-1	-1	-1	-1	-1	-1	-1	-1
8	-1	-1	-1	-1	-1	-1	-1	-1	-1
9	-1	-1	-1	-1	-1	-1	-1	-1	-1

sztringekFehér

0	1
V ₁	V ₂

sztringekFekete

0
V ₃

V₁ : (5, 5); (6, 6); (7, 5); (6, 4)

V₂ : (1, 1); (3, 2); (4, 1)

V₃ : (1, 1); (1, 3); (2, 4); (3, 2); (3, 3)

A kétdimenziós tömbök mérete 11 x 11, tehát két sorral és két oszloppal több, mint a tábla. Ennek oka egyrészt, hogy mivel az indexelés 0-tól indul, így a sorokra és oszlopokra való hivatkozáskor mindig egyel eltolt értéket kellene alkalmazni, amit könnyű elhibázni. Ezért a 0-s indexet nem használjuk valós érték ábrázolására, vagyis az indexelést valójában 1-től kezdjük. A másik ok, és ez sokkal lényegesebb, mint az átszámolás megspórolása, hogy így nem kell külön kezelnünk a pálya szélét. Ezért van szükség a 10-es indexű sorra, és oszlopra, hisz az indexátszámolás elkerüléséhez elegendő csak a 0-s sort és oszlopot figyelmen kívül hagyni. A széleken lévő plusz egy sornak és oszlopnak jelentőségét a későbbiekben jobban látni fogjuk. Az állapottér reprezentációban láttuk, hogy a korongok tömb elemei kezdetben 0-k, majd 1, illetve 3, attól függően, hogy milyen színű kő került az adott pozícióra. A pálya széleit reprezentáló pozíciók értéke 2. Tehát a mátrix kezdetben a következőképp néz ki:

2	2	2	2	2	2	2	2	2	2	2
2	0	0	0	0	0	0	0	0	0	2
2	0	0	0	0	0	0	0	0	0	2
2	0	0	0	0	0	0	0	0	0	2
2	0	0	0	0	0	0	0	0	0	2
2	0	0	0	0	0	0	0	0	0	2
2	0	0	0	0	0	0	0	0	0	2
2	0	0	0	0	0	0	0	0	0	2
2	0	0	0	0	0	0	0	0	0	2
2	0	0	0	0	0	0	0	0	0	2
2	2	2	2	2	2	2	2	2	2	2

A két tömblista értelemszerűen üres az elején, hiszen egy sztring sincs még a pályán. Így néz ki egy kezdőállapot, melyet a paraméter nélküli konstruktor hoz létre. Az első lépés megtétele nem más, mint erre a kezdő állapotra egy operátor alkalmazása. Ezt az *alkalmaz* függvény végzi, mely paraméterül egy operátort kap meg, és ezt az operátort alkalmazza. A paraméter nélküli konstruktor mellett létezik egy másik konstruktor is, mely paraméterül egy állapotot kap. Ez az úgynevezett „copy konstruktor”, mely „lemásolja” a paraméterül kapott állapotot.

Az *alkalmaz* első lépésben létrehoz egy új állapotot, és lemásolja azt az állapotot, amelyhez tartozik, a *this* kulcsszó segítségével. Az operátor olyan információkat tartalmaz, melyekre szükségünk lesz az *alkalmaz* működése során. Ezért ahelyett, hogy mindig egy metódushívást hajtánánk végre, inkább változókbán tároljuk a sokszor használandó adatokat.

Először a *korongok* tömbben beállítjuk a *mit*-re a megfelelő pozíción levő értéket. Aztán megvizsgáljuk, hogy milyen színű követ kell elhelyezni a táblán. Ezt a *mit* mondja meg nekünk. A két szín elhelyezésének hasonló a mechanizmusa, egyetlen különbség, hogy az egyik esetben a *sztringekFekete*-t, a másik esetben a *sztringekFeher*-et használjuk meg.

Ezt követően megvizsgáljuk, hogy azon pozíció körül, ahová be kell helyeznünk a követ, mik vannak. Itt „körül” négszomszédságot jelent. Három eset lehetséges:

- Ha a szomszédjában üres helyet találunk, akkor ezt a szomszédos pozíciót egy *eletek* nevű tömbben letároljuk, hisz ez majd élete lesz a kövünknek.
- Ha a kövünkkel megegyező színű kő van a szomszédban, akkor kikeressük, hogy ez a saját színű szomszéd melyik indexű sztringhez tartozik, és ezt az indexet letároljuk egy *sztringIndexek* nevű tömbben. Erre azért lesz szükségünk, mert az új kövünk ehhez a sztringhez fog kapcsolódni, és az életei hozzáadódnak a sztring életeihez.
- Ha pedig ellenfél követ találjuk, akkor megkeressük, hogy ez az ellenséges kő melyik ellenfélbeli sztringhez tartozik, és azt a pozíciót, ahová most lépni akarunk, kivesszük ezen sztring életei közül. A lépésünk után ugyan is ez a hely már nem lesz üres.

Itt látszik igazán a jelentősége a reprezentációban megjelenő plusz soroknak, és oszlopoknak, hiszen ha a pálya szélére akarunk tenni követ, akkor biztos, hogy valamelyik szomszédunk, hiányzik, esetleg több is, ha sarokra teszünk. Ez esetben már maga a lekérés, hogy mi van a szomszédban, hibát okozna, hiszen a tömb egy nem létező indexére hivatkozna. Ezt el lehetne kerülni úgy, hogy mindig megvizsgálánánk, hogy van-e szomszédos pozíció, vagy sem, de ez plusz feltételvizsgálat, ami időbe kerül. A plusz sorok, és oszlopok letárolása pedig csak memóriát igényel. Ezeknek köszönhetően tehát itt van egy negyedik lehetőség, mikor a szomszédban ez a „fal” van, amit ugye 2-essel ábrázolunk, de ennek nincs semmi információtartalma, úgy érve, hogy semmilyen teendőnk nincs ebben az esetben. Ezért ezt a negyedik esetet el is hagytuk.

Mikor ezekkel végeztünk, tehát az összes szomszédot felderítettük, akkor le vannak tárolva a kövünk életei, valamint azon sztringek indexei, melyekhez kapcsolódni fog. Nyilván elő állhat olyan eset, mikor több sztringhez is kapcsolódik, ekkor ezeket a sztringeket úgymond „összevonja”, vagyis egy sztringgé köti össze őket.

Ezután egy ciklussal bejárjuk a *sztringIndexek* tömböt. Három eset lehetséges:

- A tömb üres. Ez azt jelenti, hogy nem találtunk egyetlen egy saját színű szomszédot sem. Ekkor a kövünk létrehoz egy új sztringet. Ezen új sztring életeit a kövünk életei alkotják, ezekkel az életekkel bekerül a saját színű sztringek életeit tartalmazó tömblista végére, az indexe pedig a *sztringSorszam* nevű kétdimenziós tömbbe.
- Ha nem üres, akkor meg kell vizsgálnunk, hogy ez az első olyan sztring-e, amihez majd kapcsolnunk kell a követ. Ha ez első, akkor a kövünk ehhez a sztringhez fog tartozni, ennek a sztringnek az indexét letároljuk, mert ha találunk még többet, akkor szükségünk lesz rá. A kő életeit hozzáadjuk a sztring életeihez, és a *sztringSorszam*-ot is módosítjuk. Egy élet csak egyszer szerepelhet. Ennek megvalósítását egy másik fejezetben tárgyaljuk.
- Ha nem ez az első ilyen sztring, akkor egyrészt meg kell bizonyosodnunk róla, hogy ez tényleg egy másik sztring, nem pedig ugyan azt találtuk meg még egyszer, mert ez is előfordulhat. Probléma ugyan nincs belőle, de fölösleges műveleteket eredményez. Ha két külön sztringről van szó, akkor az utóbb talált sztringet hozzáfűzzük az először megtalálthoz. Ezért kellett annak indexét letárolnunk. A hozzáfűzés annyit jelent, hogy a sztring összes életét hozzáadjuk a másik sztring életeihez. Ezt követően a tömblistából töröljük a hozzáfűzendő sztringet, a *sztringSorszam*-ot módosítjuk úgy, hogy a hozzáfűzendő sztring indexeit rendre átírjuk annak a sztringnek az indexére, amihez hozzáfűzzük, a tömblistából töröljük, valamint a törlés miatt, a törölt sztring utáni sztringek indexei is megváltoznak, ezért ezeket is módosítani kell a mátrixban, valamint a szomszédok felderítésekor használt *sztringIndexek* tömbben.

Mindezek után már csak annyi van hátra, hogy a játékost átállítsuk, és visszatérjünk az új állapottal.

Hasznosság

Elméletben

A hasznosság megadja, hogy egy állapot mennyire „jó” egy játékosnak. Ezt a játszma jelenlegi állása alapján kell valamilyen módon meghatároznunk, felhasználva az állapotban tárolt információkat. Minél nagyobb ez az érték, annál kedvezőbb az adott állapot. Arra, hogy mi alapján számoljuk ki a hasznosságot, kétféle módszer van.

- Az egyik, hogy mindig a soron következő játékos alapján számoljuk ki a hasznosságot, attól függően, hogy ki következik lépni. Ez lehetővé teszi, hogy a két játékosnak más-más módon számoljunk hasznosságértéket. Ez olyan típusú játékok esetében fontos, ahol a két játékos nem ugyanolyan végállapotra törekszik. A mi esetünkben ezt alkalmaztuk, hiszen a fekete játékos arra törekszik, hogy a fehér sztringek közül valamelyiknek elfogyjanak az életei, a fehér pedig arra, hogy a fekete sztringek közül legyen valamelyik körülzárva. Ezt nevezzük negamax hasznosságnak.
- A másik megoldás, hogy mindig a kezdő játékos alapján számoljuk ki a hasznosságot, és ha ő következik lépni, akkor ezt az értéket szolgáltatjuk, ha az ellenfele következik lépni, akkor pedig az ellentettjét. Ez a minimax hasznosság. Ezt tipikusan olyan esetekben lehet alkalmazni, mikor a két játékos „egyenértékű”, azaz egy állapot mindkét játékos számára egyformán jó, illetve rossz. Például, ha egy táblára egymás után helyeznek el egyenértékű bábúkat valamilyen szabály szerint, és az veszít, aki már nem tud lépni. Ebben az esetben mindkét játékos ugyan arra a végállapotra törekszik, nevezetesen, hogy mikor már csak egy szabad hely van, akkor ő következzen lépni. Ez esetben a hasznosságot csak az határozza meg, hogy van-e még olyan hely, ahová lépni lehet, de oda akár melyik játékos léphet. Tehát ami az egyik játékosnak jó állapot, az a másiknak rossz. A mi esetünk nem ilyen, hiszen mindkét játékos csak a saját színű köveit rakhatja le.

Gyakorlatban

Célunk, hogy az ellenfelet megverjük. Ezt úgy tehetjük meg, ha valamelyik sztringjét körbekerítjük négyyszomszédosan, vagyis ha elveszük az összes életét. Ez alapján azt mondhatjuk, hogy egy játékosnak egy állapot annál jobb, minél kisebb az ellenfele sztringjei életei számának a minimuma, valamint annál rosszabb, minél kisebb a saját, hasonló módon

kiszámolt minimumértéke. Tehát a soron következő játékosnak úgy számoljuk ki az állapot hasznosságát, hogy megnézzük az ellenfele sztringjeinek az életeit, amik ugye egy tömblistában lévő vektorok. Ezek közül a vektorok közül kiválasztjuk azt, amelyikben a legkevesebb elem – azaz élet – van. Ennek a vektornak az elemszáma minél kisebb, annál hasznosabb az állapot, tehát annál nagyobb értékkel térünk vissza. Jelen esetben ezt úgy valósítjuk meg, hogy ezt az elemszámot kivonjuk 50-ből, hisz biztos, hogy egy sztringnek kevesebb, mint 50 élete van. Valójában jóval kevesebb, de ez az 50 sem olyan nagy szám, így elég jól lehet vele számolni.

Ha a reprezentációnál azt a megoldást választjuk, hogy inkább a memóriával takarékoskodunk, és nem tárolunk plusz információkat az állapotban, akkor ez a hasznosság megállapítása elég sok műveletet igényelne. Hiszen ha csak a kövek táblán lévő elhelyezkedését ismernénk, akkor össze kellene gyűjtenünk minden sztring köveit, egyáltalán meghatározni, hogy mely kövek tartoznak egy sztringbe, annak melyek az életei, és ezek közül melyik rendelkezik a legkevesebbel. Ez a tábla többszöri bejárását eredményezné, és egy sor feltételvizsgálatot. Így azonban már ismerjük az egyes sztringek életeit, mivel egy kollekcióban gyűjtöttük, ezért nem is kell manuálisan bejárnunk, és összeszámolnunk, hiszen van rá függvény, ami a kollekció elemeinek számát szolgáltatja. Ezért aztán csak le kell kérnünk ezen értékeket, és kiválasztani a legkisebbet.

Ez a megoldás egy támadó jellegű stratégiát eredményez, de gondolnunk kell a védekezésre is. Hisz nem elég csak körbekeríteni az ellenfél egy sztringjét, azt is meg kell akadályoznunk, hogy a mi sztringjeink közül bármelyiket körbekerítsék.

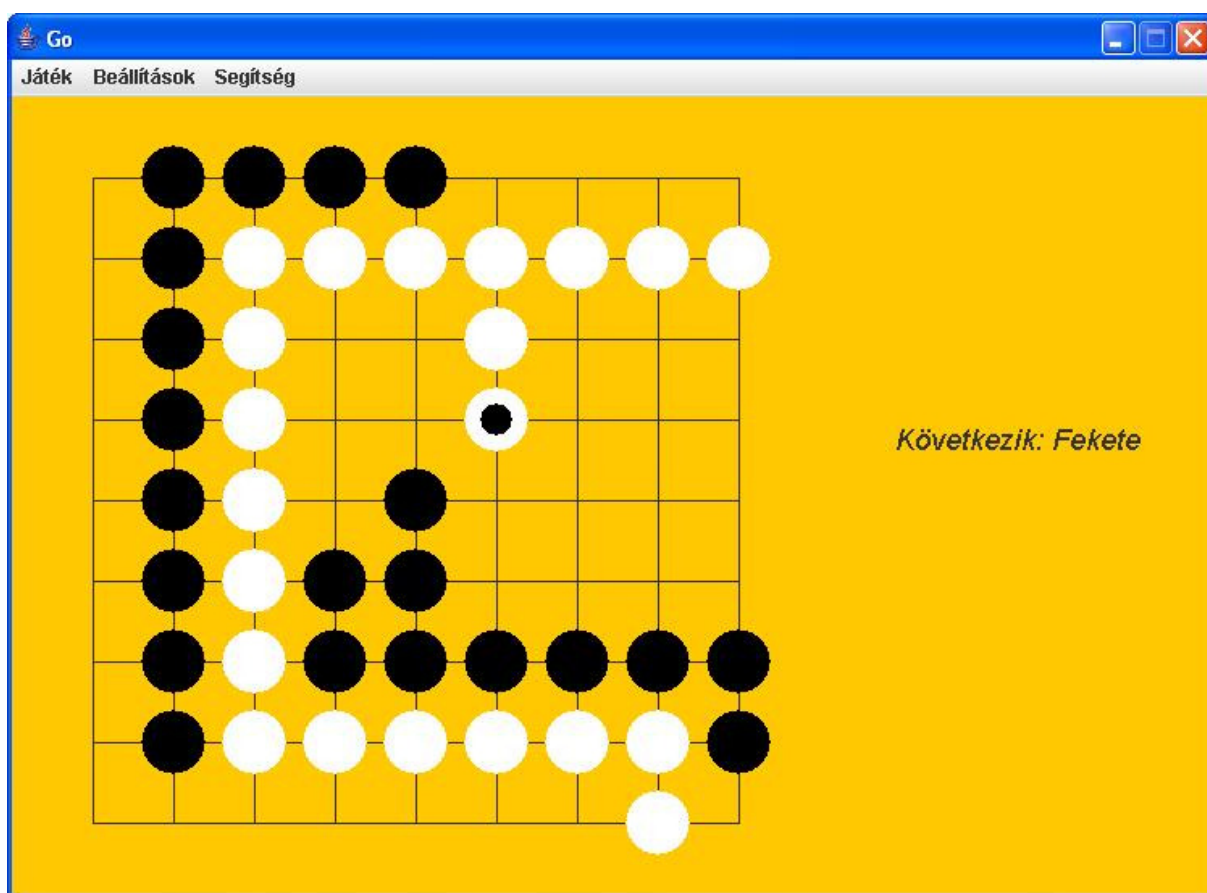
Ezért készült egy másik, védekezést megvalósító hasznosság, ami nem az ellenfél sztringjeinek életeit igyekeznénk minimalizálni, hanem a sajátjainkat maximalizálni. Érdekes módon, tesztelés során kiderült, hogy ez hatékonyabb, mint a támadó jellegű stratégia. Az optimális hasznosság valahol a kettő között található.

Ez a két szempont egyszerre történő figyelembevétele nem is olyan egyszerű feladat. Meg kell találni az arany középutat. Ez most nem sikerült. A hasznosságunk koránt sem a leghatékonyabb. A végleges verzióban a védekező stratégiát megvalósító hasznosság maradt benne. Ez elég hatékonynak tűnik, bár nem kell sok gyakorlás ahhoz, hogy meg tudjuk verni.

Kinézet

A játék egy ablakban jelenik meg. Elindulás után egy üres tábla jelenik meg, de nem indul el egy játszma automatikusan, amint azt már említettük. Játשמát a menü segítségével indíthatunk. Egy korábbi verzióban a program indulásakor csak egy üres, szürke ablak jelent meg, és szintén a menüből kellett indítani egy új játszmat, és csak ekkor jelent meg maga a tábla. Ebből a megoldásból azonban ablak kirajzolási gondok származtak, de erről majd részletesebben később.

Tehát a játék a következőkép néz ki:



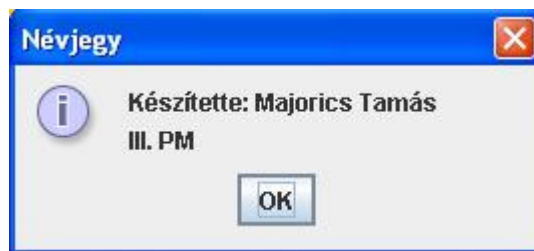
Baloldalon a táblát láthatjuk, jobb oldalra pedig a program az üzeneteit írja. Jelen esetben azt, hogy ki következik lépni, de ide írja például a játszma végén az eredményt is. Most épp egy játszma közepén tartunk. Ebben a pillanatban a Beállítások menü menüpontjainak nincs azonnali hatása, kivéve a nyelv beállítást, ha átállítunk egy beállítást, az csak egy új játszma indításakor lesz hatással a játszmánkra.

A menü

Amint az a képen is látszik, három menü van a menüsorban: *Játék*, *Beállítások* és *Segítség*.

A *Játék* menüben két menüpont található: *Új játék* és *Kilép*. Ezek nem igényelnek különösebb magyarázatot, funkciójuk világos.

A *Segítség* menüben csak egyetlen egy menüpont van, a *Névjegy*. Ennek hatására egy üzenőablakban megjelenik a készítő neve és szakja, vagyis az enyém, a nyelvi beállításnak megfelelően.

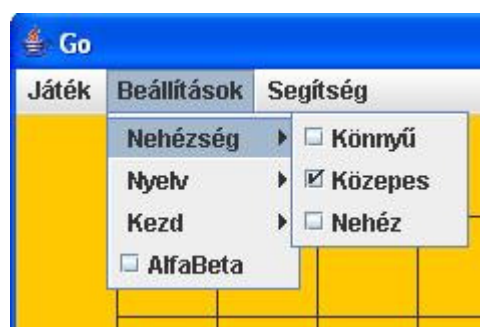


A *Beállítások* menüben már több dolog van, ezért is hagytam utoljára. Itt három almenü, valamint egy kiválasztó menüpont található, amit idegen szóval checkbox-nak szokás nevezni.



Az *AlfaBeta* checkbox bekapcsolásával az algoritmusunk alfa-bétavágást fog használni. Ennek hatása a gép gyorsaságában fog látszani.

A másik három almenü menüpontjai szintén checkbox-ok, melyek rádiógombként működnek, vagyis az egy almenübe tartozók közül csak egyetlen egy lehet kiválasztva.



A *Nehézség* háromszintű választási lehetőséget ad, ezek azt határozzák meg, hogy a lépésajánló algoritmus milyen mélységig működjön. Minél nagyobb a mélység, annál több lépést tud előre kiszámolni a gép, tehát annál nehezebb ellene játszani.



A *Nyelv* almenü is elég egyértelmű. Beállításával mind a menücímkek, mind a névjegy, mind a program által küldött üzenetek megváltoznak.



A *Kezd* pedig azt állítja be, hogy ki kezdje a játékot, a gép, vagy pedig az ember. A kövek szempontjából mindig a fekete kezd, ez a szabályokból adódik, csak az változik meg, hogy ki lép a feketével.

A fejlesztés során felmerült problémák

A fejlesztés során felmerült egy sor probléma, melyek egy részére megtaláltam a megoldást, másokat megkerültem. Ezeket lehet csoportosítani aszerint, hogy melyik témakörhöz tartoznak.

Programozás technikai problémák

Set

Amikor megfogalmazódott a reprezentáció, akkor ennek megfelelően ki kellett választani az alkalmazandó programozási eszközöket. Elsősorban az adattárolási struktúrát. Mind a sztringek tárolására, mind az egyes sztringek életeinek tárolására dinamikus adatszerkezetre volt szükség. A sztringek tárolására használt *ArrayList* meg is felelt a feladatnak, az egyes sztringek életeinek letárolására pedig *Set*-et használtam.

Ez nagyon jónak tűnt első ránézésre, hiszen nekünk a hasznosság megállapítása során nem kell elérnünk minden egyes életet külön, csak az életek darabszáma szükséges, valamint egy pozíciónak csak egyszer kell szerepelnie az életek között, hogy ne számoljunk egy életet kétszer. Ennek a két feltételnek a halmaz tökéletesen eleget tesz, mondhatni szinte erre találták ki.

De, és most jönnek a gondok, a Java-ban a *Set* szerintem, el van rontva. Ez kicsit erős kifejezés, de én erre a következtetésre jutottam. Ugyanis a *Set*, annak megállapítására, hogy egy adott elem szerepel-e már a halmazban, vagy sem, a benne szereplő elem *equals* metódusát használja. A *Set*-ben, mint minden kollekcióban, *Object* típusú objektumokat lehet elhelyezni. Gyakorlatilag minden objektum *Object* típusú a Java-ban, csak a dinamikus típusuk lehet más. Tehát a mi esetünkben *Pozicio* dinamikus típusú objektumokat tárolunk a halmazban. Ezt a *Pozicio* osztályt mi írtuk, és felüldefiniáltuk az *equals* metódust úgy, hogy akkor adjon vissza egyezést, ha mind a két koordináta megegyezik. Viszont a *Set* *add* metódusa, ami összehasonlításra az *equals*-t használja, nem a dinamikus típus szerinti *equals*-t hívja, hanem a statikus típus szerinti, ami nem más, mint az *Object*, tehát referencia alapján végzi az összehasonlítást. Ami érdekes, hogy a *Vector* például nem így tesz, hanem az a dinamikus típus szerinti *equals*-t használja. Ezért lett kicserélve a *Set*, *Vector*-ra.

A *Vector* azonban megengedi az elemduplikációt. Ezért ezekhez az életeket tartalmazó vektorokhoz történő hozzáadást külön függvények végzik, melyek először megvizsgálják,

hogyan az adott elem benne van-e már. Erre a *contains* metódust használják, mely szintén az *equals*-t hívja, de a „jó” *equals*-t. Azért mondtam, hogy függvények, mert kettő van. Az egyik egy elemet helyez be egy vektorba, a másik két vektort fűz össze.

Grafika

A grafikai résznél is felmerült egy olyan probléma, amit csak megkerülni sikerült. A program egy korábbi verziójában a játszmat vezérlő szál nem a *Jatek* osztályban volt, hanem a *GoMain*-ben. A *GoMain* indított egy szálát, amiben létrehozott egy *Jatek* objektumot. A kirajzolandó dolgokat egy *Container*-ben gyűjtöttük. Mikor egy új játszmat indítottunk, akkor az éppen aktív *Jatek* befejeződött, kitörlődött a *Container*-ből, létrehoztunk egy új *Jatek*-ot, aminek a konstruktorát a beállításoknak megfelelően hívtuk meg, így léptek érvényre a megváltoztatott beállítások, majd ezt az új *Jatek*-ot hozzáadtuk a *Container*-hez.

Ez majdnem tökéletesen működött is, de mikor egy új játszmat indítottunk, akkor kirajzolás frissítési gondok jelentkeztek. Ennek oka nem derült ki. A megoldást az jelentette, hogy a szál a *GoMain*-ből átkerült a *Jatek*-ba, és amikor egy új játszmat indítunk, akkor az eredeti *Jatek* objektum megmarad, csak alaphelyzetbe áll a beállításoknak megfelelően. Így a *Container*-be csak egyszer kerül be, és mindig ezt az egy objektumot kell kirajzolni.

Szemantikai hibák

Az egyik nagy szemantikai hiba a *GoAllapot* osztály alkalmaz metódusában jelentkezett. Az alkalmaz működését már ismertettük. Lényege, hogy alkalmaz egy operátort, és az operátor hatásának megfelelően karban tartja a letárolt adatokat. A hiba, mint az általában lenni szokott, egy szélsőséges esetben jelentkezett. Mégpedig akkor, mikor egy elhelyezett kő három, vagy esetleg négy sztringet összeköt. Ez igen ritka eset, de lehetséges. A gond a következőből adódott. Az algoritmus ugye azzal kezd, hogy megvizsgálja a szomszédokat, és ha ott saját színű követ talál, akkor az ahhoz tartozó sztring indexét letárolja egy tömbben. Ha legalább három különböző ilyen sztringünk van, akkor letárolunk legalább három indexet. Miután megvizsgáltuk a szomszédokat, következik a letárolt adatok feldolgozása, azaz bejárjuk ezt a tömböt. Megtaláljuk az első szomszédos sztringet. A követ hozzávesszük ehhez a sztringhez (Láttuk, hogy tulajdonképpen nem a követ vesszük hozzá, hanem annak életeit, de ez most lényegtelen). Majd ennek a sztringnek megjegyezzük az indexét, hisz minden

esetleges további szomszéd sztringet ehhez fogunk hozzáfűzni. Eddig semmi probléma nincsen.

Megyünk tovább, és megtaláljuk a második szomszéd sztringet. Ezzel már több teendőnk van. Mivel a kövünk már tartozik egy sztringhez, mégpedig ahhoz, amit elsőre megtaláltunk, ezért ehhez a sztringhez nem kell hozzávenni, sőt, a másodjára megtalált sztringünket kell hozzáfűzni az első sztringhez, a másodikat pedig törölni. A törlés abból áll, hogy a tömblistából kitöröljük, és az indexeket tartalmazó mátrixban a másodjára megtalált sztring indexeit rendre átírjuk az először megtalált sztring indexeire, amik pedig nagyobb indexűek voltak, mint a törölt sztringünk indexe, azok indexeit eggyel csökkentjük, hisz a tömblistában egy belső elem törlése úgy történik, hogy a rákövetkezőket eggyel balra toljuk. Ezzel sincs semmi probléma, működik rendesen.

A gond most következik, mikor a harmadjára megtalált sztringet dolgozzuk fel. Ugyanis megyünk tovább, és látjuk, hogy van még egy további szomszéd sztring, ezt abból látjuk, hogy már letároltuk az első szomszéd sztring indexét (Egyébként a másodjára megtalált sztringnél is így vesszük észre, hogy ez nem az első). Tehát ugyanazt tesszük, mint az előbb, az újonnan megtalált sztringet hozzáfűzzük a legelőször megtalálthoz, és mint az előbb, módosítjuk a mátrixot, és a tömblistát. És ez így nem jó! Pedig ugyanaz az eljárás, mint a második sztringnél, mégsem lesz jó.

A dolog nyitja a következő, és ez nem kevés álmatlan éjszakát okozott, mire sikerült rájönnöm, bár lehet, hogy nálam gyakorlottabb programozók már látják hol a hiba. A hiba ott van, hogy ha a szomszédban saját színű sztringek vannak, akkor ezek indexeit az elején, minden szomszédhoz letároljuk. Viszont ha két sztringet összevonunk, akkor bizonyos sztringeknek megváltozik az indexe. Ha pedig ezt követően még további sztring összevonást kell végeznünk, akkor lehetséges, hogy nem a jó indexszel fogunk dolgozni. Két megoldás van a problémára.

- Az egyik, hogy a szomszédokat nem egy menetben derítjük fel. Tehát először csak megnézzük, hogy melyek lesznek életek, aztán megnézzük, hogy van-e szomszédos saját színű sztring, és ha találunk, akkor csak az első indexét tároljuk le, és mikor találunk továbbiakat, azokat rögtön feldolgozzuk. Ebben az esetben a szomszédokat kétszer kell végignézni.
- A másik megoldás, hogy a szomszédokat egy menetben derítjük fel, és kibővítjük egy sztring törlését azzal, hogy az indexeket a szomszédos sztringek indexeinek

tárolására használt tömbben is aktualizáljuk. Én ezt a megoldást választottam, hogy ne kelljen kétszer felderíteni a szomszédokat.

Összefoglalás

Röviden összefoglaljuk, hogy milyen témákat érintettünk a dolgozatban. Melyek azok a dolgok, amikről bővebben esett szó, melyek azok, amiket csak érintőlegesen tárgyaltunk meg. Először is megtárgyaltuk, hogy mi is az a Go. Honnan ered, mik a játékszabályok, hogyan zajlik egy játszma, milyen stratégiákat alkalmaznak a profik, és úgy általánosságban, hogyan is néz ki a Go-világ. Ezt azért tartottam fontosnak, mert úgy érzem, hogy nálunk nem annyira ismert ez a játék. Például, az ismerőseim között, aki eddig megkérdezte tőlem, hogy milyen témában írom a szakdolgozatom, azok között egy olyan ember sem volt, aki hallott volna már erről a játékról. Az anyaggyűjtés során megtudtam ugyan, hogy egyre népszerűbbé válik hazánkban is, de én személy szerint ezt még nem tapasztaltam.

A következő fejezet a programot tárgyalta. Röviden megismerkedtünk a fejlesztői környezettel, illetve rálátást nyertünk azon tényezőkre, melyek befolyásolták a programozási nyelv kiválasztását. Ezt követően megnéztük a program osztálystruktúráját, az egyes osztályok felépítését, valamint funkcióját. Ez tulajdonképpen az alapja annak, hogy értelmezni tudjuk a kódot, hogy egyáltalán mondjon valamit számunkra annak olvasása, még ha elsőre nem is értjük azt teljesen. Majd szemantikailag vizsgáltuk a kódot. Ezáltal ismertük meg, hogy az egyes osztályok, metódusok, a funkciójukat hogyan is látják el. Megismerkedtünk az alkalmazott algoritmussal, melyet pszeudó kóddal adtunk meg. Ez azért jó, mert ha valaki nem ismeri az alkalmazott programnyelv szintaktikáját, akkor ebből a pszeudókódból mégis átlátja az algoritmus működési elvét.

Mivel a problémánk egy mesterséges intelligenciai probléma, ezért megadtuk az állapottér-reprezentációját. Az állapottér gráf megadásától most eltekintettünk, mivel meglehetősen sok operátorunk van, valamint az állapotunk struktúrája eléggé bonyolult. Ezért nemhogy a teljes, de egy gráf részlet megadása is túl sok időbe, és energiába került volna. Ráadásul a gráf egy csúcának bonyolultsága miatt, a gráf részlet átláthatatlan lenne, és semmilyen plusz információval nem szolgálna számunkra.

Ezen reprezentáció megvalósítása, programozástechnikailag volt érdekes. Hiszen a bonyolultsága miatt nagy hangsúlyt kellett fektetni az alkalmazott eszközök kiválasztására. Gondolok itt elsősorban azokra az adatszerkezetekre megvalósító kollekciókra, melyek segítségével egy állapot által hordozott információkat tároltuk, kezeltük. Itt nem tudtuk a leghatékonyabbnak tűnő eszközt alkalmazni. Ez a *Set* kollekció lett volna, helyette a *Vector*-t

alkalmaztuk. Ennek oka a fejlesztői környezet sajátosságában keresendő, amit én „Java hibaként” könyveltem el. Persze lehet, hogy tévedek, nem ismerem ugyanis ezen Java sajátosság konkrét okait. Elképzelhető, hogy ez a „hiba” mélyebben található tényezőkre vezethető vissza.

Ismertettük néhány bonyolultabb metódus pontos működését. Ezt azért tartottam lényegesnek, mert a fejlesztés során, mikor pár hétig nem foglalkoztam egy programrésszel, mert egy teljesen más részt fejlesztettem, akkor bizony nekem is idő kellett ahhoz, hogy újra rálátást nyerjek a félbehagyott rész működési elvére, holott én magam írtam azt a részt is. Ezért gondoltam, hogy egy „kívülálló”, magából a kódból csak nehezen látná át annak működési sajátosságait.

Szó esett a hasznosságról, mind elméleti szinten, mind pedig a mi konkrét esetünkben. Láthattuk, hogy a védekező stratégia a célra vezetőbb, nem pedig a támadó. Igaz, hogy a leghatékonyabb a kettő ötvözete lenne.

Ezt követően egy kicsivel könnyedebb téma következett. Megnéztük a programunk kinézetét, tehát amit egy átlag felhasználó lát a programból. Ismertettük a menüpontokat, és azok funkcióját, mindezeket képekkel illusztrálva.

Mint minden projectben, felléptek problémák a fejlesztés során. Végezetül ezek közül tekintettünk át néhány fontosabbat, illetve érdekesebbet. Itt kapott helyet a már említett „Java hiba” részletes ismertetése is. Bár ismétlem, nem vagyok olyan mélységig jártas a Java-ban, hogy meg tudnám mondani ennek konkrét okait. Lehet, hogy ez nem véletlenül van így megoldva a Java-ban.

A függelékben helyet kapott néhány fontosabb kód is, ha valaki meg szeretné nézni azok pontos szemantikáját.

Remélem sikerült jó áttekintést adnom mind a Go-ról, mint játékról, mind pedig a programról. Ha valaki e dolgok olvasása után dönt úgy, hogy őt érdekli a Go, és szeretné komolyabban megismerni, annak én szívből örülök.

Irodalomjegyzék

Írott anyagok

Várterész Magda: Mesterséges intelligencia 1 előadások

Elektronikus anyagok

<http://hu.wikipedia.org/wiki/Go>

<http://mge.eqnet.hu/gorol/japan/japan.php>

<http://gowiki.hu>

Függelék

Kódok

A *Jatek run* metódusa

Ez vezérli a játszmát.

```
public void run() {
    GoAllapot allapot = kezdoAllapot;
    LepesAjanlo lepesAjanlo;
    while(!allapot.vegAllapot()) {
        GoLepes op;
        if(!emberEllen && (allapot.getJatekos() == 1 && gepKezd ||
            allapot.getJatekos() == 3 && !gepKezd)) {
            if(!negamax && !alfabeta) {
                lepesAjanlo = new MinimaxAlgoritmus(allapot, gepiMelyseg);
            }
            else if(negamax && !alfabeta)
                lepesAjanlo = new NegamaxAlgoritmus(allapot, gepiMelyseg);
            else if(!negamax && alfabeta)
                lepesAjanlo = new AlfaBetaMinimax(allapot, gepiMelyseg);
            else
                lepesAjanlo = new AlfaBetaNegamax(allapot, gepiMelyseg);
            op = lepesAjanlo.operator;
        }
        else {
            if(!nincsAjanlat) {
                if(!negamax && !alfabeta) {
                    lepesAjanlo = new MinimaxAlgoritmus(allapot, emberiMelyseg);
                }
                else if(negamax && !alfabeta)
                    lepesAjanlo = new NegamaxAlgoritmus(allapot, emberiMelyseg);
                else if(!negamax && alfabeta)
                    lepesAjanlo = new AlfaBetaMinimax(allapot, emberiMelyseg);
                else
                    lepesAjanlo = new AlfaBetaNegamax(allapot, emberiMelyseg);
                op = lepesAjanlo.operator;
            }
            repaint();
            synchronized (mutex) {
                try {
                    mutex.wait();
                } catch (InterruptedException e) {
                    fut = false;
                    return;
                }
            }
        }
        while (true) {
            if (veg) {
                veg = false;
                fut = false;
                return;
            }
            if(voltLepes) {
```

```

        voltLepes = false;
        op = new GoLepes(allapot.getJatekos(), sor, oszlop);
        if (allapot.elofeltetel(op)) {
            break;
        }
        else {
            synchronized (mutex) {
                try {
                    mutex.wait();
                } catch (InterruptedException e) {
                    fut = false;
                    return;
                }
            }
        }
    }
    }
    }
    }
    allapot = allapot.alkalmaz(op);
    aktAllapot = allapot;
    JTA.setText(next);
    if(aktAllapot.getJatekos() == 1)
        JTA.append(black);
    else
        JTA.append(white);
    repaint();
}
JTA.setText(end + '\n');
if (allapot.nyertB() && allapot.nyertB()) {
    if(allapot.getJatekos() == 1) {
        JTA.append(winWhite);
    }
    else {
        JTA.append(winBlack);
    }
}
else if(allapot.nyertA()) {
    JTA.append(winBlack);
}
else if(allapot.nyertB()) {
    JTA.append(winWhite);
}
else {
    JTA.setText(egal);
}
fut = false;
}

```

A GoAllapot alkalmaz metódusa

Alkalmaz egy operátort az állapotra, és szolgáltat egy új állapotot.

```
public GoAllapot alkalmaz(GoLepes op) {
    int sor, oszlop, mit;
    Vector sztring;
    Pozicio[] eletek = new Pozicio[4];
    int[] sztringIndexek = {-1, -1, -1, -1};
    int elsoSzomszedIndexe = -1;
    GoAllapot uj = new GoAllapot(this);
    if(op instanceof GoLepes) {
        GoLepes gl = (GoLepes)op;
        sor = gl.getSor();
        oszlop = gl.getOszlop();
        mit = gl.getMit();
        uj.korongok[sor][oszlop] = mit;
        if(mit == 1) { //fekete
            switch (korongok[sor - 1][oszlop]) { //fölrötte
                case 0: { //üres
                    eletek[0] = new Pozicio(sor - 1, oszlop);
                    break;
                }
                case 1: { //saját szín
                    sztringIndexek[0] = uj.sztringSorszam[sor - 1][oszlop];
                    break;
                }
                case 3: { //ellenfél
                    sztring = new Vector((Vector)uj.sztringekFeher.get(
                        uj.sztringSorszam[sor - 1][oszlop]));
                    sztring.remove(new Pozicio(sor, oszlop));
                    uj.sztringekFeher.set(uj.sztringSorszam[sor - 1][oszlop], sztring);
                    break;
                }
            }
        }
        switch (korongok[sor][oszlop + 1]) { //jobbra
            case 0: { //üres
                eletek[1] = new Pozicio(sor, oszlop + 1);
                break;
            }
            case 1: { //saját szín
                sztringIndexek[1] = uj.sztringSorszam[sor][oszlop + 1];
                break;
            }
            case 3: { //ellenfél
                sztring = new Vector ((Vector)uj.sztringekFeher.get(
                    uj.sztringSorszam[sor][oszlop + 1]));
                sztring.remove(new Pozicio(sor, oszlop));
                uj.sztringekFeher.set(uj.sztringSorszam[sor][oszlop + 1], sztring);
                break;
            }
        }
    }
    switch (korongok[sor + 1][oszlop]) { //alatta
        case 0: { //üres
            eletek[2] = new Pozicio(sor + 1, oszlop);
            break;
        }
    }
}
```

```

    }
    case 1: { //saját szín
        sztringIndexek[2] = uj.sztringSorszam[sor + 1][oszlop];
        break;
    }
    case 3: { //ellenfél
        sztring = new Vector((Vector)uj.sztringekFeher.get(
            uj.sztringSorszam[sor + 1][oszlop]));
        sztring.remove(new Pozicio(sor, oszlop));
        uj.sztringekFeher.set(uj.sztringSorszam[sor + 1][oszlop], sztring);
        break;
    }
}
switch (korongok[sor][oszlop - 1]) { //balra
    case 0: { //üres
        eletek[3] = new Pozicio(sor, oszlop - 1);
        break;
    }
    case 1: { //saját szín
        sztringIndexek[3] = uj.sztringSorszam[sor][oszlop - 1];
        break;
    }
    case 3: { //ellenfél
        sztring = new Vector((Vector)uj.sztringekFeher.get(
            uj.sztringSorszam[sor][oszlop - 1]));
        sztring.remove(new Pozicio(sor, oszlop));
        uj.sztringekFeher.set(uj.sztringSorszam[sor][oszlop - 1], sztring);
        break;
    }
}
}
for(int i = 0; i <= 3; i++) {
    /* Volt saját színű szomszédja, és ez az első!
    */
    if(sztringIndexek[i] != -1 && elsoSzomszedIndexe == -1) {
        elsoSzomszedIndexe = sztringIndexek[i];
        sztring = new Vector((Vector)uj.sztringekFekete.get(sztringIndexek[i]));
        for(int j = 0; j <= 3; j++) {
            if(eletek[j] != null) {
                beszur(sztring, eletek[j]);
            }
        }
        sztring.remove(new Pozicio(sor, oszlop));
        uj.sztringekFekete.set(sztringIndexek[i], sztring);
        uj.sztringSorszam[sor][oszlop] = sztringIndexek[i];
    }
    /* Volt saját színű szomszédja, és ez nem az első!
    * Tehát sztringösszevonás lesz!
    */
    else if(sztringIndexek[i] != -1 && elsoSzomszedIndexe != -1) {
        if(elsoSzomszedIndexe != sztringIndexek[i]) {
            sztring = new Vector((Vector)uj.sztringekFekete.get(
                elsoSzomszedIndexe));
            beszurMind(sztring, (Vector)uj.sztringekFekete.get(sztringIndexek[i]));
            sztring.remove(new Pozicio(sor, oszlop));
            uj.sztringekFekete.set(elsoSzomszedIndexe, sztring);
        }
        /* A törlés miatt módosítani kell a sztringek indexeit
        */
    }
}

```

```

        int kul = 0; // A módosított indexet kell-e csökkenteni
        if(sztringIndexek[i] < elsoSzomszedIndexe)
            kul = 1;
        for(int k = 1; k <= 9; k++) {
            for(int j = 1; j <= 9; j++) {
                if(uj.sztringSorszam[k][j] == sztringIndexek[i] &&
                    uj.korongok[k][j] == 1)
                    uj.sztringSorszam[k][j] = elsoSzomszedIndexe - kul;
                else if (uj.sztringSorszam[k][j] > sztringIndexek[i] &&
                    uj.korongok[k][j] == 1) {
                    uj.sztringSorszam[k][j] = uj.sztringSorszam[k][j] - 1;
                }
            }
        }
        /* A sztringIndexek tömböt karban kell tartani!
        * A benne lévő indexeket aktualizálni kell,
        * a változásoknak megfelelően!
        */
        for(int n = i + 1; n <= 3; n++) {
            if(sztringIndexek[n] > sztringIndexek[i])
                sztringIndexek[n] = sztringIndexek[n] - 1;
        }
        /* Az első szomszéd indexét is karban kell tartani!
        */
        if(elsoSzomszedIndexe > sztringIndexek[i])
            elsoSzomszedIndexe = elsoSzomszedIndexe - 1;

        uj.sztringekFekete.remove(sztringIndexek[i]);
    }
}

/* Ez az az eset, mikor új 1 elemű sztring jön létre!
* Mikor nem találtunk szaját színű szomszédot!
*/
if(elsoSzomszedIndexe == -1) {
    sztring = new Vector();
    for(int j = 0; j <= 3; j++) {
        if(eletek[j] != null) {
            beszur(sztring, eletek[j]);
        }
    }
    uj.sztringekFekete.add(sztring);
    uj.sztringSorszam[sor][oszlop] = uj.sztringekFekete.indexOf(sztring);
}

else { //feher
    switch (korongok[sor - 1][oszlop]) { //föölötte
        case 0: { //üres
            eletek[0] = new Pozicio(sor - 1, oszlop);
            break;
        }
        case 3: { //saját szín
            sztringIndexek[0] = uj.sztringSorszam[sor - 1][oszlop];
            break;
        }
        case 1: { //ellenfél
            sztring = new Vector((Vector)uj.sztringekFekete.get(
                uj.sztringSorszam[sor - 1][oszlop]));

```

```

        sztring.remove(new Pozicio(sor, oszlop));
        uj.sztringekFekete.set(uj.sztringSorszam[sor - 1][oszlop], sztring);
        break;
    }
}
switch (korongok[sor][oszlop + 1]) { //jobbra
    case 0: { //üres
        eletek[1] = new Pozicio(sor, oszlop + 1);
        break;
    }
    case 3: { //saját szín
        sztringIndexek[1] = uj.sztringSorszam[sor][oszlop + 1];
        break;
    }
    case 1: { //ellenfél
        sztring = new Vector((Vector)uj.sztringekFekete.get(
            uj.sztringSorszam[sor][oszlop + 1]));
        sztring.remove(new Pozicio(sor, oszlop));
        uj.sztringekFekete.set(uj.sztringSorszam[sor][oszlop + 1], sztring);
        break;
    }
}
}
switch (korongok[sor + 1][oszlop]) { //alatta
    case 0: { //üres
        eletek[2] = new Pozicio(sor + 1, oszlop);
        break;
    }
    case 3: { //saját szín
        sztringIndexek[2] = uj.sztringSorszam[sor + 1][oszlop];
        break;
    }
    case 1: { //ellenfél
        sztring = new Vector((Vector)uj.sztringekFekete.get(
            uj.sztringSorszam[sor + 1][oszlop]));
        sztring.remove(new Pozicio(sor, oszlop));
        uj.sztringekFekete.set(uj.sztringSorszam[sor + 1][oszlop], sztring);
        break;
    }
}
}
switch (korongok[sor][oszlop - 1]) { //balra
    case 0: { //üres
        eletek[3] = new Pozicio(sor, oszlop - 1);
        break;
    }
    case 3: { //saját szín
        sztringIndexek[3] = uj.sztringSorszam[sor][oszlop - 1];
        break;
    }
    case 1: { //ellenfél
        sztring = new Vector((Vector)uj.sztringekFekete.get(
            uj.sztringSorszam[sor][oszlop - 1]));
        sztring.remove(new Pozicio(sor, oszlop));
        uj.sztringekFekete.set(uj.sztringSorszam[sor][oszlop - 1], sztring);
        break;
    }
}
}
for(int i = 0; i <= 3; i++) {
    /* Volt saját színű szomszédja, és ez az első!

```

```

*/
if(sztringIndexek[i] != -1 && elsoSzomszedIndexe == -1) {
    elsoSzomszedIndexe = sztringIndexek[i];
    sztring = new Vector((Vector)uj.sztringekFeher.get(sztringIndexek[i]));
    for(int j = 0; j <= 3; j++) {
        if(eletek[j] != null) {
            beszur(sztring, eletek[j]);
        }
    }
    sztring.remove(new Pozicio(sor, oszlop));
    uj.sztringekFeher.set(sztringIndexek[i], sztring);
    uj.sztringSorszam[sor][oszlop] = sztringIndexek[i];
}
/* Volt saját színű szomszédja, és ez nem az első!
* Tehát sztringösszevonás lesz!
*/
else if(sztringIndexek[i] != -1 && elsoSzomszedIndexe != -1) {
    if(elsoSzomszedIndexe != sztringIndexek[i]) {
        sztring = new Vector((Vector)uj.sztringekFeher.get(
            elsoSzomszedIndexe));
        beszurMind(sztring, (Vector)uj.sztringekFeher.get(sztringIndexek[i]));
        sztring.remove(new Pozicio(sor, oszlop));
        uj.sztringekFeher.set(elsoSzomszedIndexe, sztring);

        /* A törlés miatt módosítani kell a sztringek indexeit
        */
        int kul = 0; // A módosított indexet kell-e csökkenteni
        if(sztringIndexek[i] < elsoSzomszedIndexe)
            kul = 1;
        for(int k = 1; k <= 9; k++) {
            for(int j = 1; j <= 9; j++) {
                if(uj.sztringSorszam[k][j] == sztringIndexek[i] &&
                    uj.korongok[k][j] == 3)
                    uj.sztringSorszam[k][j] = elsoSzomszedIndexe - kul;
                else if (uj.sztringSorszam[k][j] > sztringIndexek[i] &&
                    uj.korongok[k][j] == 3) {
                    uj.sztringSorszam[k][j] = uj.sztringSorszam[k][j] - 1;
                }
            }
        }
        /* A sztringIndexek tömböt karban kell tartani!
        * A benne lévő indexeket aktualizálni kell,
        * a változásoknak megfelelően!
        */
        for(int n = i + 1; n <= 3; n++) {
            if(sztringIndexek[n] > sztringIndexek[i])
                sztringIndexek[n] = sztringIndexek[n] - 1;
        }
        if(elsoSzomszedIndexe > sztringIndexek[i])
            elsoSzomszedIndexe = elsoSzomszedIndexe - 1;
        uj.sztringekFeher.remove(sztringIndexek[i]);
    }
}
}
if(elsoSzomszedIndexe == -1) {
    sztring = new Vector();
    for(int j = 0; j <= 3; j++) {
        if(eletek[j] != null) {

```

```

        beszur(sztring, eletek[j]);
    }
}
uj.sztringekFeher.add(sztring);
uj.sztringSorszam[sor][oszlop] = uj.sztringekFeher.indexOf(sztring);
}
}
}
uj.valtJatekos();
return uj;
}

```

A MinimaxAlgoritmus konstruktora

Ez tartalmazza a minimax algoritmust.

```

public MinimaxAlgoritmus(GoAllapot allap, int melys) {
    this.allapot = new GoAllapot(allap);
    this.melyseg = melys;
    allasokSzama = 1;
    if(allapot.vegAllapot() || melyseg == 0) {
        hasznossag = allapot.minimaxHasznossag();
    }
    else if(allapot.getJatekos() == 1) {
        hasznossag = MINUSZ_VEGTELEN;
        Iterator iter = GoAllapot.getOperatorok().iterator();
        for(;iter.hasNext();){
            GoLepes op = (GoLepes)iter.next();
            if(allapot.elofeltetel(op)) {
                GoAllapot uj = allapot.alkalmaz(op);
                MinimaxAlgoritmus minimax = new MinimaxAlgoritmus(uj, melyseg-1);
                if(minimax.hasznossag > hasznossag) {
                    hasznossag = minimax.hasznossag;
                    operator = op;
                }
                allasokSzama += minimax.allasokSzama;
            }
        }
    }
    else {
        hasznossag = PLUSZ_VEGTELEN;
        Iterator iter = GoAllapot.getOperatorok().iterator();
        for(;iter.hasNext();){
            GoLepes op = (GoLepes)iter.next();
            if(allapot.elofeltetel(op)) {
                GoAllapot uj = allapot.alkalmaz(op);
                MinimaxAlgoritmus minimax = new MinimaxAlgoritmus(uj, melyseg-1);
                if(minimax.hasznossag < hasznossag) {
                    hasznossag = minimax.hasznossag;
                    operator = op;
                }
                allasokSzama += minimax.allasokSzama;
            }
        }
    }
}

```