Debreceni Egyetem

Informatika Kar

A hagyományos sakk és egyéb variánsai

Témavezető:

Dr. habil. Nagy Benedek

Készítette:

Szántó Gergő és Tóth Sándor

PTI

Debrecen

2010

Tartalomjegyzék

Tartalomjegyzék

Tartalomjegyzék

1. Bevezetés

A szakdolgozatunk témája a hagyományos sakk és annak variánsai. Mivel mindketten szeretünk sakkozni, ezért nem is lehetett kétséges, hogy sakkprogramot fogunk szakdolgozatként elkészíteni. Eleinte más ötleteink is voltak, de mindvégig a táblajátékok terén gondolkoztunk. A témavezető tanárunk ajánlotta fel, ha már a két személyes játékokat preferáljuk, akkor írjunk sakkprogramot. Mivel egyikünktől sem ált távol a mesterséges intelligencia, így belevágtunk. Amiatt, hogy ketten fogtunk neki a nem is kicsi projektnek megtapasztalhattuk, milyen előnyökkel és hátrányokkal fogunk szembenézni projektmunka tekintetében a későbbi munkahelyeinken. Mindenképpen sok időbe került a munkánk összehangolása, illetve a módszereink tekintetében is akadt eltérés. De az összehangolt munka végül meghozta a gyümölcsét, és kezünkben foghatjuk életünk első nagyobb projektjét, ami a szakon programozott feladatoktól eltérően lényegesen nagyobb, és több munkát igényelt az elkészítése. Maga a projekt megtervezése nem okozott sok gondot számunkra, mivel a témavezetőnk felosztotta helyettünk a feladatokat, így a konfliktusokat elkerülve hatékonyabban tudtunk dolgozni a saját feladatainkon. Természetesen voltak félreértések, de alapjában véve egymásra voltunk utalva, így azon túl, hogy mindenki végezte a saját dolgát, tudott segíteni a társának is. Abban mindketten egyetértünk, hogy ezzel rengeteg időt spóroltunk meg. A programunk Java nyelven készült, a forráskódok értelmezéséhez a nyelv ismerete szükséges.

A célunk az volt, hogy elkészítsünk egy hagyományos sakkjátékot annak minden szabályával együtt, illetve emellett legyen lehetőség a sakk határán mozgó variánsokat is elkészítenünk, lehetőleg minél általánosabb formában, hogy az esetleges későbbi bővítések ne okozzanak gondot. A variánsok tekintetében próbáltunk minél több irányba kitérni, illetve minél többet megvalósítani, de a megvalósított variánsok száma mellékes volt amellett, hogy az összes elkészített variánsunk általános jellegű legyen.

A sakkprogram elkészítése folyamán lehetőségünk nyílt gyakorlati szinten betekinteni a mesterséges intelligencia ezen ágába is. Az elméleti részekkel mindketten tisztában voltunk a Mesterséges Intelligencia c. kötelező szakmai tárgyunk miatt, de a félév folyamán kétszemélyes játékok leprogramozására sajnos nem volt idő. Így vettük a bátorságot és nekiláttunk egy olyan projektnek, amelyben félig-meddig kezdők voltunk, és a két szakdolgozással eltöltött félév folyamán váltunk a kezdőből kevésbé kezdővé. Sajnos az

időbeli lehetőségekbe csak az alapprogram és néhány variánsa fért be, de végzősöktől talán még ez is szép teljesítmény.

A vizsgálati módszerek szintjén megpróbáltunk minél alaposabbak lenni, és mindenre kitérni, bár ez nem volt egyszerű egy ilyen bonyolult és elterjedt program tekintetében. Hasznosságkiértékelő függvényből például többet is készítettünk, mire a végleges verzió bekerült a programba. Továbbá nagyon sok időt igénybevett az elkészített programrészek átvizsgálása, tesztelése, illetve a többi komponenssel való összeegyeztetése.

Fontos célkitűzésünk volt továbbá, hogy elkészítsünk egy olyan programot, amely a variánsai miatt élvezetes, és mindig új kihívásokat rejt magában, de mindemellett ne csak a számítógép ellen lehessen játszani, hanem nyújtson lehetőséget két barátnak egymással szemben leülni és egymás ellen - a mindennapok zajos forgatagából kiszakadva – játszani. Ahogy azt mi is tettük korábban.

2. Sakktörténelem

2.1. Elődök

[1]A sakk a közhiedelem szerint Észak-Nyugat Indiából származik, a Gupta Birodalom idejéből. A 6. században a sakk korai formáját caturanga-nak hívták, ami a szanszkrit nyelvben a hadsereg négy osztagát (gyalogság, lovasság, elefántok, harci szekerek) reprezentálták, amik a mai sakk bábuinak is megfelelnek (gyalog, huszár, futó, bástya).

A sakk meglétére utaló legkorábbi jeleket a szomszédos Sassand Perzsiában 600 körül fedezték fel. A játékot Chatrang-nak hívták. Perzsia iszlám meghódítása után a muszlim világ átvette a Chatrangot, átnevezte, és shatranj-nak hívták. A bábuk perzsa elnevezését megőrizték. A spanyolok ajedrez-nek, a portugálok xadrez-nek, a görögök zatrik-nak hívták a mai sakk elődjét. Európa többi részében viszont a perzsa shah nevet használták és ebből származik az angol check és chess.

Nyugat Európát és Oroszországot a 9. században legalább három féle úton érte el a sakk. 1000-re a sakk egész Európában elterjedt. Egy 13. századi kéziratból sok információt tudhattak meg ezzel kapcsolatban.

Egy másik elmélet szerint a sakk egy kínai játékból, a xianqi-ből ered, de ezt az elképzelést még nem bizonyították.

2.2 A modern játék gyökerei (1000-1850)

1200 körül a shatranj szabályai változáson mentek keresztül Dél-Európában, és 1475-ben számos, a játék alapvető képét megváltoztató változást vittek véghez, és így nyerte el azt a formát, amely a mai játék alapját adja. Olaszországban és Spanyolországban szintén alkalmazták a modern szabályokat. Ekkor alakult ki az a szabály, miszerint a gyalog kezdésből kettőt is léphet, míg a futók és a vezérek elérték mai képességeiket. A vezér (királynő) a 10. század vége felé a vizier-t felváltotta, és a 15. századra a legnagyobb erővel rendelkező bábuvá vált. Következésképp a modern sakkra csak a „Királynő sakkja”-ként vagy „Őrült királynői sakk”-ként utaltak. Az új szabályok gyorsan elterjedtek Nyugat-Európában, kivéve a patt szabályai, amelyek csak a korai 19. században nyerték el végleges formájukat.

Annak érdekében, hogy megkülönböztessük az elődjeitől, a szabályok ezen verziójára nyugati sakk-ként, nemzetközi sakk-ként utalnak.

A 15. században olyan írások kezdtek megjelenni, amelyek már a játék elméletéről szóltak. A legrégibb fennmaradt sakk könyv: Repetición de Amores y Arte de Ajedrez. Egy spanyol pap Luis Ramirez de Lucena írta, és Salamancában adták ki 1497-ben. Lucena és későbbi mesterek a nyitás elemeit fejlesztették, és az egyszerű végjátékot elemezték.

A 18. században az Európai sakk élet központja Dél-Európából Franciaországba helyeződött át. A két legfontosabb sakkmester: Francois-André Danican Philidor, egy zenész, aki felismerte a gyalogok fontosságát a sakk stratégájának szempontjából, és később Louis-Charles Mahé de La Bourdonnais, aki számos játszmát megnyert a híres ír mester Alexander McDonnell ellen 1834-ben. Ebben az időszakban a sakk élet központjai a nagy európai városok kávézói voltak.

A 19. században a sakk fejlődésnek indult. Sok klub alakult, könyv és újság jelent meg a témában. Városok közötti versenyeket rendeztek, és a sakk problémák az újságok szerves részévé váltak. 1843-ban von der Lasa publikálta az első átfogó könyvet, ami a sakk elméletével foglalkozott, a címe Handbuch des Schachspiels volt.

2.3 Egy sport születése (1850-1945)

Az első modern sakk torna Londonban került megrendezésre 1851-ben, egy német, akkor még ismeretlen Adolf Anderssen nyerte meg. Adnerssen vezető sakkmesterré vált, és a korra jellemző támadó stílusban játszott, habár később stratégiailag gyengének titulálták játékát.

Mélyebb betekintést a sakk természetébe két fiatal játékos által nyerhettünk. Az egyikük, az amerikai Paul Morphy, az összes fontosabb játékos ellen nyert, még Anderssen ellen is. Jó intuitív érzékkel rendelkezett a támadások előkészítéséhez. A prágai Wilhelm Steinitz leírta, hogy kerülhetőek ki a pozícióból fakadó gyengeségek, és hogy használhatók ki az ellenfél pozíciójából adódó gyengeségek. Steinitz nevéhez ezenkívül még egy fontos hagyomány tapad, Johannes Zukertort elleni meccseit 1886-ban az első sakk világbajnokságnak tekintjük. Steinitz trónját 1894-ben vesztette el.

José Raul Capablanca véget vetett a németek domináns szerepének, és egyszerű pozícióival, végjátékával 8 éven keresztül verhetetlenné vált, egészen 1924-ig.

A két világháború között a sakk forradalmi változásokon ment keresztül, az úgynevezett hypermodernisták idejének nevezhető ez az időszak. A gyalogok helyett nagyobb hatótávolságú bábuknak szánták a center pozíciók kontrollálását, ezzel az ellenfél gyalogjait becsalogatják, amelyek így támadási felületté válnak.

A 19. század befejeztével, az évente tartott mester tornák száma meggyarapodott. Néhány forrás szerint 1914-ben a sakk nagymesteri címet II. Miklós orosz cár adományozta, de ezt a formáját a címszerzésnek nagyon sokan vitatták. Hasonló címek átadásának hagyományát a Sakk Világ Szövetség (FIDE) folytatta.

2.4 A háború utáni időszak (1945-)

A világbajnokok a FIDE által felügyelt tornákon résztvevő játékosok közül kerültek ki. Egészen a Szovjet Birodalom bukásáig egyetlen nem szovjet játékos férkőzött be az elsők közé, Bobby Fischer 1972-1975-ig volt világbajnok. Rajta kívül csak szovjet versenyzők dominálták a mezőnyt.

A korábbi nem hivatalos rendszerben a világbajnoknak jogában állt dönteni, hogy mely vállalkozó szellemű játékost választja, a kiválasztottnak pedig saját magának kellett biztosítania a szponzorokat a meccsre. A FIDE egy új rendszert dolgozott ki a felvezető tornák és a világbajnoki döntő lebonyolítására. A felvezető tornák győztese hívhatta ki az aktuális uralkodó világbajnokot. A legyőzött bajnoknak lehetősége volt visszavágóra egy évvel később. Ez a rendszer egy 3 éves időtartam alatt bonyolódott le. 1961 után a FIDE eltörölte a visszavágó lehetőségét.

2.5 A számítógép és a sakk

Az utóbbi évtizedek legnagyobb változását a számítástechnikának köszönhetjük. A sakk és a számítógép közös történetének kezdete már az 1950-es évekre tehető, de csak New York-ban 1997-ben fordult élesre a gép-ember párharc. Ezt megelőzően a gépek esélytelenek voltak az emberrel szemben, ez alkalommal viszont az IBM gépe, a Deep Blue le tudta győzni a világ akkori legjobbját Garri Kaszparovot. Akkor a Deep Blue másodpercenként 200 millió állást tudott értékelni, Kaszparov hármat. A hatjátszmás mérkőzés egyébként szoros volt,

Kaszparov vereségét az utolsó partiban mutatott érthetetlenül gyenge játéka okozta, a végeredmény 3,5 - 2,5 lett.

A Kaszparovénál sokkal nagyobb vereséget szenvedett el 2005-ben az angol Michael Adams, aki 5,5 - 0,5 arányban kapott ki a hétköznapinak nem nevezhető erőforrású Hydrától, egyetlen adat: a Hydra 64 gigabyte RAM-ot használt. Ezt követően a sakk programok fejlődése a kisebb erőforrásigények felé haladt tovább. 2006-ban az aktuális világbajnok Vladimir Kramnik 4-2-re kapott ki a Deep Fritz névre hallgató géptől, ami már egy közönséges személyi számítógépen futott. A sakk gépek legújabb példánya a Pocket Fritz, ami már egy PDA-n is elfut.

3. Sakkvariánsok

3.1 Sakkvariánsok

[2][3]Sakkvariánsoknak azokat a táblajátékokat nevezzük, melyek valamilyen kapcsolatban vannak a sakkal. Lehet ez a kapcsolat akár szorosabb, a verziók csak néhány szabályban különböznek, például hogy a sáncolás megengedett-e, de lehetnek lazábbak is, ahol különbözik a tábla formája, a bábuk, a játék célja és a szabályok is variálhatóak.

A legfőbb különbségek a normál sakkhoz képest, ami alapján besorolhatóak a különböző sakkvariációk:

- Megváltoztatott tábla. Ez a tábla mérete is lehet akár nagyobb akár kisebb, de lehet a tábla formája is például a hexasakkban a tábla és a mezők formája is hatszögletű.
- Megváltozott kezdő állások. Ezek közül a legismertebb a sakk960, ahol a tisztek véletlenszerűen össze vannak keverve.
- Megváltozott szabályok, amik lehetnek megváltozott lépési szabályok, lehet más a játék célja, akár az ellenfél leütött bábuinak visszahelyezését is megengedhetik.
- Különbség lehet hogy hány táblán folyik a játék, vagy hogy hányan játsszák.

3.2 Néhány érdekes sakkvariáns

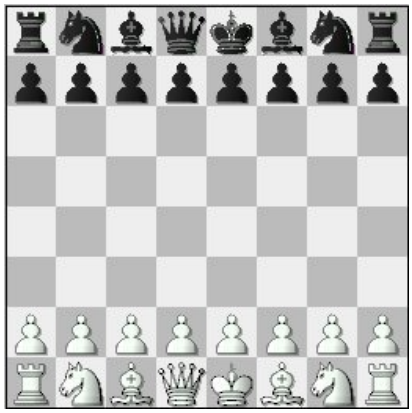
A sakkvariánsok népszerűsége időben és térben is változó, megpróbáltunk itt néhányat a teljesség igénye nélkül kiválasztani az Európában legnépszerűbb most is játszott variánsok közül kiválasztani, és még néhányat hozzá tettünk az általunk legérdekesebbnek tartottakból.

3.2.1 Normál sakk

Ahhoz hogy beszélhessünk a sakk különböző variánsairól először is tisztáznunk kell a sakk szabályait. A normál sakkot két fél játssza egy négyzet alakú táblán, ami 64 darab egyenlő nagyságú négyzetre van osztva. A mezők színei váltakoznak. A játékosoknak összesen 32 darab figurájuk van, a két játékosnak különböző színűek, az egyik világos a

másik sötét, ez általában fehér és fekete. A tábla sorait számokkal, az oszlopokat pedig betűkkel jelöljük, az első sorban vannak a világos tisztjei, a másodikban a világos gyalogjai, a hetedikben a sötét gyalogjai, a nyolcadikban a sötét tisztjei.

Kezdőállás:



Figurák:

-  Gyalog: Csak előre léphet, kezdőállásból akár kettőt is, de azután mindig csak egy mezőt, ütni viszont csakis átlósan tud, akkor is csak egy mezőt.
-  Huszár: Mindig „L" alakban lép, függőlegesen kettő és vízszintesen egyet, vagy fordítva. A huszár az egyedüli bábu, amely nem egy adott irányba, hanem ugrásban közlekedik, ezért nem is lehet arról beszélni, hogy valami az útjában lenne, ha teljesen körbe van véve, akkor is képes kilépni, például a kezdőállásból is.
-  Futó: Átlósan tud lépni bármennyit egészen addig, amíg a tábla széle, vagy egy másik bábu az útjába nem áll. Egy üres táblán a c1-en álló futó léphet akár a3-ra és h6-ra is
-  Bástya: Lépése nagyon hasonlít a futóéra, azzal a különbséggel, hogy nem átlósan lép, hanem a mezők oldalára merőlegesen.

-  Vezér: A legértékesebb tiszt, ő rendelkezik a legnagyobb mozgástérrel, képes úgy lépni ahogy a futó és a bástya. Királynő néven fogunk rá hivatkozni.
-  Király: A játék céljából adódóan a legfontosabb figura a játékban. Minden irányba képes lépni a vezérhez hasonlóan, de csak egy mezőt. A király lépését befolyásolja az, hogy nem léphet sakkba. Királlyal nem lehet sakkot adni, illetve a két király nem állhat szomszédos mezőn.

Különleges lépések, és fontos szabályok:

- Sáncolás: Szokták rosálásnak is nevezni, két féle van, rövid és hosszú sáncolás, rövid amikor a király a hozzá közelebb eső bástyával sáncol, a hosszú az amikor a tőle távolabbival, az mindkét esetben megegyezik, hogy a király két mezőt lép, és a bástya átkerül a király túloldalára. sáncolásnál be kell tartani néhány szabályt:
    1. A király nem léphetett még a parti során
    2. Az a bástya amivel sáncolni akarunk nem léphetett még a játszma során
    3. Nem állhat a két sáncoló bábu között más bábu.
    4. A király nem állhat sakkban
    5. A király nem haladhat át olyan mezőn, amit az ellenfél bábuja támad.
- Ütés menet közben(en passant): Ezt a lépést akkor alkalmazhatjuk, ha az ellenfél gyalogja kezdőlépésként kettőt lépett, és a mi gyalogunk leüthette volna, ha csak egyet lép, ezt az ütést csak az ellenfél gyalogjának lépése utáni következő lépésben tehetjük meg, később már nem.
- Gyalog cseréje: Abban az esetben ha az egyik játékos gyalogja eléri a másik játékos alapsorát(világos gyalog ha belép a 8. sorba, vagy sötét az elsőbe), akkor a gyalogot lecserélheti egy általa választott akármilyen figurára, így elképzelhető egy játékban, hogy akár 18 vezér is a táblán legyen.
- Az egyik legfontosabb szabály a sakkban, hogy a király nem állhat támadás alatt, ezért ha valaki sakkot ad, a másik játékosnak kötelező megszüntetni azt a helyzetet, különben elveszíti a partit. A támadás megszüntetésére három mód

van, vagy a királlyal kell ellépni, vagy be kell lépni a támadás útvonalába, vagy le kell ütni a támadó bábut.

- Hivatalos mérkőzéseken fontos szabály még az is, hogy a fogott bábuval lépni kell, és az elengedett bábut már nem lehet áthelyezni. Ez azt jelenti, hogy ha a saját bábunkat érintjük meg, akkor mindenképp azzal kell lépni, ha az ellenfélét, akkor mindenképp le kell azt ütni. Ez alól csak az kivétel, ha a lépés nem kivitelezhető, mert adott lépéssel a király mondjuk sakkba kerülne, vagy ha előtte jelez a játékos, hogy meg szeretné igazítani a figurát.

3.2.2 Sakk960

Más néven ezt a változatot szokták Fischer random sakknak is nevezni, a kitalálójáról Bobby Fischer-ről kapta ezt a nevét. Fischer célja az volt, hogy létrehozzon egy olyan variációt, melyben sokkal nagyobb szerepe van a kreativitásnak és a tehetségnek, mint a megnyitások elemzésének és betanulásának. Ezért ebben a játékban a tisztek felállítása bizonyos szabályok betartása mellett véletlenszerűen történik. Összesen 960 felállás lehetséges, innen kapta a nevét. A kezdőálláson kívül mindenben megegyezik a normál sakk szabályaival, a cél itt is az ellenfél királyának mattolása. A felhelyezés szabályai a következők:

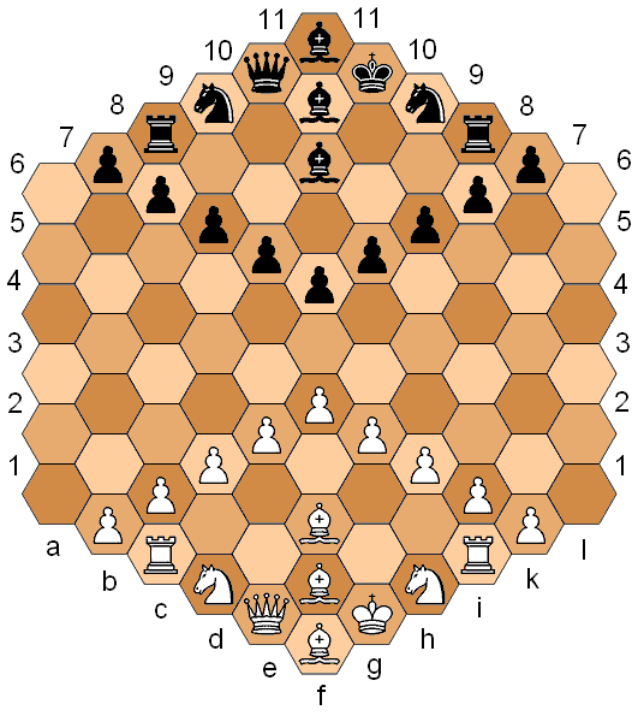
- A gyalogok ugyanúgy helyezkednek el mint normál sakkban
- A király a bástyák között kell hogy legyen
- A futóknak különböző színre kell esniük
- A sötét figurák a világos figuráknak a tükörképei szerint állnak fel

Ez a variáció 1996-ban született, 2001-óta már világbajnokságot is rendeznek belőle. Az első ilyen világbajnokságot Lékó Péter nyerte Michael Adams ellen.

3.2.3 Hexasakk

A játék hatszögletű táblán folyik, amely szintén hatszögletű mezőkből áll. A mezők színe váltakozva fehér, kávébarna és világosbarna. Az azonos színű mezők nem érintkeznek egymással. A mezők színek szerinti megoszlása:

- 30 fehér
- 30 kávébarna
- 31 világosbarna



A hexasakkban a hagyományos bábukat használjuk, de mindenkinek egyel több gyalogja és futója van. Lépési szabályok:

A *bástya* a mező oldalaira merőleges irányba haladhat tetszőleges távolságot.

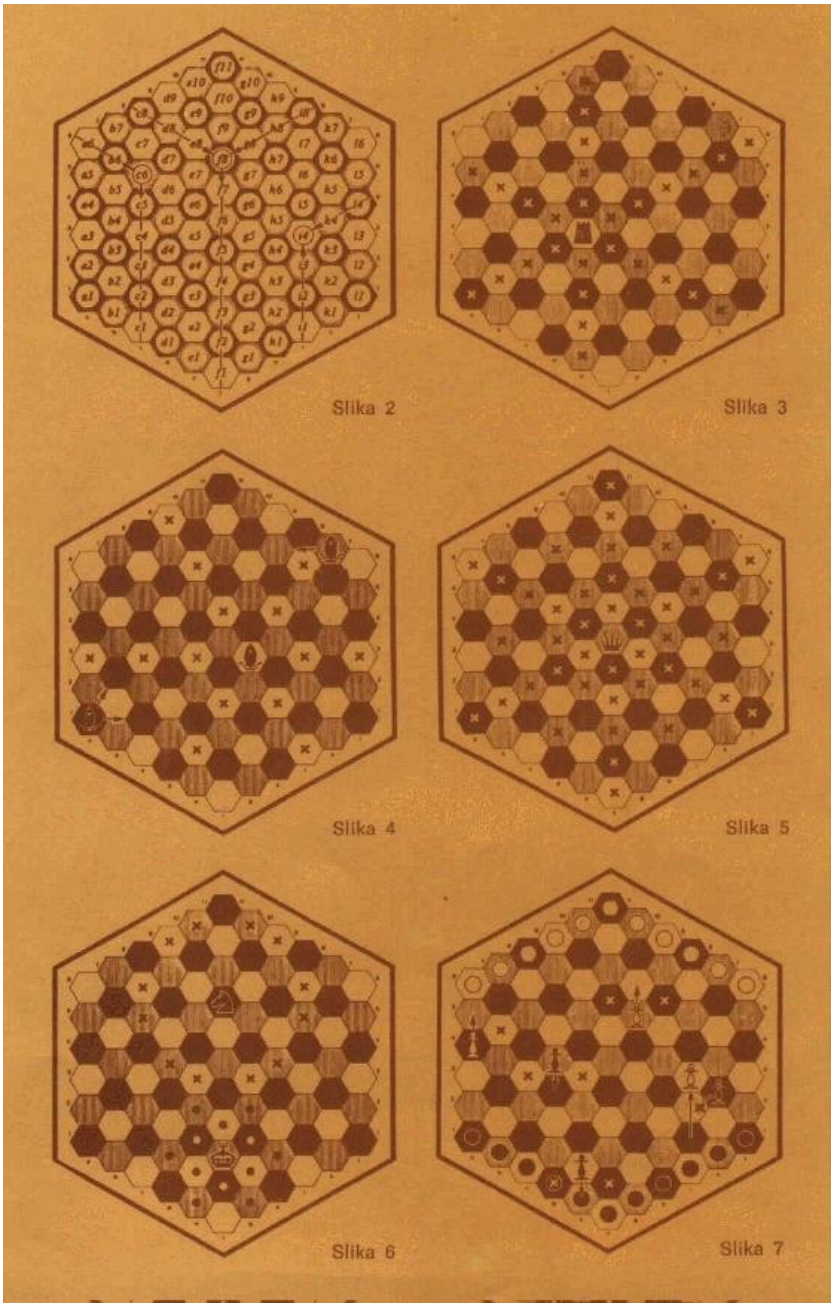
A *futó* a bástyához hasonlóan 6 irányba léphet, de nem merőlegesen, hanem átlósan, és mindig csak olyan színű mezőre léphet, amilyenen állt.

A *huszár* a normál sakkhoz hasonlóan az egyedüli, mely képes átugrani más bábukat, L-alakban mozog, nem merőlegesen vagy átlósan. Az L két szára 2 illetve 3 hosszúságú, és 120 fokos szöget zárnak be.

A *vezér* egyesíti a bástya és a futó tulajdonságait.

A *király* minden irányba léphet, de csak egy mezőt.

A *gyalog* előre léphet, kezdésből akár két mezőt is egyszerre, ütni viszont csak átlósan üthet.



3.2.4 Suicide Chess avagy franciasakk

Ezt a változatot hazánkban franciasakk-ként vált ismertté. A játék célja hogy az összes bábunkat elveszítsük. A hagyományos sakktól eltér még abban is, hogy itt kötelező ütni. Sakk nincs, en passant ütés viszont van. Ha valamelyik játékos nem tud lépni, akkor véget ér a játék, és az nyer, akinek kevesebb bábuja van, a minőség itt nem számít. Ez a variáció játszható a sakk960-nal variálva is.

A suicide chess az egyik legnagyobb elmélettel rendelkező variáns. Bár általában sokkal kevesebb lépésünk van, mint a normál sakkban, de éppen ezért, egy rossz lépésnek sokkal nagyobb következményei lehetnek, például már egy rossz megnyitás is könnyen ahhoz vezethet, hogy választási lehetőség nélkül az ellenfél összes bábuját le kell ütni, és így veszítünk.

3.2.5 Crazy House

Ennek a variációnak a lényege, hogy az ellenfél leütött bábuit vissza lehet rakni a táblára a sajátunkként. Az éppen lépni következő játékos választhat, hogy mit kíván tenni, vagy lép a táblán lévő bábuival, vagy a leütöttek közül rak fel egyet a táblára. Bár a bábukat be lehet rakni sakkra és mattra is, megszorítások itt is vannak, nem rakható be gyalog az ellenfél alapsorára, és a már átalakult gyalogokat, ha leütjük, nem tisztkét hanem gyalogként rakhatjuk fel a táblára.

3.2.6 Makarenkó-sakk

Ebben a variációban nincsenek bábuk, csak korongok. Összesen 47-47 korongot használhatnak a játékosok, és ebből építenek a hagyományos bábuknak megfelelő tornyokat. A tornyok magassága jelzi a figurát:

1 korong: gyalog

3 korong: bástya

4 korong: huszár

5 korong: futó

7 korong: vezér

8 korong: király

A lépések többfélék is lehetnek, léphet az egész torony, de léphet a torony egy része is külön. A hagyományos lépésnek megfelelő lépésnél az egész torony mozdul, de előfordulhat, hogy egy erősebb figurából két gyengébb lesz, például egy vezérből ellép 3 korong egy bástyának megfelelően, és a helyén marad még egy huszárnak megfelelő figura. Ha annyi korong marad, ami nem felel meg egyetlen figurának sem, ez kettő vagy hat lehet, akkor a kettő korong gyalognak, a hat pedig futónak számít. Az egész és a rész figurák is ráléphetnek más azonos színű korongokra, feltéve hogy az olyan mezőn van, amit le tudna ütni, ha az ellenfél bábuja lenne ott. A játék célja hogy leüssük az ellenfél összes királyát, és a játék így addig tart amíg le nem tudjuk úgy ütni a királyokat, hogy a másik játékos a következő lépésében már nem tud királyt létrehozni. Fontos szabály, hogy mindig lenni kell királynak a lépésünk után, tehát ha a király nyolc korongjából egy sem léphet el, ha nincs másik királyunk a táblán, mert akkor létrejönne olyan pillanat, amikor nincs királyunk.

3.2.7 Benedict chess

A figurák mozgása a hagyományos sakkéval megegyező, a legnagyobb különbség az, hogy itt nincs ütés, a megtámadott figurák átalakulnak a mi figuráinkká . Csak azok a bábuk alakulnak át, melyeket az aktuálisan lépő bábu támadott meg, ha az átkonvertált bábu által valami ütésbe kerülne, akkor az nem változik meg. A játék célja az, hogy sakkot adjunk, tehát hogy az ellenfél királyát átalakítsuk. Ebben a variációban nincs sakk, tehát a király lépését nem akadályozza, hogy így elvileg sakkba kerülne. Van sáncolás, de csak a király által támadott bábuk fognak átalakulni, a bástyának ilyen szempontból nincs jelentősége, itt kivételesen a sáncolásnál megengedett, hogy a király olyan mezőn haladjon át, amely az ellenfél által ütésben van. Ha a gyalog beér az ellenfél alapsorára, akkor az alapján alakulnak át az ellenfél figurái, hogy milyen tisztre cseréljük be a gyalogunkat.

Ennek a variációnak a legnagyobb hátránya, hogy a fehérrel játszó játékosnak nagyon nagy az előnye, ezért érdemes Benedict 960-at játszani, mert így a fekete hátránya nagymértékben lecsökkenhet.

3.2.8 Capture chess

Ez a változat két nagy részben és több kisebben tér el a normál sakktól. A kisebbek hogy nem lehet sáncolni, a gyalog átalakulhat királlyá, nincs sakk, a királyt is le lehet ütni. Ezen szabályok nagy része a játék céljából adódik, az összes bábu leütése a végső cél. A játék véget érhet úgy is, hogy valamelyik játékos nem tud lépni, ebben az esetben az nyer, akinek több bábuja van, ezek minősége nem számít. A másik nagy különbség az, hogy kötelező ütni. Ennek a szabálynak köszönhetően nagyon oda kell figyelni, mert egy rossz megnyitással már el is veszíthetjük akár a vezérünket is.

3.2.9 Atomic chess

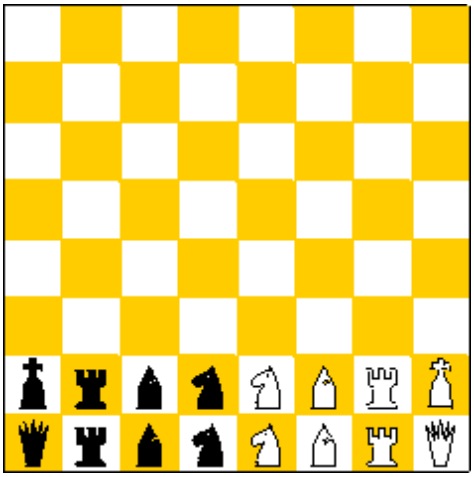
A figurák felállása és mozgása megegyezik a normál sakkból ismertekkel. A legnagyobb különbség az, hogy itt a leütött bábu, az ütő bábu, és ezeknek a szomszédos nyolc mezőn lévő tisztek is megsemmisülnek. A játék célja vagy a király bemattolása, vagy a felrobbantása. Mivel ha a király felrobbanna, ha ütne, így nem üthet, és így nem tiltott, a két király egymás mellé állása sem, ez egyébként egy esetleges rosszabb állásból patthoz segítheti a játékost. Viszont lehetővé teszi azt is, hogy akár egy vezérrel is mattot tudjunk adni. Sakkból itt is ki kell lépni, viszont nem számít sakknak, ha a robbanás fenyeget.

3.2.10 Dark sakkok

A dark játékok lényege, hogy nem látjuk az egész táblát, csak a tábla azon darabját, amelyen a mi bábuink állnak, és amit a mi bábuink elérnek. A dark játékoknak két változata van, a dark 1-nél van sakk és a cél a király bemattolása, a dark 2-nél nincs sakk, és a cél is megváltozik, le kell ütni a királyt. További variációs lehetőségek is vannak. A lao tzu-ban ez a második variáció valósul meg, abban az esetben, ha a dark 1 kerül be, akkor sun tzu-ról beszélünk.

3.2.11 Racing Kings

A figurák mozgása a megszokott, a felállás viszont nem. A játékot az nyeri, akinek a királya először eléri a 8. sort, ha a világos be tud lépni, akkor a sötét még döntetlenre mentheti a meccset, ha a következő lépésben ő is be tud lépni, így egyenlítik ki a világos kezdésből származó előnyét. Ezeken kívül a legfontosabb különbség, hogy tilos a sakk, ez nem csak annyit jelent, hogy nem lehet sakkba lépni, hanem hogy nem is lehet sakkot adni.



3.2.12 All Queens

Ez a variáció abban tér el a normál sakktól, hogy minden tiszt helyén vezérek állnak, ebből adódik, hogy nincs sáncolás sem.

4. Heurisztika, kétszemélyes játékok

4.1 A heurisztika fogalma

[4]A heurisztikára általános definíció nem létezik, a fogalma időről időre változik és adott időben is különböző szerzők különbözőképpen adhatják meg. Azonban a különböző meghatározások két lényeges tulajdonságot kiemelnek a heurisztikát alkalmazó eljárásoknak:

- Legtöbb esetben „elég jó” megoldást adnak, bár az optimális megoldás nem garantált, sőt semmilyen megoldás nem garantált.
- Jelentős mértékben javítják a problémamegoldó program hatékonyságát, leginkább a keresés próbálkozásai számának erős csökkentésével.

Mi definiálhatjuk úgy a heurisztikát, hogy az adott feladatról szerzett konkrét ismeretet, amelyet a fenti két tulajdonság elérése miatt építünk be a vezérlési stratégiába.

A heurisztika kulcsfogalom a mesterséges intelligencia területén, mivel itt azokra a problémákra keresnek megoldást, amikre direkt megoldó stratégia nem létezik, ugyanis a legtöbb problémánál a teljes keresőfát felépíteni hatalmas adatmennyiséghez, és ezzel együtt kezelhetetlenül lassú programhoz vezetne.

4.2 A heurisztika és a megoldás keresésének költsége

Amikor egy kereső számára vezérlési stratégiát választunk, azt az alapján kell megtennünk, hogy milyen eredményre van szükségünk. Ezek szerint a feladatokat a következő módon definiálhatjuk:

- A célállapotot tetszőleges számú műveletsorozattal kell meghatározni, a költség teljesen lényegtelen.
- Viszonylagosan olcsó műveletsorozat kell.
- Optimális költségű műveletsorozat kell.

Az előállítandó műveletsorozat költségét nevezzük a megoldás költségének. Különböző vezérlési stratégiákat követő keresők különböző költségű megoldásokat állíthatnak elő, és különböző költséghez vezet az is, ha ugyanabba a vezérlési stratégiába különböző mértékben építünk heurisztikát. Általános recept nincs a heurisztika írására, az, hogy mennyire jó egy adott heurisztika csak akkor dől el, ha teszteljük.

A jó heurisztika megválasztásakor nem csak az az egyedüli szempont, hogy a megoldás költsége alacsony legyen, hanem az is, hogy ennek a megoldásnak a megkeresésének a költsége is olcsó legyen. A sakkprogram esetében ez azt jelenti, hogy nem elég nagyon erőseket lépni, az is szempont, hogy ezeket a lépéseket viszonylag rövid idő alatt tegyük meg. A cél tehát az egyensúly megtalálása, a legerősebb heurisztika megírása úgy, hogy az viszonylag rövid idő alatt kiszámolható legyen.

4.3 Kétszemélyes játékok

A sakk a kétszemélyes játékoknak az úgynevezett teljes infomációjú osztályába tartozik. (Ez alól kivételt képezhetnek az egyes variációk, mint például a dark játékok.) Ezt az osztályt a következők alapján határozhatjuk meg:

- Két játékos lép felváltva adott szabályok szerint.
- Mindkét játékos birtokában van az összes a játékkal kapcsolatos információnak, tisztában vannak azzal hogyan alakult a játék, milyen az aktuális állás, és ami a legfontosabb, hogy a szerencsének nincs semmilyen szerepe a játékban. (Ez néhány esetben megváltozhat, pl.: a sakk dark változatai)
- A játék minden állásában véges számú lépés közül lehet választani.
- A játék szabályai olyanok hogy végtelen játszmák nem fordulhatnak elő, még ha ez a lépések alapján lehetséges is lenne. Szabályokkal, gyakran időkorláttal, akadályozzák ezt meg.
- A játék végén az egyik fél nyer, a másik veszít. Van ahol előfordulhat döntetlen.

4.4 Kétszemélyes játékok teljes kiértékelése

Célszerűnek tűnhet a kétszemélyes játékok összes lehetséges játszmáját irányított gráffal reprezentálni, ahol a gráf egyes szintjein lévő csúcspontok a játék adott állásait tartalmazzák, és azt, hogy melyik a soron következő játékos. A csúcspontokból kivezető élek az adott játékos egy-egy szabályos lépését jelentik. A gráf gyökerének a kiinduló állás felel meg, azok az elemek pedig, amelyekből már nem vezet ki él, a játék végállapotait tartalmazzák. Az ilyen gráfokat nevezzük játékgráfoknak. Egy konkrét játéknak felel meg az az útvonal, ami a kezdő csúcsból valemelyik végcsúcsba vezet.

A játékok ábrázolására gyakran használnak fákat a gráfok helyett. Egy állás annyiszor szerepelhet a fában, ahányféle úton eljuthatunk hozzá, aminek köszönhetően az állások száma megnövekedik, de ennek ellenére megéri ezt használni, mert a fa könnyebben kezelhető számítástechnikailag, mint egy gráf, emellett nem kell külön jelezni, hogy melyik játékos a soronkövetkező, ugyanis az egyes szintek mindig különböző játékosoknak felel meg, a párosak az egyiknek, a páratlanok a másiknak.

Bár a játékfa véges, mégis bonyolultabb játékok esetében nem éri meg próbálkozni a teljes gráf felépítésével, mert hihetetlen nagy időre lenne szükség egy adott fa teljes kiértékeléséhez.

4.5 A játékfa részleges kiértékelése

A játékfa mérete arányos a játék bonyolultságával. A nagyon sok kiterjesztést igénylő játékok fái teljesen kezelhetetlenek még a nagyon erős számítógépek számára is. (Kiterjesztésen az egyes csúcsok vizsgálatát, és ezután új csúcsok létrehozását értjük.) Megbecsülhetjük, hogy például mekkora játékfával lenne dolgunk a sakk esetében. Egy átlagos partiban 45 lépésváltás van, ami 90 szintet jelent a fában. Az egyes állapotokban számolhatunk átlagosan 35 szabályos lépéssel, ez azt jelenti, hogy a fának 35^{90} kiértékelendő levele van. Tehát látható, hogy reális időn belül ekkora mennyiségű állást kiértékelni lehetetlen, így le kell arról mondanunk, hogy kiértékeljük az egész fát. Ezáltal le kell arról is mondanunk, hogy egy biztos nyerő stratégiát határozzunk meg, csak annyit tehetünk, hogy

egy erős lépést adunk az egyik játékos számára, és miután a másik játékos lépett, újra megadunk egy - az adódott állásban - erős lépést.

A keresésnek valamilyen korlátot kell szabni, mivel nem akarjuk kiértékelni az egész fát. Általában ez mélységi korlát szokott lenni, de időhöz vagy lehetséges vizsgált állások számához is köthetjük. Az adott állások általában nem konkrét nyerő vagy vesztő állások, mivel a keresőfán túl is lehetnek még lépések. Csak akkor beszélhetünk már biztosan nyerő állásról, ha a kereső fa mélységében már vannak olyan végállapotok, melyekhez az ellenfél minden lehetséges lépését figyelembe véve el tudunk jutni. Ezért heurisztikus keresővel, az összes állás kiértékelésével a részleges fán belül adjuk meg hogy melyik az ideális következő lépés.

4.6 Minimax eljárás

A minimax eljárás a soronkövetkező játékos legjobb lépésének kiválasztására ad módszert. Ezt a játékost nevezzük MAX-nak, az ellenfelét pedig MIN-nek. Tegyük fel, hogy találunk egy megfelelő heurisztikát. Az ideális az lenne MAX számára, ha el tudna jutni a legnagyobb értékű állásba, de előfordulhat, hogy eközben MIN tud olyat lépni, amivel megakadályozza ebben MAX-ot. Ezért a célszerű figyelembe venni MIN összes lehetséges lépését, és ez alapján meghozni a döntést, hogy MIN legjobb lépését feltételezve MAX a legjobb álláshoz jusson.

A játékfát úgy építjük fel, hogy a gyökér elem az aktuális állapot, és a fa mélysége az előre beállított korlát lesz. A heurisztikának megfelelően minden csúcs kap egy értéket, aminek kiszámítását befolyásolja még az is, hogy a fa melyik szintjén van a levél, mivel a minimax eljárásnál feltételezzük, hogy a MIN a legjobb lépését választja. Ennek megfelelően a fa páros szintejein a rákövetkezőik maximumát, a páratlan szinteken a rákövetkezők minimumát kell választani. Összefoglalva, egy adott mélységű játékfában alulról felfelé haladva, minden n csúcshoz egy v(n)-nel jelölt értéket rendelünk hozzá az alábbiak szerint:

- Ha n terminális csúcs, akkor v(n) := h(n), ahol h(n) a heurisztika segítségével becsült jóság.
- Ha n nem terminális csúcs és a rákövetkezőit n_1, ..., n_k jelöli, akkor

$v(n) := \max(v(n_1),...,v(n_k))$,

amennyiben n páros szinten található, illetve

$v(n) := \min(v(n_1),...,v(n_k))$,

ha n páratlan szinten található.

Azt külön ki kell emelni, hogy ez az eljárás mindössze MAX következő lépését adja meg, egyáltalán nem biztos, hogy MIN az általunk várt lépést fogja tenni, ennek több oka is lehet:

- MIN más mélységgel dolgozik, ezért más eredményeket kap, és más lépést kell megtennie
- MIN más heurisztikát használ
- MIN más eljárást használ
- MIN már egy lépéssel tovább lát
- esetleg MIN hibázik.

4.7 Alfa-béta vágás

Ez az eljárás a minimaxon alapul, de annak egy javított változata. Általában ezzel az eljárással jóval kevesebb csúcsot kell kiértékelni, mint a minimaxnál, mégis ugyanahhoz az eredményhez vezet. A minimax eljárásban egymás után függetlenül történik:

- a fa adott mélységű generálása
- az elemek kiértékelése
- az értékek visszaadása a fában fölfelé haladva egészen a gyökérig

Javítható a hatékonyság, ha ezeket párhuzamosan hajtjuk végre, és felismerjük, hogy nem kell minden kiértékelést elvégezni. A keresés csökkentésére az ad lehetőséget, ha nyilvántartjuk a visszaadott értékekre vonatkozó korlátokat. Megállapítható, hogy:

1. A keresés során egy csúcshoz hozzá tudunk rendelni visszaadott értéket, ha:
    a. MIN csúcs esetén előállítunk egy $-\infty$ értékű rákövetkezőt,

b. MAX csúcs esetén előállítunk egy +∞ értékű utódot,
c. a csúcs minden rákövetkezőjét kiértékeltük.

2. A keresés során a nem terminális MAX csúcsokhoz alfa értéket, MIN csúcsokhoz béta értéket rendeltünk.

   a. Egy MAX csúcs alfa értéke a már létrehozott rákövetkezők értékeinek aktuális maximuma, az alsó korlát MAX visszaadott értéke számára.
   b. Egy MIN csúcs béta értéke a már létrehozott rákövetkezők értékeinek aktuális minimuma, a felső korlát MIN visszaadott értéke számára.

3. Egy nem terminális csúcshoz alfa illetve béta érték rendelhető, ha legalább egy rákövetkezőnek már van értéke
   a. Egy MAX csúcshoz tartozó alfa érték a keresés során nem csökken.
   b. Egy MIN csúcshoz tartozó béta érték a keresés során nem nő.

Ezeknek a szabályoknak megfelelően a következő esetekben lehet abbahagyni a kiértékelést:

1. Egy MIN csúcs alatti kiértékelést abba lehet hagyni, ha a hozzá tartozó béta érték kisebb vagy egyenlő, mint ezen csúcs valamely MAX őséhez tartozó alfa érték. Ekkor ugyanis a MIN csúcs visszaadott értéke nem növelheti a MAX ős számára visszaadott értéket.

2. Egy MAX csúcs alatti kiértékelést abba lehet hagyni, ha a hozzá tartozó alfa érték nagyobb vagy egyenlő, mint ezen csúcs valamely MIN őséhez tartozó béta érték. Ekkor ugyanis a MAX csúcs visszaadott értéke nem csökkentheti a MIN ős számára visszaadott értéket.

Ha a keresést az első szabály miatt korlátozzuk, akkor alfa vágásról, ha a második miatt korlátozzuk, akkor béta vágásról beszélünk

5. A lépésajánló és működése

A sakkprogramhoz implementálva lett egy minimax lépésajánló algoritmus. Számítógép elleni játék esetén a számítógép a lépését ezzel az algoritmussal fogja meghatározni. Kérhető az emberi játékos számára is lépés ajánlat, ezt külön paraméterként várja a program az indításakor. A lépésajánlás során a kimeneten láthatjuk, mit lépne a számítógép a helyünkben, de természetesen a döntés továbbra is a miénk.

Az algoritmusnak paraméterként szüksége van egy maximális mélységinformációra. Ez egy egész érték, a lehetséges lépések által generált játékfa mélységét határozza meg. A programban megadható különböző érték a számítógép és az emberi játékos számára is, ezzel akár kisebb előnyhöz juttatva az emberi játékost. A lépésajánló feladata az aktuális játékos számára a legoptimálisabb lépést meghatározni. Ezt alfabéta vágásos minimax algoritmus segítségével teszi meg.

Amikor a lépésajánlásra van szükség, akkor a *jatek* csomagban lévő *jatek* osztály példánya létrehoz egy új *AlfaBetaMinimax* példányt, és átadja neki az aktuális állapotot, illetve a beállított maximális mélységértéket attól függően, hogy emberi vagy gépi játékos számára ajánl lépést.

```
lepesAjanlo = new AlfaBetaMinimax(allapot, emberiMelyseg);
```

Az *AlfaBetaMinimax* példány kétparaméterű konstruktorában beállítja a később fontos szerephez jutó *alfa* és *beta* hasznosság-változók értékét a lehetséges minimum és maximum értékre. Ezután első lépésként megvizsgálja, hogy elértük-e a maximális mélységet – azaz a mélységet tároló változó értéke lecsökkent-e nullára – illetve azt, hogy végállapotban vagyunk-e az aktuális állás során.

```
if (allapot.vegAllapot() || melyseg == 0) {
        hasznossag = allapot.minimaxHasznossag();
}
```

Ha bármelyik feltétel teljesül, akkor beállítja a hasznosság értéket az állapothoz tartozó *minimaxHasznossag* metódus segítségével. Ez a metódus az absztrakt *Allapot* osztályban van deklarálva, a pontos definíciója viszont az aktuális *SakkAllapot* példányban található. A fentebb definiált feltételek nem teljesülése esetén az aktuális játékostól függően halad tovább a program futása.

Ha az 'A' játékos van soron, akkor az állapotra alkalmazható összes operátort bejárva megpróbálja megtalálni azt az operátort, amely alkalmazása után a legnagyobb hasznosság-értékkel bíró állapot jönne létre. Közben megvizsgálja az *alfa* és *beta* változók aktuális értékeit, és ha az *alfa* >= *beta* egyenlőtlenség igaz, akkor elvágja a játékfát az aktuális állapotnál. Ezt azért teszi, mert feltételezi, hogy az ellenfél lépése oly mértékben „jó" a számára, hogy az esetleges egyéb lépéseinek vizsgálatától innentől eltekinthet. Ha ez a feltétel sem teljesül, akkor megvizsgálja az aktuális operátort, teljesül-e rá az operátoralkalmazási előfeltétel. Ha nem, akkor lép a következő operátorra, és az *alfa* – *beta* egyenlőtlenség vizsgálattal folytatja tovább. Előfeltétel teljesülés esetén létrehoz egy új állapotot, és meghívja rá önmagát, egyel kisebb mélység illetve az aktuális *alfa* és *beta* értékekkel.

```
AlfaBetaMinimax abMinimax =
                        new AlfaBetaMinimax(uj, melyseg - 1, alfa, beta);
```

Ezzel az önrekurzióval éri el, hogy amikor végállapotban van, vagy elérte a maximális mélységet, mindig a legjobb lépés legyen letárolva. A rekurzió során visszatér a legutolsóként meghívott *AlfaBetaMinimax* példány, és folytatódik az aktuális metódus futása. Megvizsgálja, hogy a beállított hasznosság érték nem nagyobb-e az aktuális *alfa* értéknél. Ha nagyobb, akkor *alfa*-nak értékül adja ezt az értéket, és eltárolja az operátort, amivel ezt az állapotot létrehozta. Amint az összes operátort bejárta (maximum annyiszor, amekkora mélységet megadtunk a számára), beállítja a minimális hasznosság értékét *alfa*-ra vagy *beta*-ra, attól függően, melyik a kisebb a kettő közül. Erre a „? :"(kérdőjel-kettőspont) operátort használtuk helytakarékosság miatt.

```
hasznossag = alfa < beta ? alfa : beta;
```

Ha az állapotban a ’B’ játékos van soron, akkor hasonlóképpen folytatódik a program futása, apró változtatásoktól eltekintve. Ezen változtatások oka maga a minimax algoritmus. A ’B’ játékos esetén a program ugyanazokat a lépéseket teszi meg ezen eltérésekkel:

- a visszatért *AlfaBetaMinimax* példány által beállított hasznosságértéket a *beta*-val hasonlítja össze, és a *beta* értékét állítja be az egyenlőtlenség teljesülése esetén.

- az aktuális állapotra alkalmazható összes operátor vizsgálata után az aktuális állapot hasznosságának az *alfa* < *beta* esetén *beta*-t adja értékül, nem *alfa*-t.

Ezen lépésajánló hatékonyságát jobb heurisztika megírásával növelhetjük. Szerencsére a jól megírt heurisztika nem lassítja le nagyon a programunk futását. A heurisztikával szemben a maximális mélység információ az, ami behatárolja a gépi ellenfél képességeit. Állapotonként sok alkalmazható operátor esetén óriási számításigényre lehet szükség, és ez a sakkjátékokra különösen igaz. Minél nagyobb maximális mélységet tudunk megadni, a gépi ellenfél, vagy esetünkben akár az ajánlott lépés annál pontosabb és hatékonyabb lehet. A híresebb sakkprogramok is a finomított heurisztika mellett nagy mélységben képesek vizsgálni az aktuális ellenfél lépéseit. A mi sakkprogramunknak is csak az alatta futó számítógép jelent határt. Illetve a számára megírt *minimaxHasznossag* heurisztika, amelyet - ha birtokunkban van a forráskód – pillanatok alatt meg tudunk változtatni, optimalizálva, jobbá téve ezzel a lépésajánló működését.

6. Az operátoraink,
és az *operatorok* kollekció feltöltése

6.1 Az operátoraink fajtái

Minden egyes operátort, amit használhatunk az *allapotter* csomag *Operator* osztályából származtatunk. Ezzel elérhetjük azt, hogy egy bizonyos szinten közösen tudjuk kezelni az egyébként egymástól eltérő operátorokat. Az operátoroknál az egyszerű lépést és az egyszerű ütést megvalósító operátort közösen kezeljük, bábutól függetlenül. Ezeket egy *Lep* operátorral tudjuk megvalósítani.

```
if (allapot.sakkTabla[k][j] == 0) {
        temp.add(new Lep(i, j, k, j));
}
```

Maga az operátor 4 egész értékű paramétert tartalmaz, amelyekkel egyértelművé tehetjük, hogy a táblán honnan hova szeretnénk lépni. A lépést és az ütést sem korlátoztuk le az egyes bábuk által megtehető lépésekre. Létezik továbbá *SancolLep* és *EnpassantLep* operátor is. A *SancolLep* operátor értelemszerűen a sáncolást, az *EnpassantLep* operátor pedig a menet közbeni ütést fogja végrehajtani. A *SancolLep* paraméterként tartalmaz egy egész értéket, ezen érték beállításával tudjuk meghatározni melyik sáncolást szeretnénk végrehajtani. A lenti kódrészlet azt mutatja meg, hogy mely szükséges, de nem elegendő feltételekre van szükség a sáncolás operátor elkészítéséhez.

```
if (hosszuSancolFekete == true) {
                if (allapot.sakkTabla[8][5] == 0 &&
                   allapot.sakkTabla[8][6] == 0 &&
                   allapot.sakkTabla[8][7] == 0 &&
                   allapot.sakkTabla[8][8] == FEKETE_BASTYA) {
                        temp.add(new SancolLep(4));
                }
```

Az EnpassantLep operátor pedig tartalmazza a lépést végrehajtható bábut, a játékos színét, akinek ezt engedélyezzük, illetve az ütés irányát. A lenti kódrészletben a fehér játékos számára készítünk egy balról menetközben ütő *enpassantLep* operátort.

```
EnPassantLep el = new EnPassantLep(FEHER, BALROL, hovax, hovay - 1);
```

Ezen operátorok csak akkor jönnek létre, ha az *szabadSancolni* vagy az *enpassant* változók értéke igaz. Ennek a különböző variációknál van jelentősége. Normál alapjáték esetén ugyanis mindkét lépés megengedett, természetesen csak akkor, ha teljesülnek a lépés előfeltételei. A játék tartalmaz továbbá egy *VisszaRak* nevű operátort is, amelynek a crazy house játéknál és változatainál lesz szerepe. Ezt operátort ott fogjuk ismertetni. Azt, hogy egy adott operátor a sakk szabályait az aktuális tábla állapota mellett betartsa, az *operatorEllenorzes* metódus meghívásával tudjuk majd ellenőrizni.

6.2 Az operátoraink feltöltése

Az *operatorok* kollekcióban tartjuk nyílván az aktuális játékos által az aktuális állapotra alkalmazható operátorokat. Hasonlóképpen az *ellenfelOperatorok* kollekció tartalmazza az ellenfél által az adott állapotra alkalmazható operátorokat. Ezen kollekciókat vagy a kezdőállapot beállítása során vagy az operátor alkalmazás során előálló új állapotnál töltjük fel. A feltöltéshez a *feherOperatorokFeltoltese* és a *feketeOperatorokFeltoltese* metódusokat használjuk. A két metódus nagyon hasonló, így egyszerre tárgyaljuk mindkettőt.

```
this.ellenfelOperatorok.addAll
        (operatorEllenorzes(feketeOperatorokFeltoltese(this), this));
```

A feltöltés előtt létrehozunk egy segéd kollekciót, amelyet kettő - a táblát bejáró egymásba ágyazott - ciklussal töltünk fel. Amint végigértünk a táblán, akkor eltároltuk az összes lehetséges alkalmazható hagyományos sakk operátort a segéd kollekciónkban. Ez a

kollekció lesz a *feherOperatorokFeltoltese* és a *feketeOperatorokFeltoltese* metódusok visszatérési értéke. A lentebb ismertetendő feltételeken túl egyéb feltételek teljesülésére is szükség lesz az operátorok szabályos alkalmazásához.

6.2.1 Gyalog bábu a táblán

A bejárás során, ha egy - a játékos színével megegyező - gyalogot találunk, akkor három különböző lehetőséget vizsgálunk meg. Ha nem áll előtte bábu, akkor létrehozunk egy operátort, amely alkalmazásával a gyalog egy mezőt halad előre. Ha az előtte lévő és az azelőtt lévő mezőn sem áll bábu, akkor szintén létrehozunk egy operátort, amivel a második mezőre léphet a gyalog. Ha tőle egy mezővel balra vagy jobbra illetve egy mezővel az ellenfél felé előrelépve az ellenfél bábuja áll, akkor szintén létrehozunk egy operátort, amellyel az ellenfél bábuját le tudjuk ütni. Ezt ugyanúgy egy Lep operator példányosításával érjük el, nincs szükség külön az ütést végrehajtó operátorra.

6.2.2 Király bábu a táblán

Ha a bejárás során az aktuális mezőn egy királyt találunk, akkor a honnan paraméternek a király által elfoglalt pozíciót, a hova paraméternek pedig a körülötte levő maximum 8 mezőt állítjuk be. Ez változhat aszerint, hogy a pálya szélén áll-e az adott király. Ez esetben is egy *Lep* operátort fogunk példányosítani. További előfeltételeket az operátorok létrehozásának ezen szakaszában nem vizsgálunk, ezt majd az *elofeltetel* metódusban fogjuk megtenni. Király bábu esetén létrehozhatunk egy *SancolLep* operátort is, ha a *szabadSancolni* változó értéke igaz. Ezen lépés előfeltételét is később fogjuk megvizsgálni.

6.2.3 Huszár bábu a táblán

Ha az aktuális mezőn huszárt találunk, akkor szintén egy *Lep* operátort fogunk példányosítani. Az operátor paraméterinek megadjuk az aktuális mezőt, ahol a huszár található, illetve egyenként maximum 8 lehetséges mezőt, ahova a huszár léphet. Ezen lépések számát a sakktáblán elfoglalt pozíció határozza meg. A huszár lehetséges lépéseit az alábbi képen ábrázoljuk.

6.2.4 Futó bábu a táblán

A futó lépései ennél kicsit bonyolultabban jönnek létre. Ezen bábu lépéseihez is a *Lep* operátort fogjuk használni. A futó kezdőpozíciójából kiindulva megvizsgáljuk a tőle egy oszloppal balra és egy sorral felfelé álló mezőt. Ha itt a mi színünknek megfelelő bábut találunk, akkor továbblépünk a következő sakktábla mezőre, ugyanis a saját bábunkra lépni sem tudunk, illetve nem is tudjuk leütni. Ha az ellenfél bábuját találjuk ezen a mezőn, akkor ezt a bábut leüthetjük. Erre továbbra is a *Lep* operátort használjuk, amelynek a hova mezőjébe a legutolsó vizsgált mezőt állítjuk be. Ezután haladunk a következő mezőre tovább. Harmadik lehetőségünk az, ha üres mezőt találunk tőlünk a táblán egy oszloppal balra és egy sorral felfelé lépve. Ekkor létrehozunk egy *Lep* operátort, majd haladunk tovább ebbe az irányba. Ezen lépéseket addig ismételjük, amíg el nem értük a pálya szélét, vagy bábut nem találtunk az aktuálisan vizsgált mezőn. A lenti kódrészlet a fehér futó egyik irányban megtehető lehetséges lépéseit mutatja be.

```
for (int k = i + 1, l = j - 1; k < 9 && l >= 1; k++, l--) {
        if (allapot.sakkTabla[k][l] == 0) {
                temp.add(new Lep(i, j, k, l));
        }
        else {
                if (allapot.sakkTabla[k][l] <= 0) {
                        temp.add(new Lep(i, j, k, l));
                }
                break;
        }
}
```

6.2.5 Bástya bábu a táblán

A táblán találhatunk továbbá bástya figurát is. A futó lépéseihez hasonlóan megvizsgáljuk a bástyával függőlegesen és vízszintesen egyvonalban lévő bábukat. Ha saját bábut találunk, akkor a bástya lépéseinek ebbe az irányba történő lépéseivel véget értünk, és haladhatunk a következő irányra. Ha az ellenfél bábuját találtuk meg, akkor példányosítunk egy újabb *Lep* operátort a bástya által elfoglalt pozícióval és az ellenfél bábujának helyzetével. Ezután haladunk tovább a többi irányba. Ha a vizsgálat során üres mezőt találunk, akkor erre a mezőre is példányosítunk egy operátort, és haladunk tovább, de ugyanebben az irányban. Miután ezeket a lehetőségeket megvizsgáltuk mind a négy irányban, akkor az aktuális bástyának elfogytak a további lehetséges lépései. A sáncolás vizsgálatát ugyanis akkor tesszük meg, amikor király bábuval találkozunk a táblán.

6.2.6 Királynő bábu a táblán

Ha királynő bábuval találkozunk, akkor lényegében a bástyánál és a futónál felmerült lehetőségeket vizsgáljuk meg. Ez kódszinten teljesen megegyezik az említetteknél történt leírással. Össze is lehetett volna vonni a kettőt, de átláthatósági szempontból külön lettek választva a bástya és a futó illetve a királynő lépései.

7. Az operátorok ellenőrzése és alkalmazása

7.1 Az *operatorEllenorzes* metódus

Mielőtt használni tudnánk a *feherOperatorokFeltoltese* és a *feketeOperatorokFeltoltese* metódusok által feltöltött kollekciókat, végre kell hajtanunk rajtuk egy operátorellenőrzést. Ennek az elvégzésében a beszédes nevű *operatorEllenorzes* metódus segít nekünk.

```
public Collection<Operator> operatorEllenorzes(Collection<Operator> _operatorok,
        SakkAllapot _allapot) {
                Collection<Operator> uj = new HashSet<Operator>();
                for (Operator o : _operatorok) {
                        if (elofeltetel(o, _allapot)) {
                                uj.add(o);
                        }
                }
                return uj;
        }
```

A metódus egy operátor kollekciót és egy állapotot vár paraméterül. Gyárt egy új segéd kollekciót, majd bejárja a paraméterül kapott operátorok kollekcióját. A bejárás során megvizsgálja az összes kollekcióban található operátort, és azokat, amelyek teljesítik az operátoralkalmazási előfeltételt, beszúrja az új kollekcióba.

7.2 Az *elofeltetel* metódus

Az *elofeltetel* metódus a program korábbi verzióihoz való kompatibilitás miatt többféle paraméterezéssel is ellátható.

```
@Override
 public boolean elofeltetel(Operator op) {
        return elofeltetel(op, this);
}
public boolean elofeltetel(Operator op, SakkAllapot _allapot) {}
```

s állapotban és egyúttal az aktuális sakktáblán vizsgálja meg azt, hogy a paraméterként kapott

operátor teljesíti-e a játék alapszabályai által generált előfeltételeket. Ezzel a metódussal vizsgáljuk meg az előfeltételek teljesülését a végállapot vizsgálatánál, a menet közbeni ütés soron kívüli operátorokhoz hozzáadásánál, illetve a minimax algoritmusnál is.

Második paraméterként megadhatjuk azt az állapotot is, amelyben szeretnénk, a vizsgálatot végrehajtani. Ezekre az *operatorEllenorzes* metódus fog hivatkozni. Erre a metódusra az esetleges klónozott állapotok vizsgálatánál lesz szükség, melynek szerepét később fogjuk kifejteni.

A kétparaméteres *elofeltetel* metódus egy operátorvizsgálattal indul. Mivel az operátorainkat alapvetően egységesen kezeljük, ezért a program ezen szakaszában szét kell bontani az egyes operátorok fajtái szerint az előfeltételek vizsgálatát.

7.2.1 A *Lep* operátorokra vonatkozó előfeltételek

Ha az aktuálisan vizsgált operátor *Lep* operátor, akkor belépünk a hozzá tartozó blokkba, és lekérjük az adattagjait. Először megvizsgáljuk, hogy király bábuval szeretnénk-e lépni. Ha ez a feltétel teljesül, akkor meghívjuk a *vaneKorulotteKiraly* metódust paraméterként átadva az aktuális király helyzetét. Ezután megvizsgáljuk, hogy a lépés megtétele után sakkban lenne-e az aktuális játékos királya. Ezt úgy tesszük meg, hogy létrehozunk egy klón állapotot, amin végrehajtjuk a lépést. Beállítjuk a klón állapot játékosát, a klón játékos királyát, illetve ezen játékostól függően létrehozzuk az ellenfél operátorait.

```
clone.ellenfelOperatorok.addAll(clone.jatekos == 'A' ?
        feketeOperatorokFeltoltese(clone) :
        feherOperatorokFeltoltese(clone));
```

Ezután a *sakkbanVan* metódussal eldöntjük, hogy az adott klónállapotban sakkban van-e az adott játékos. Ha sakkban volt a lépés után, akkor az operátor - amellyel létrehoztuk a klónállapotot - szabálytalan lépést tartalmaz, így az *elofeltetel* metódus hamis értékkel tér vissza. Ha ezen az utolsó akadályon is túljutott az operátorunk, akkor a lépés szabályos, és a metódus visszatérési értéke igaz lesz.

Annak, hogy az aktuális állapotban sakkban vagyunk-e vagy sem, ebben a

kontextusban nincs jelentősége. Ugyanis ha sakkban voltunk a lépés előtt, akkor ugyanúgy csak azon operátorokat nyilvánithatjuk szabályosnak, amelyek végrehajtása után a sakk állapot megszűnik. Ha nem voltunk sakkban a lépés vizsgálatakor, akkor is csak azon lépéseket engedhetjük meg, amelyek végrehajtása után nem kerültünk sakk állapotba. Így a dupla vizsgálat felesleges, és a két feltétel egybeolvasztható.

7.2.2 Az *EnpassantLep* operátorokra vonatkozó előfeltételek

Ha az alábbi kódrészlet igazat ad vissza, akkor egy EnpassantLep operátornak fogjuk az előfeltételeit vizsgálni.

if (op instanceof EnPassantLep)

Ezen operátor a menet közbeni ütést fogja megvalósítani. Azaz ha az ellenfelünk az egyik gyalogjával átugrotta a mi gyalogunk által védett mezőt, akkor azt a gyalogot leüthetjük, és a védett mezőre léphetünk a gyalogunkkal. Ezen lépésnek a konkrét előfeltétele az, hogy a lépést közvetlenül az ellenfél lépése után hajtsuk végre. Ugyanis ha nem éltünk ezzel a lehetőséggel, akkor később már nem üthetünk menet közben ezzel a bábuval. A lépés másik előfeltétele, hogy a lépés után ne kerüljünk sakkhelyzetbe, továbbá ha sakk helyzetben voltunk, akkor ezt szüntessük meg a lépés után. Ezt a *Lep* operátornál leírtakhoz hasonlóan fogjuk megtenni. Ehhez ismét szükségünk van egy klón állapotra, amelyben ugyanaz a játékos következik lépni, mint a vizsgált állapotunkban. Aszerint állítjuk be a klón tábla új állapotát, hogy melyik játékos illetve melyik irányba szeretné megtenni az menet közbeni ütést.

```
if (enp_irany == BALROL) {
        if (enp_szin == FEHER) {
                clone.sakkTabla[enp_melyik_babuvalx][enp_melyik_babuvaly] = 0;
                clone.sakkTabla[enp_melyik_babuvalx - 1][enp_melyik_babuvaly + 1] =
                        FEHER_GYALOG;
                clone.sakkTabla[enp_melyik_babuvalx][enp_melyik_babuvaly + 1] = 0;
        }
}
```

Ezután újra létrehozzuk az ellenfél klón állapotra érvényes operátorait, és a *sakkbanVan* metódus segítségével megvizsgáljuk, hogy a lépés után előállt-e sakk helyzet. Ha igen, akkor az operátor szabálytalan, ha nem, akkor az operátor szabályos, és igaz értékkel tér vissza az *elofeltetel* nevű kiértékelő metódusunk.

7.2.3 Az *SancolLep* operátorokra vonatkozó előfeltételek

SancolLep operátor esetén első lépésként lekérjük az operátortól, hogy melyik sáncolásról van szó. Erre azért van szükség, mert bár ritka esetben, de előfordulhat, hogy egyszerre a hosszú és a rövid oldalon is tudunk sáncolást végrehajtani, így kénytelenek vagyunk a lépés elején eldönteni, hogy melyikről van szó. Ezután megvizsgáljuk, hogy az adott állapotban az adott játékos sakkban van-e. Ha igen, akkor nem alkalmazhatja a sáncolás lépést, és az *elofeltetel* metódus visszatérési értéke hamis lesz. Ha nem volt sakkban a játékos, akkor a metódus halad tovább a következő feltételre, és megvizsgáljuk, hogy azon mező körül, ahova a királlyal lépni fogunk, nincs-e közvetlenül az ellenfél királya. Szintén szélsőséges esetnek számít, de azért erre is figyelnünk kell. A harmadik és egyben utolsó feltételünk pedig az, hogy az ellenfél ne támadja se azt a mezőt, ahova a királlyal lépni fogunk a sáncolás folyamán, se azt a mezőt, amelyen áthalad a sáncoló játékos királya. Ha mindhárom feltétel teljesült, akkor az operátor szabályos, és igaz értékkel tér vissza az előfeltétel ellenőrző metódus.

```
for (Operator eop : _allapot.jatekos == this.jatekos ?
        this.ellenfelOperatorok : this.operatorok) {
            if (eop instanceof Lep) {
                Lep el = (Lep) eop;
                int ehovax = el.getHovax();
                int ehovay = el.getHovay();
                if (x == ehovax && (y == ehovay || z == ehovay)) {
                    return false;
                }
            }
    }
```

7.3 Az *alkalmaz* metódus

Ha az operátor kollekcióinkat szabályosan feltöltöttük, és bekértük a felhasználótól vagy a gépi játékostól az általa meghatározott legoptimálisabb lépést, akkor alkalmazhatjuk az operátort. Az operátor alkalmazása során első lépésben létrehozunk egy új állapotot, majd megvizsgáljuk, milyen operátort szeretnénk alkalmazni, és csak a neki megfelelő ágba lépünk be.

7.3.1 A *Lep* operátor alkalmazása

Lep operátor esetén első lépésként lekérjük az operátor adattagjainak értékét. Ezután megvizsgáljuk, hogy nem-e ütés, vagy gyaloglépés történt-e. Ha ütés történt, akkor ürítjük az eddigi állapotokat tartalmazó listát, majd ezen lista első elemének az aktuális állapotot állítjuk be egyes számossággal. Ha gyaloglépés történt, akkor szintén ürítjük a listát, és felvesszük bele egyedülálló elemként az aktuális állapotot továbbá az utolsó gyaloglépés óta eltelt lépések számát nullára állítjuk.

```
if (Math.abs(sakkTabla[honnanx][honnany]) == FEHER_GYALOG) {
        uj.setLepesekSzama(0);
        allapotokListaja.clear();
        allapotokListaja.put(uj, 1);
}
```

Ezekre a döntetlen állás vizsgálatakor lesz szükség. Az állapotokat tartalmazó lista ürítésére azért van szükség, mert ha ütés vagy gyalog lépés történt, akkor az előző állapotok ezentúl nem fordulhatnak többé elő. Ezen lista törlése nem lett volna kötelező, mégis megtesszük a program az erőforrásokkal való minimálisan jobb gazdálkodásának érdekében. Ezután, ha nem volt gyaloglépés, akkor növeljük a gyaloglépések számát eggyel, illetve, ha ütés volt, akkor pedig elrakjuk az adott bábut a *LeutottBabuk* listába.

```
if (sakkTabla[hovax][hovay] != 0) {
        uj.LeutottBabuk.add(sakkTabla[hovax][hovay]);
}
```

Következő vizsgálatunk tárgya a gyalog az ellenfél kezdősorára való beérése. Ha a gyaloglépésünk következtében elértük a tábla tetejét vagy alját, akkor lehetőségünk van bábut cserélni. A bábucserét a *csere* metódus valósítja meg. A visszatérési értéke egy egész érték lesz, amely egy bábut fog jelenteni. A metódus során a gépi vagy az emberi játékos választ magának egy bábut a leütött bábui közül. Gépi játékos esetén bejárja az összes leütött bábut, majd az összes a saját színével megegyező bábu visszatételére hasznosságot számol. Amelyik bábu visszarakásával a legjobb állást érné el, azt fogja választani.

```
if (_jatekos == 'A') {
        for (Integer i : LeutottBabuk) {
                if (i > 0) {
                        _allapot.sakkTabla[_hovax][_hovay] = i;
                        temp = minimaxHasznossag(_allapot);
                        if (minertek < temp) {
                                minertek = temp;
                        }
                }
        }
        for (Integer i : LeutottBabuk) {
                if (i > 0) {
                        _allapot.sakkTabla[_hovax][_hovay] = i;
                        if (minertek == minimaxHasznossag(_allapot)) {
                                return i;
                        }
                }
        }
}
```

Ha nem a gépi játékos hajtja végre a cserét, akkor meghívjuk a *cseretBeker* metódust, amely bekéri a játékostól a számára legjobb választásnak tűnő bábut. A heurisztika számolására azért van szükség, mert nem mindig a legerősebb bábu vezet a győzelem felé.

Miután végrehajtottuk vagy sem az esetleges cserénket, üressé tesszük azt a mezőt, ahonnan elléptünk a bábuval. Ezután megvizsgáljuk, hogy királlyal vagy bástyával történt-e a lépés. Ha bármelyik feltétel igaznak bizonyul, akkor a neki megfelelő sáncolási lehetőségeket hamisra állítjuk. Ennek az az oka, hogy sáncolni csak a játék során mozdulatlan bábukkal lehet. A következő feltételünk azt vizsgálja, hogy a lépésünk gyaloggal történt-e. Ekkor lehetséges, hogy az ellenfél menet közbeni ütést hajthat végre. Ha a gyalogunkkal átugrottunk

egy az ellenfél gyalogja által támadott mezőt, akkor létrehozzunk a menet közbeni ütést megvalósító *EnpassantLep* operátort. Ezt az operátort az *elofeltetel* metódusnak átadva megbizonyosodhatunk arról, hogy az adott operátor szabályos lépést hajt-e végre. Ezután az új állapot számára beállítjuk játékosnak az ellenfelet, eltároljuk a királyát, illetve beállítjuk számára az operátorait. Végül már csak egy feladatunk maradt: Megvizsgáljuk, hogy az új állapotban az állapot játékosa sakkban van-e, illetve ha ez igaz, akkor ezt az értéket eltároljuk az állapotban. Ezután visszatérünk az újonnan létrehozott állapottal.

7.3.2 Az EnpassantLep operátor alkalmazása

Ha az alkalmazandó operátor *EnpassantLep*, akkor menet közbeni ütést kell végrehajtanunk. Mivel ezt az operátort az előző játékos *alkalmaz* metódusában adtuk hozzá az operátorainkhoz, ezért a szabályoknak megfelelően azonnal fel lesz használva. Ha ez nem történne meg, akkor a lépés elveszne, és később már nem lehetne végrehajtani.

Az EnpassantLep során először beállítjuk az utolsó gyaloglépések óta eltelt lépések számát nullára, ugyanis ezt a lépést mindenképpen gyalog bábuval tesszük meg. Ezután az operátor paramétereiben található játékos színe, a lépés iránya, és a lépő bábu segítségével beállítjuk a lépés után létrejött tábla állapotot.

```
if (enp_irany == BALROL) {
        if (enp_szin == FEHER) {
              uj.sakkTabla[enp_melyik_babuvalx][enp_melyik_babuvaly] = 0;
              uj.sakkTabla[enp_melyik_babuvalx + 1][enp_melyik_babuvaly + 1] =
                FEHER_GYALOG;
              uj.sakkTabla[enp_melyik_babuvalx][enp_melyik_babuvaly + 1] = 0;
              hovax++;
        }
}
```

A lépés után töröljük az eddigi állapotokat, illetve felvesszük az új állapotot, majd a *Lep* operátornál leírtak szerint haladunk tovább a játékos színének beállításával.

7.3.3 A *SancolLep* operátor alkalmazása

A *SancolLep* operátor alkalmazása során első lépésként növeljük az utolsó gyaloglépés óta eltelt lépések számát. Ezután végrehajtjuk magát a sáncolást, azaz a bástyát és a királyt a sáncolást tartalmazó operátor paramétere alapján a megfelelő pozíciókba helyezzük.

```
switch (melyiket) {
        case 1: {
                uj.sakkTabla[1][2] = FEHER_KIRALY;
                hovax = 1;
                hovay = 2;
                uj.sakkTabla[1][3] = FEHER_BASTYA;
                uj.sakkTabla[1][1] = 0;
                uj.sakkTabla[1][4] = 0;
                break;
        }
}
```

Végül töröljük az eddig eltárolt állapotokat, majd az állapotokat nyilvántartó listába beszúrjuk az aktuális állapotot. Ezután a *Lep* operátornál leírtak szerint folytatjuk az új állapot beállítását a játékosával, annak királyával, az operátorokkal, illetve a *sakkbanVan* változóval.

8. Az alapjáték menete

Az alapjáték - ahogy a variánsok is - a *jatek* osztály példányosításával kezdődik. Paraméterként megkap egy kezdő sakkállást, illetve egyéb beállítási lehetőségeket, mint:

- egy további közös változóban további paramétereket:
    - számítógép ellen játsszunk-e vagy 2 emberi játékos játszik
    - kérünk-e a számítógéptől lépésajánlatot vagy sem
    - ha gépi ellenfél ellen játszunk, akkor ki legyen a kezdőjátékos
- a gép számára beállított lépésajánlás maximális mélységének méretét
- a játékos számára beállított lépésajánlás maximális mélységének méretét

/* A végleges példányosításról szövegdoboz*/

8.1 Közös inicializáló blokk lefutása

A kezdőállapotot beállító konstruktor végrehajtása előtt lefut egy példányszintű közös inicializáló blokk, amely létrehozza a kötelezően minden állapot által tartalmazott tagokat illetve esetlegesen beállít ezeknek egy kezdőértéket. Ezen tagok rendre:

- az állapothoz tartozó sakktábla
- az állapothoz tartozó *sakkbanVan* változó
- az aktuális játékos operátorai, illetve az ellenfél operátorai
- *LeutottBabuk* listája
- egy láthatósági mátrix
- *az_allapotban_kotelezo_utni* változó

A táblát az egyszerű kezelhetőség érdekében 9*9-es egész értékeket tartalmazó mátrixként tároljuk. Erre azért van szükség, mert a java nyelvben a tömbök (mátrixok) indexelése 0-tól indul és nagymennyiségű hibát ki lehet úgy szűrni, hogy a tömb által tartalmazott metódusokat emberi nyelven, példák alapján készítjük el.

A *sakkbanVan* változó értékét alapértelmezetten hamisra állítjuk, egy későbbi vizsgálat folyamán állítjuk majd át igazra, ha erre szükség lesz.

Az aktuális játékos és az ellenfél operátorai bár hasonlóak és könnyen megkülönböztethetőek, mégis külön vannak tárolva. Ennek az az oka, hogy gyakran csak az egyik játékos operátoraira van szükség, és ezzel értékes tizedmásodpercekre tehetünk szert, ami könnyen megsokszorozódhat a lépésajánló futtatása során.
A *LeutottBabuk* listája tartalmazza a közösen a két fél által leütött bábukat. Ezt a listát egyben érdemes kezelni, ugyanis a játék folyamán viszonylag kisméretű marad, és könnyen szétválaszthatóak az egyes játékosok leütött bábui.
(Nem az alapjáték része, de az inicializáló blokkunk tartalmaz egy a tábla láthatóságát befolyásoló mátrixot is, illetve egy *az_allapotban_kotelezo_utni* változót, amely igaz értéke esetén csak olyan lépéseket enged majd megtenni, amely során az ellenfél egyik bábuját leütjük. Ezekre különböző variációk esetén lehet majd szükségünk.)

8.2 A *SakkAllapot* konstruktora

A blokk lefutása után meghívódik a *jatek* osztály példánya által meghívott konstruktor. Kezdésként beállítja a játék alapvető paramétereit, majd felrakja a táblára a hagyományos sakkban definiált alapállást.

```
sakkTabla[1][1] = sakkTabla[1][8] = FEHER_BASTYA;
sakkTabla[8][1] = sakkTabla[8][8] = FEKETE_BASTYA;
sakkTabla[1][2] = sakkTabla[1][7] = FEHER_HUSZAR;
sakkTabla[8][2] = sakkTabla[8][7] = FEKETE_HUSZAR;
sakkTabla[1][3] = sakkTabla[1][6] = FEHER_FUTO;
sakkTabla[8][3] = sakkTabla[8][6] = FEKETE_FUTO;
sakkTabla[1][5] = FEHER_KIRALYNO;
sakkTabla[8][5] = FEKETE_KIRALYNO;
sakkTabla[1][4] = FEHER_KIRALY;
sakkTabla[8][4] = FEKETE_KIRALY;
for (int i = 1; i < 9; i++) {
        sakkTabla[2][i] = FEHER_GYALOG;
        sakkTabla[7][i] = FEKETE_GYALOG;
}
```

Ezután az *enpassant* és a *szabadSancolni* változóknak igaz értéket ad, mivel az alapjátékban sem a menet közbeni ütés sem a sáncolás nem tiltott. A *szabadSancolni* változóhoz kapcsolódóan beállít négy segédváltozót, amelyek igaz értéke esetén lehet majd az egyes játékosoknak a hosszú, avagy a rövid oldalon sáncolniuk. Ezután példányosít egy *allapotokListaja* nevű változót, amely értelemszerűen az állapotok listáját fogja tartalmazni. Kezdőjátékosnak az 'A' játékost állítjuk be. Szintén a gyorsítás érdekében előre eltároljuk az aktuális játékos és az ellenfél királyát is, ezekre majd később lesz szükség. Ezután feltöltjük az alkalmazható operátoraink és az ellenfél operátorainak kollekcióját is. Ennek a menetére az „Az operátorok, és feltöltésük" fejezetben részletesen kitérünk. Az operátorok feltöltése után megvizsgáljuk, hogy az aktuális játékos sakkban van-e. Ha a *sakkbanVan* metódus igaz értékkel tér vissza, akkor ezt beállítjuk a kezdőállapot számára. Ezen kívül a kezdőállapotban már csak egy fontosabb változónak adunk értéket, ez pedig a *lepesekSzama*. Ebben a változóban az utolsó gyaloglépés vagy ütés óta eltelt lépések számát tároljuk. Ugyanis ha ezen lépések száma eléri az 50-et, akkor a játszma döntetlennel fog véget érni.

8.3 A *Jatek* osztály és a *jatszik* metódusa

Miután a kezdőállapotot beállítottuk, visszatérünk a hívó függvénybe, átadva a kezdőállapotot az aktuális *Jatek* példánynak, és meghívjuk a konstruktorát a fent kifejtett paraméterekkel. Ezután elindítjuk a tényleges játék algoritmusát a *jatek.jatszik();* utasítással.

A *jatszik* metódus működése nagyon egyszerű. Amíg nem vagyunk végállapotban, addig felváltva kér be egy-egy lépést a játékot játszó felhasználótól illetve a másik játékostól, legyen az gép vagy ember. Ha a számítógép a soron következő játékos, akkor példányosít egy lépésajánlót az *AlfaBetaMinimax(allapot, gepiMelyseg)* konstruktorral. Ennek részletes leírása a „A lépésajánló és működése" fejezetben található. Amikor a konstruktor által előidézett rekurzió véget ér, a lépésajánlónak lesz egy alkalmazható operátora. Ezután kiírjuk a számítógép lépését. Ha emberi játékos van soron és nem kér lépésajánlatot, akkor meghívjuk az aktuális állapot *beker* metódusát, amellyel bekérünk a felhasználótól egy általa végrehajtani kívánt lépést. Ha játékos kért ajánlatot, akkor szintén példányosítunk egy lépésajánlót, ezúttal *gepiMelyseg* helyett *emberiMelyseg* paraméterrel. A lépésajánló visszatér egy operátorral, amit kiírunk, és ugyanúgy meghívjuk a *beker* metódust, mivel ez az operátor

csak egy ajánlat volt, nem pedig döntés. Ha a választott operátor sáncolás volt, akkor ezt eltároljuk, ugyanis a hasznosság kiszámításánál a sáncolásért plusz pont jár az adott játékosnak. Ezután végrehajtjuk a választott operátort az aktuális állapoton, majd megvizsgáljuk, hogy tartalmazza-e már az állapotok listája az aktuális állapotot. Ha nem tartalmazta, akkor beszúrjuk az aktuális állapotot a listába egyes értékkel, ha már tartalmazta, akkor növeljük az állapothoz tartozó értéket egyel. Ezek után újra visszatérünk a végállapot vizsgálatához. A játéknak akkor van vége, amikor végállapotban vagyunk. Végállapot esetén megvizsgáljuk, hogy a játszma döntetlennel vagy valamelyik játékos győzelmével zárult, és a vizsgálat eredményét tudatjuk a felhasználóval. Ezt a vizsgálatot a *nyertA* és a *nyertB* metódusok segítségével végezzük el.

```
public boolean nyertA() {
        return (this.jatekos == 'B' && this.sakkbanVan);
}
```

A *nyertA* és *nyertB* metódusok az alapján térnek vissza igaz, vagy hamis értékkel, hogy melyik játékos van soron. Ha a ’B’ játékos van soron és nem tud szabályos lépést végrehajtani miközben sakkban van, akkor a játék nyertese egyértelműen az ’A’ játékos. Hasonlóképpen, amikor az ’A’ játékos van soron és sakkban van, akkor a ’B’ játékos nyerte a játszmát. Ha az aktuális játékos nincs sakkban, és mégis végállapotban vagyunk, akkor a játék döntetlennel ér véget. Döntetlennel ér véget a játék, ha:

- Az ütéseket és gyaloglépéseket nélkülöző lépések száma elérte az 50et.
- Ha a játék során a tábla aktuális állapota már két korábbi alkalommal is előfordult. Ezt hívják háromszori tükörképnek, és leggyakrabban örökös sakknál fordulhat elő.
- Ha egyik játékosnak sincs mattadó ereje
- Ha az aktuális játékos nincs sakkban, mégsem tud szabályos lépést végrehajtani

9. A variánsok megvalósítása

A sakkprogramunk fejlesztése során először az alapprogramot készítettük el. Ezután az inkrementális fejlesztés elvét vallva egyesével adtuk hozzá az egyes variánsokat. Némely sakkvariáns elkészítése könnyű feladatot jelentett, míg más variánsok megvalósítása komolyabb fejlesztési időt vett igénybe. A normál szabályokkal játszott sakk mellett további 7 új szabályokkal ellátott sakkvariánst implementáltunk, nem beszélve arról, hogy lehetőségünk van az egyes variánsok szabályait ötvözni, és egy teljesen új sakkvariánst így létrehozni.

9.1. All Queens

Az All Queens variáns megvalósítása jóval egyszerűbb volt a többihez képest. A játék kezdőállapotának létrehozása során a kezdő felállástól tértünk el. Az All Queens játék során ugyanis minden tiszt helyén királynő áll, mint ahogy azt a játék neve is mutatja. Az egyetlen további szabály, amely az előzőből következik, hogy a játék során nem lehet sáncolást végrehajtani.

```
if (all_queens) {
        for (int i = 1; i < 9; i++) {
                sakkTabla[1][i] = FEHER_KIRALYNO;
                sakkTabla[2][i] = FEHER_GYALOG;
                sakkTabla[7][i] = FEKETE_GYALOG;
                sakkTabla[8][i] = FEKETE_KIRALYNO;
        }
        sakkTabla[1][4] = FEHER_KIRALY;
        sakkTabla[8][4] = FEKETE_KIRALY;

        szabadSancolni = false;
}
```

9.2. Crazy House

A második megvalósított variánsunk a Crazy House. Ennél a variánsnál megengedett az ellenfél bábuinak használata. Ezért létrehoztunk egy új operátort, amely a *VisszaRak* nevet kapta. Ezen operátornak 3 paramétere van, az első kettő segítségével beállíthatjuk, hogy a pályán belül pontosan hova szeretnénk visszatenni a bábut, a harmadik paraméter pedig a bábut definiálja. Ezt a bábut a leütött bábuk közül fogjuk kivenni. Ezen operátor csak akkor áll a rendelkezésünkre, ha már van ütésünk. Ezért az operátorok kollekció ezen elemekkel való feltöltése az *alkalmaz* metódusba került. Két segéd kollekcióval dolgozunk. Az egyikbe az aktuális játékos *VisszaRak* operátorai kerülnek, a másikba pedig az ellenfél által alkalmazható *VisszaRak* operátorok. Első lépésként egy egymásba ágyazott dupla ciklus segítségével bejárjuk a sakktáblát. Közben minden egyes mezőre létrehozzuk az összes visszarakható bábut tartalmazó *VisszaRak* operátort. Aszerint kerülnek be az egyes *VisszaRak* operátorok az egyes segéd kollekciókba, hogy az aktuális bábu a soron következő játékosé vagy az aktuális játékosé. Az aktuális játékos bábuit visszarakó operátorok a következő játékos saját operátorai közé fog bekerülni a Crazy House variáns szabályai szerint. Sok üres mező esetén nagyon sok operátor jön létre. A sok operátor egyszerűbb kezelése érdekében egy bábu hiába szerepel többször is a leütött bábuk között, minden egyes mezőhöz csak egy az adott bábut visszarakó *VisszaRak* operátor készül.

```
for (int i = 1; i < 9; ++i) {
        for (int j = 1; j < 9; ++j) {
                if (uj.sakkTabla[i][j] == 0) {
                        for (Integer k : uj.LeutottBabuk) {
                                VisszaRak vr = new VisszaRak(i, j, k);
                                if (uj.jatekos == 'B') {
                                        if (k > 0 && !s_op1.contains(vr)) {
                                                s_op1.add(vr);
                                        } else if (k < 0 && !s_op2.contains(vr)) {
                                                s_op2.add(vr);
                                        }
                                }
                        }
                }
        }
}
```

Miután a segédkollekcióinkat feltöltöttük, végrehajtunk egy operátor ellenőrzést a kollekciókon, és a rostán fennmaradó operátorokat hozzáadjuk az *operatorok* és az *ellenfelOperatorok* kollekcióinkhoz.

A *VisszaRak* operátor alkalmazása során az új állapot leütött bábui közül törlünk egyet abból a bábuból, amelyet vissza szeretnénk tenni. Ezután már csak annyi a dolgunk, hogy az operátor paramétereinek megfelelő helyre visszateszünk egy már a saját színünkre konvertált a paraméter által meghatározott bábut.

Ezen kívül *VisszaRak* operátorokkal bővítettük ki az előfeltétel ellenőrző metódusainkban a klónállapotokat is, így ezen operátorokat is tudjuk ezentúl kezelni.

9.3. Dark Crazy House

A Dark Crazy House variáns a Crazy House és a Dark variáns keveréke. A Crazy House variánsból átveszi az a visszatehető bábuk szabályát, azaz az általunk leütött bábukat visszatehetjük a táblára saját bábunkként. A leütött bábut a sakktáblán bárhova elhelyezhetjük. A Dark variánsból pedig átveszi a láthatóság szabályát. Ebben az esetben csak annyit láthatunk a táblából amennyit a bábuink is látnak. Ez a bábuk által üthető mezőket jelenti. (Ha a gyalogunkkal nem tudunk lépni, és nem látjuk a gyalog előtti mezőt, akkro az azért lehet, mert ott az ellenfél bábuja található.) A mi programunkban úgy valósítottuk meg ezt, hogy létrehoztunk egy *lathatosag* nevű metódust, illetve egy *lathato* nevű mátrixot. A mátrix elemei 0-ásak akkor, ha az adott mező a táblán nem látható az aktuális játékos számára, ellenkező esetben az adott mezőre 1-es értéket kell beállítani. Ezt a beállítást a *lathatosag* metódus végzi el. Ez a metódus paraméterként egy állapotot kap, ezen állapotnak fogja beállítani a *lathato* mátrixát. Első lépésként bejárja a paraméterül kapott állapot operátorainak kollekcióját. Ha a bejárás során *Lep* operátort talál, akkor a láthatósági mátrixban láthatóvá teszi az operátor által megjelölt két mezőt, azaz ahonnan lépünk illetve ahova lépünk az operátor segítségével.

```
if (op instanceof Lep) {
        Lep l = (Lep) op;
                _allapot.lathato[l.getHovax()][l.getHovay()] = 1;
                _allapot.lathato[l.getHonnanx()][l.getHonnany()] = 1;
}
```

EnpassantLep operátor esetén lekéri az operátor paramétereiben megadott szín, irány és mezőinformációkat, és ez alapján állít be láthatóság információkat a *lathato* tömbbe. A *Lep*

```
if (enp_irany == BALROL) {
        if (enp_szin == FEHER && this.jatekos == 'A') {
                _allapot.lathato[enp_melyik_babuvalx][enp_melyik_babuvaly] = 1;
                _allapot.lathato[enp_melyik_babuvalx + 1][enp_melyik_babuvaly + 1] =
1;
        } else if (enp_szin == FEKETE && this.jatekos == 'B') {
                _allapot.lathato[enp_melyik_babuvalx][enp_melyik_babuvaly] = 1;
                _allapot.lathato[enp_melyik_babuvalx - 1][enp_melyik_babuvaly + 1] = 1;
        }
}
```

Ha *SancolLep* operátort tartalmaz a kollekciónk, akkor a sáncolásnak megfelelően azokat a mezőket teszi láthatóvá, amelyeken jelenleg áll a sáncoló bástyánk és királyunk, illetve azon mezőket, ahova majd a lépés végrehajtása után kerülnek.

Az operátorok között találkozhatunk *VisszaRak* operátorral is. Ebben az esetben minden üres mezőt látnunk kell majd a Dark Crazy House szabályai miatt, azaz egy dupla ciklus segítségével tesszük láthatóvá a mezőket.

Végül az egész mezőt bejárva megkeressük a saját bábuinkat, és láthatóvá tesszük az általuk látható mezőket. Miért van erre szükség? Azért, mert ha sakkban vagyunk, akkor csak azon operátorokat tartalmazza a kollekciónk, amelyek ezt a sakk helyzetet megszűntetik. Viszont mi az összes bábunkat szeretnénk a táblán látni, ez indokolja az előbbi tevékenységet.

9.4. Dark Crazy House 2

A Dark Crazy House 2 csak egy apró szabályváltoztatásban tér el a fentebb ismertetett Dark Crazy House variánstól. Ez a változtatás pedig az, hogy a táblára bábut csak a bábuink

által látható mezőkre rakhatunk vissza. A mi programunkban ez a *visszarak_barhova* változó segítéségével van megvalósítva, ebben az esetben ez a változó hamis értéket vesz fel. Ezen változó értékét több helyen is vizsgáljuk, így a *VisszaRak* operátorok készítése során is. Csak olyan mezőre engedjük meg a bábuk visszarakását, ahol a *lathatosag* mátrix értéke 1-es, azaz az adott mező a bábuink számára látható. A *visszarak_barhova* változó értékét ezután a *lathatosag* metódusban vizsgáljuk, ugyanis ha ez hamis, azaz Dark Crazy House 2 variánst játszunk, akkor az üres mezők láthatósága a visszarakás ellenére is nem látható marad.

9.5. Lao Tzu

A Lao Tzu a Dark Crazy House 2 variánson alapul. Ugyanazon szabályok érvényesek rá, mint a Dark Crazy House 2-re egy kivétellel. Ez pedig a véletlenszerűen felrakott alapállás. Mi ezt a *tablaVeletlenszeruFeltoltese* metódus segítségével valósítjuk meg. A feltöltés során nyilvántartunk egy 6 elemű tömböt, amely azt tartalmazza, hogy melyik bábuból mennyit kell még felraknunk a táblára. Ezután bejárjuk a tábla első két sorát, és minden egyes mezőre generálunk egy véletlen számot 1-től 6ig. Ezután megvizsgáljuk, hogy az adott szám által meghatározott tömbelem által tartalmazott szám nagyobb-e mint nulla. Ha igen, akkor a megfelelő bábut felrakhatjuk a táblára. Különben új számot generálunk, és újra megtesszük a fenti vizsgálatot. Amint végigértünk mindkét soron, a tábla első két sora fel lesz töltve az egyik játékos bábuival. A másik játékos bábuinak feltöltésével hasonlóképpen járunk el. A Lao Tzu variáns futása ezen a feltöltésen kívül megegyezik a Dark Crazy House 2-ével.

9.6. Franciasakk

A franciasakkban vagy ahogy külföldön jobban ismerik a suicide chess-ben a célunk a bábuk elvesztése. Ennél fogva a játékban nincs sakkhelyzet, sem sáncolás. Utóbbit egyszerűen valósítjuk meg, a *szabadSancolni* változó értékét hamisra állítjuk. A további szabályokat egyéb változók igaz-hamis állításával oldjuk meg. A kotelezo_utes változó legtöbb szerepe az előfeltétel ellenőrző metódusunkban van. Ha igaz a változó értéke, akkor értelemszerűen minden menet közbeni ütés esetén az előfeltétel ellenőrzés igazzal tér vissza.

Az operátorok alkalmazása során is szerepe van a kötelező ütés szabálynak. Mielőtt a következő játékos operátorait feltöltenénk, beállítjuk az új állapotba az *az_allapotban_kotelezo_utni* változó értékét. Ehhez egy segéd kollekcióba betöltjük az összes alkalmazható operátort ellenőrzés nélkül. Ezután meghívjuk az állapotra a *kotelezo_utes* metódust. Ez a metódus majd beállítja a paraméterül kapott állapotban a paraméterül kapott operátorok alapján az *az_allapotban_kotelezo_utni* változó értékét. A metódus a futása során bejárja a paraméterül kapott operátor kollekciót, és amint legalább egyet talál, amelynél ütés hajtódna végre, akkor a kötelező ütést nyilvántartó változó értékét igazra állítja. Ebben az esetben operátorok feltöltése és ellenőrzése során az ellenőrző metódus csak olyan operátorokat fog engedélyezni, amelyekkel ütést hajtunk végre.

```
for (Operator seged : _operatorok) {
        if (seged instanceof Lep) {
                Lep l = (Lep) seged;
                if (_allapot.sakkTabla[l.getHovax()][l.getHovay()] != 0) {
                        _allapot.az_allapotban_kotelezo_utni = true;
                        return;
                }
        } else if (seged instanceof EnPassantLep) {
                _allapot.az_allapotban_kotelezo_utni = true;
                return;
        }
}
_allapot.az_allapotban_kotelezo_utni = false;
```

Mivel a franciasakkban nem lehet sakkot adni, így ebben a játéktípusban felesleges az *elofeltetel* metódusban definiált klónállapotok kezelése.

Franciasakk esetén módosítanunk kell a végállapot ellenőrzésen is. Akkor vagyunk végállapotban, ha az aktuális játékos nem tud szabályosan lépni.

```
if (suicide) {
        if (!operatorok.isEmpty()) {
                return false;
        }
}
```

A franciasakkban az állapotok hasznosságát is más képlettel számolja ki, mint normál játék esetén. Ebben a variánsban úgy pontozza az egyes állásokat, hogy minél kevesebb

bábuja van az aktuális játékosnak, annál jobb értékelést kap. Ez fordítva is igaz, azaz úgy próbál lépni, hogy az ellenfélnek minél több bábuja maradjon.

9.7. Benedict

A Benedict variáns legnagyobb újítása, hogy nincs benne se ütés, se sakk. Emiatt a programunkban mi is létrehoztunk egy *nincs_sakk* illetve *nincs_utes* változót, amellyel a már megírt részeket benedict variáns kompatibilissé tudjuk tenni. Mivel a játékban nincs ütés, ezért a menet közbeni ütés, azaz enpassant lépés sem valósítható meg. Továbbá egyszerűsíti a sáncoló lépések vizsgálatát, hogy mivel nincs sakk, ezért a sáncolás vizsgálata során kihagyható azon ellenőrzés, hogy sakkban tartott mezőn nem haladhatunk át, továbbá olyan mezőre is sáncolhatunk, amely után a királyunk sakkban lesz. A benedict variánsban a végállapot eldöntése is egészen leegyszerűsödik. Ha az aktuális játékos nem tud lépni, vagy nincs már királya, akkor végállapotban vagyunk.

```
if (benedict) {
        if (operatorok.isEmpty()) {
            return true;
        }
        for (int i = 1; i < 9; i++) {
            for (int j = 1; j < 9; j++) {
                if (this.jatekos == 'A' && sakkTabla[i][j] == FEHER_KIRALY) {
                    return false;
                }
                if (this.jatekos == 'B' && sakkTabla[i][j] == FEKETE_KIRALY) {
                    return false;
                }
            }
        }
        return true;
```

Ha a játékosnak nincs királya, akkor azt az ellenfél átszínezte, így az ellenfél játékos győzött. Ha viszont végállapotban vagyunk, de a királyunk még megvan, de mégsem tudunk szabályos lépést végrehajtani, akkor a játék döntetlennel ér véget.

Mivel ebben a variánsban nincs sakk, ezért az előfeltételeknél használt klón állapot elkészítése is felesleges ugyanúgy, mint a franciasakk esetében.

A benedict sakk esetében már csupán egy feladatunk maradt, ez pedig a támadott bábuk átszínezése. Ehhez az új állapot készítése során egy segédkollekciót használunk fel, amelyet bejárunk, és mindegy operátorát megvizsgáljuk. Ha az aktuális operátor *honnan* mezője a mi általunk támadott *hova* mező, és még nem színeztük át ezt a bábut, akkor átszínezzük.

```
for (Operator seged : s_op1) {
        if (seged instanceof Lep) {
                Lep ll = (Lep) seged;
                        if (ll.getHonnanx() == hovax && ll.getHonnany() == hovay) {
                                if((uj.sakkTabla[ll.getHovax()][ll.getHovay()] > 0 &&
                                        this.jatekos == 'B') ||
                                (uj.sakkTabla[ll.getHovax()][ll.getHovay()] < 0 &&
                                        this.jatekos == 'A'))
                                   uj.sakkTabla[ll.getHovax()][ll.getHovay()] *= -1;
                        }
        }
}
```

Ezután az *alkalmazas* metódus fut tovább a normál sakkban meghatározottak szerint.

9.8. Benedict 960

A Benedict 960 a hagyományos Benedict sakk szabályait követi azzal a kivétellel, hogy az alapállása nem a hagyományos sakk szerint történik, hanem véletlenszerűen tesszük fel a táblát. Ezzel a lépéssel eltűntethető a világos játékos játék elején szerzett előnye. A véletlenszerű táblafelrakás a Lao Tzu variánsban meghatározottak szerint tesszük meg, a *tablaVeletlenszeruFeltoltese* metódus segítségével.

10. Az általunk alkalmazott heurisztika

A programunkban talán ez az a rész, amelyen a legtöbbet lehetne fejleszteni. Ha belegondolunk, ez a fejlesztés (fejlődés) talán sosem érhetne véget, mindig találhatnánk új pontozási elveket, mechanizmusokat, amelyeket folyamatosan tesztelni lehetne, összevetni a gép által korábban megtett lépésekkel. Az alapprogramunkban ez a rész készült el utoljára, és ez a rész talán az, amely – ha sokkal több idő lenne egy szakdolgozatot megírni – a legtöbb idejébe kerülne egy fejlesztőnek a sakkprogram fejlesztése során.

A hasznosság annál jobb, minél közelebb van nullához. Ezért a mi hasznosságfüggvényünk 5000-ről indul, majd minden egyes jutalompontot, illetve a játékos által birtokolt bábu értékét ebből vonja le. Az esetleges büntetőpontokat, mint a tábla centrumának ellenfél által való birtoklása ehhez a hasznosságértékhez adja hozzá.

A mi heurisztikánk egyszerű lépéseken alapul. Megvizsgáljuk az egész sakktáblát, minden egyes mezőt egyesével, a rajta található bábukkal egyetemben. Minden bábunak van egy alapértelmezett értéke, amely segít a számítógép számára a megfelelő lépést megtenni. Ezen értékek a következők:

```
public static final int GYALOG_ERTEK = 10;
public static final int FUTO_ERTEK = 30;
public static final int HUSZAR_ERTEK = 30;
public static final int BASTYA_ERTEK = 50;
public static final int KIRALYNO_ERTEK = 100;
public static final int KIRALY_ERTEK = 2000;
```

Ezen alapértékeket szorozzuk fel minden bábu esetén több értékkel. Az első ilyen érték az oszloptól függ:

- Ha a vizsgált bábu az ’a’ vagy a ’h’ oszlopban található, akkor a szorzónk 1.0 marad.
- Ha a vizsgált bábu az ’b’ vagy a ’g’ oszlopban található, akkor a szorzónk 1.2-re növekszik.
- Ha a vizsgált bábu az ’c’ vagy a ’f’ oszlopban található, akkor a szorzónk 1.5-re növekszik.

- Ha a vizsgált bábu az ’d’ vagy a ’e’ oszlopban található, akkor a szorzónk 1.9-re növekszik.

Az egyes bábucsoportok esetén különbözőképpen változik ez a szorzó. Ha az aktuális bábu egy centrumban lévő gyalog és a játék megnyitás szakaszában tartunk, akkor a szorzót tovább növeljük az 1.3-szorosára. Ezután a gyalog értékét, ami esetünkben 10, megszorozzuk az aktuálisan kiszámolt szorzóval, az eredményt pedig hozzáadjuk a hasznosságot nyilvántartó *hasznossag* változó értékéhez. Ezután haladhatunk a következő mezőre.

Egy példa: Ha a fehér bábukkal játszunk, és a heurisztikát számoló metódus egy fehér gyalogot talál az f4-es mezőn, akkor ennek a gyalognak az értéke: 10 * 1.5 * 1.3, azaz 19.5 lesz szemben egy szélen álló gyaloggal szemben, amely továbbra is 10-et fog érni.

Így érjük el, hogy a játék elején a gép próbálja meg elfoglalni gyalog bábukkal a tábla centrumát. A centrumot, külső centrumot illetve az egyéb mezőket egyébként egy *centrum* nevű segédmátrixban tartjuk nyílván. Mivel a sakktáblánk indexelése megegyezik a centrum mátrix indexelésével, ezért nagyon könnyű meghatározni, hogy egy bábu a centrumban van-e vagy sem.

```
switch (_allapot.sakkTabla[i][j]) {
    case FEHER_GYALOG: {
        if (szakasz == 0 && (centrum[i][j] * _allapot.sakkTabla[i][j] == -1)) {
            szorzo *= 1.3;
        }
        hasznossag -= GYALOG_ERTEK * szorzo;
        break;
    }
}
```

A többi bábu esetén hasonló a helyzet. Huszár, futó, bástya avagy királynő esetén annyiban térünk el a fent vázolt mechanizmustól, hogy ezen bábuk 1.1-es szorzót kapnak, ha a játék megnyitás szakaszánál a külső centrumban helyezkednek el, vagy ha a közép- illetve végjátéknál a belső centrumban helyezkednek el. Ezzel érjük el azt is, hogy megnyitáskor védjék a centrumot, illetve a játék más szakaszaiban pedig próbáljanak arra törekedni, hogy foglalják el azt. A fent említett oszlopok miatti szorzás az említett bábukra is érvényes, ugyanis egy belső pozícióból sokkal jobb lehetőségei vannak egy királynőnek, de még egy huszárnak is, mintha a szélső pozíciókon lennének.

```
case FEKETE_FUTO: {
        if ((szakasz == 0 && (centrum[i][j] * _allapot.sakkTabla[i][j] == -6)) ||
        ((szakasz == 1 || szakasz == 2) &&
        (Math.abs(centrum[i][j] * _allapot.sakkTabla[i][j]) == 3))) {
                szorzo *= 1.1;
        }
        hasznossag += FUTO_ERTEK * szorzo;
        break;
}
```

A játék szakaszának kiértékelése is egyszerűen zajlik. Először összeszámoljuk az egyes játékosok bábuinak számát. Ezután ezen eredmény tudatában döntünk:

- Ha az aktuális játékos 7 vagy annál kevesebb bábuval rendelkezik, akkor a játék a végjáték szakaszba ért.
- Ha a játékosnak több bábuja van, mit 7, sáncolt már, illetve elmozdította már a futóját és huszárját a kezdőhelyéről, akkor a játék középjáték szakaszban van.
- Ha a fenti feltételek nem teljesülnek, akkor a játék a megnyitás szakaszban jár.

A játék szakaszán és a bábuk elhelyezkedésén túl még további eseményeket is pontoz a heurisztikus metódus. Ha az aktuális játékos már sáncolt a játék folyamán, akkor ezt további 90 ponttal jutalmazza a heurisztika. Ezek után egy *centrum* nevű metódus segítségével eldöntjük, hogy mely játékosok bábuja van centrumban. Ha az aktuális játékos bábui tartózkodnak ott, akkor ezt bábunként 8 ponttal jutalmazza. Ellenkező esetben 8 pontot von le a heurisztika az állás hasznosságából.

11. Interakció a felhasználóval

A felhasználóval való kommunikációnkat a grafikus kinézet mögött a *lepestBeker* és a *cseretBeker* metódusok valósítják meg.

11.1. Lépés bekérése a felhasználótól

Amint a programunk futása odáig jutott, hogy az emberi játékos van soron, meghívódik a *lepestBeker* metódusunk. A metódus a futásának első lépéseként meghívja az *alkalmazhatoOperatorokKiiratasa* metódust.

11.1.1. Az *alkalmazhatoOperatorokKiiratasa* metódus

Ez a metódus a nevéből adódóan a felhasználó számára láthatóvá teszi, milyen szabályos lépésekkel rendelkezik az adott állásban. A metódus egy *StringBuffer*-rel dolgozik, ebbe gyűjtjük össze az alkalmazható operátorok szöveges reprezentációját. A metódusunk az aktuális állapot *operatorok* kollekcióját járja be, és minden egyes operátorra végrehajtja a fentebb leírt lépést. Miután kész a *StringBuffer*-ünk, a tartalmát kiíratjuk a standard kimenetre, és a játéktábla súgó részébe is.

11.1.2 A *lepestBeker* és a *cseretBeker* metódusok

A *lepestBeker* metódus vár, amíg a játékosunk nem lép a felhasználói felületen, ezután értékül átadja a felületen megadott lépést. Rossz lépést nem kaphat meg, mert átadjuk az *asztal* osztálynak az aktuálisan alkalmazható összes operátort, és csak akkor ad engedélyt a *lepesBeker* folytatására, ha a megadott lépést tartalmazza az átadott kollekció.

A *cseretBeker* ugyanígy működik, ez a metódus akkor hívódik meg, ha egy gyalog beér az ellenfél alapsorára.

11.2 Lépés megadása

A lépések megadásához két mezőt hoztunk létre. A hagyományos lépésnél az első mezőbe lehet beírni zárójelezett formában, hogy honnan, a második mezőbe pedig azt, hogy hova kívánunk lépni. Azt nem kell megadni, hogy milyen típusú bábuval kívánunk lépni, ezt a számítógép meghatározza helyettünk. Sáncoláshoz az első mezőbe a „sáncolás” szót kell írni, a másodikba pedig, hogy melyik oldalon kívánunk sáncolni. Ez lehet a „rövid” illetve a „hosszú” oldal. Ha a lépésünk a Crazy House játékokban lehetséges visszarakás, akkor az első mezőbe kell megadni azt, hogy milyen bábut akarunk visszarakni, a másodikban pedig azt, hogy hova szeretnénk visszarakni. A *csereBeker* számára átadandó lépésnél nem kell megadni a második mezőben semmit, csak az elsőben azt, hogy milyen bábut akarunk a helyére tenni.

Példák:

	Első mező:	Második mező:
Normál lépés:	(a,5)	(a,6)
Sáncolás:	sáncolás	rövid
Visszarakás:	királynő	(a,6)
Csere:	királynő	

12. Köszönetnyilvánítás

Szeretnénk külön köszönetet nyilvánítani a témavezetőnknek, dr. Nagy Benedek tanár úrnak, aki a feladataink kiosztása mellett segítségünkre volt mindenben bármikor is kerestük fel. Szeretnénk köszönetet mondani Kósa Márknak, Jeszenszky Péternek illetve dr. Várterész Magdolnának, az ő mesterséges intelligencia kurzusaik nélkül nem sikerült volna elsajátítani a mesterséges intelligencia és azon belül a kétszemélyes játékok alapjait. Ezen kívül nagy szerepet játszottak abban is, hogy mindketten megkedveltük a számítástechnika eme csodálatos ágát. Továbbá köszönetet szeretnénk mondani a sakkozó múlttal rendelkező Szabó Károlynak, aki nagyon sokat segített a megfelelő heurisztika megtalálásában, illetve annak programba való átültetésében. Végül, de nem utolsósorban külön köszönetet érdemelnek a szüleink, barátaink, barátnőink, akik egész évben támogattak, bátorítottak bennünket, és segítettek a külső zavaró tényezők minimális szintre történő leredukálásában.

13. Összegzés

A sakkprogramunk a variánsával együtt 2 félév alatt elkészült, működőképes. Sajnos nem sikerült minden eltervezett funkciót megvalósítani időhiány miatt. Rengeteg érdekes variánsra derítettünk fényt a szakirodalom és az internet segítségével, melyek közül próbáltuk a legérdekesebbeket megvalósítani. Találkoztunk olyan variánssal is, amely az új szabályok, bábuk terén oly mértékű komplexitással rendelkezett, ami miatt - bár mindenképpen meg szerettünk volna valósítani - a már elkészült programmal való inkompatibilitása kénytelenek voltunk elvetni a megvalósítását. A 3. fejezetben ismertetett variánsok felölelik az elkészíteni tervezett variációkat. Végül a hagyományos sakk mellett annak további 8-féle variációját valósítottuk meg. Ezen variációk között vannak hasonlóak, és nagymértékben különbözőek is. A program fejlesztése során a hagyományos sakk elkészítése – amiatt, hogy egyetem mellett készült – közel 3-4 hónapig tartott. Ezután kezdtük el a variánsok komponensenként való elkészítését. Ebből természetesen az következett, hogy romlott a program architekturális felépítése, és viszonylagosan lelassult a működése.

A program fejlesztése során megbizonyosodhattunk arról, hogy a sakk az egyik legbonyolultabb és legkiszámíthatatlanabb játék. A szabályok sokasága és esetlegesen bonyolultsága, illetve a heurisztika kiszámításának nehézsége a bizonyíték erre. Egy komolyabb heurisztika elméleti kidolgozása, majd implementálása akár egy egész féléves feladat is lehetett volna. Ezzel szemben a mi heurisztikánk viszonylag sok elemet tartalmaz, illetve kísér figyelemmel, emiatt nagyon nehezen tesztelhető lett. Utóbbinak köszönhető a gépi ellenfél furcsa, néha túlontúl támadó jellegű stratégiája. A heurisztika tesztelése mellett az egyes komponensek tesztelésére próbáltunk minél több időt fordítani. A komponensek száma, és a variációs lehetőségek miatt az egyéb játék lehetősége szinte teszteletlenül került a programba.

A programunkba a grafikus felület került be utoljára. Erre azért volt mindenképpen szükség, hogy ne vesszen el a játékélmény a tábla mátrixszal való reprezentálása miatt. A fejlesztés során a lehetőségek teszteléséhez mi ezt a fajta kimenetet használtuk, és számunkra teljesen játszható és átlátható játékot eredményezett. De gondolnunk kellett az átlagfelhasználókra is, akik nem egész számokkal képzelik el a sakktáblát.

A program futtatásához Java futtató környezet (JRE) szükséges. Ezt legegyszerűbben az alábbi oldalról érhetjük el: http://java.sun.com/

A program fejlesztése során belekóstolhattunk egy az egyetemi programokhoz képest nagyobb mértékű projekt fejlesztésébe, melynek során sokat tanulhattunk a határidők betartásáról, a feladatok megfelelő kiosztásának fontosságáról illetve a hatékony csapatmunka megvalósításának módszereiről.

Reméljük az olvasónak fel tudtuk kelteni az érdeklődését a sakkprogramok és főleg azok egyéb variánsai iránt. Jó szórakozást és jó játékot szeretnénk kívánni mindenkinek!

14. Irodalomjegyzék

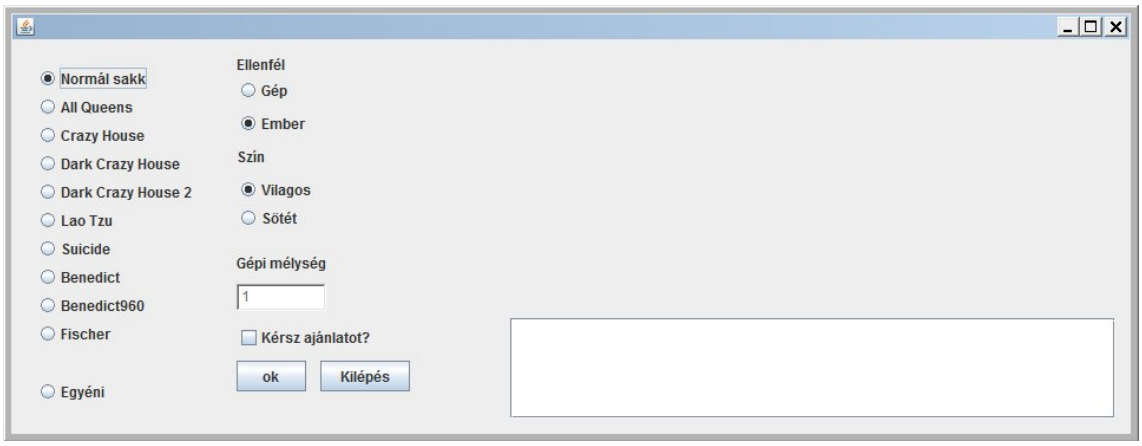
[1] http://en.wikipedia.org/wiki/Chess

[2] http://hu.wikipedia.org/wiki/Sakkv%C3%A1ltozatok

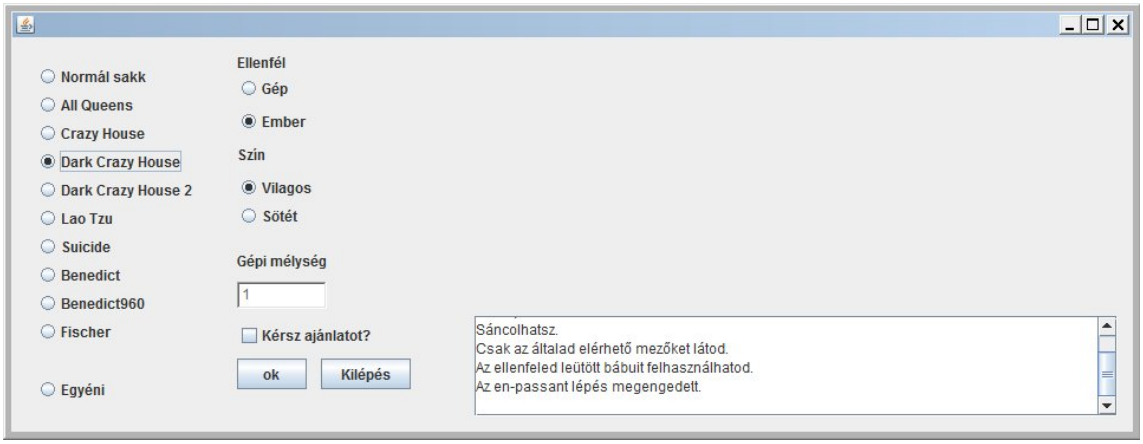
[3] http://barlanglako.com/sakk/leirasok.php

[4] Fekete István, Gregorics Tibor, Nagy Sára – Bevezetés a mesterséges intelligenciába.

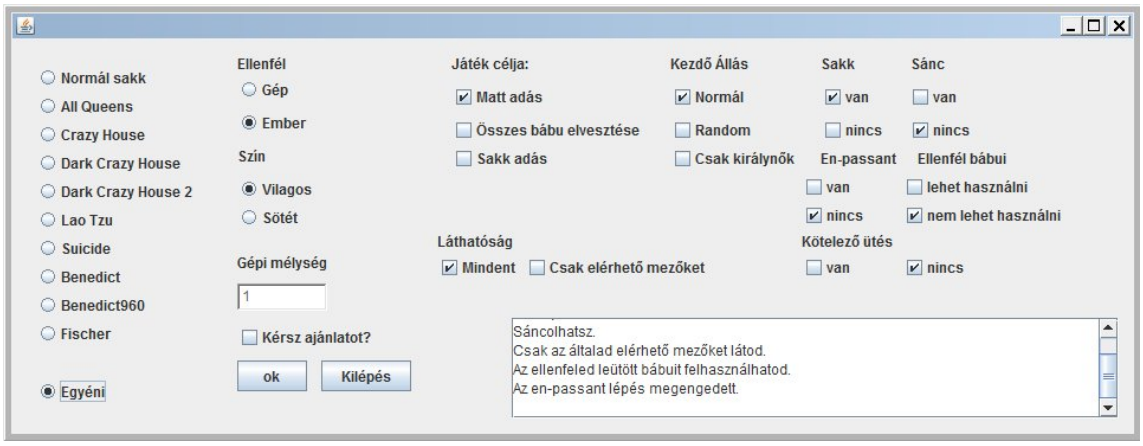
15. Függelék (képek)



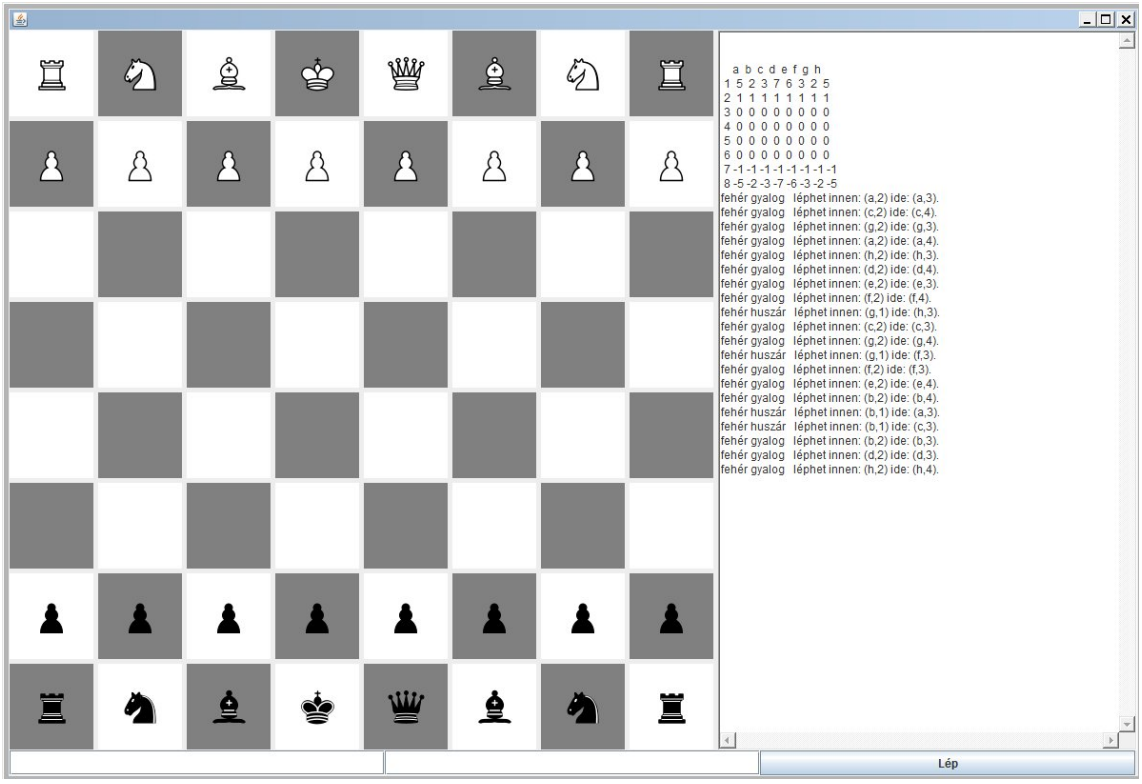
1.1. Kép a kezdőképernyőről.



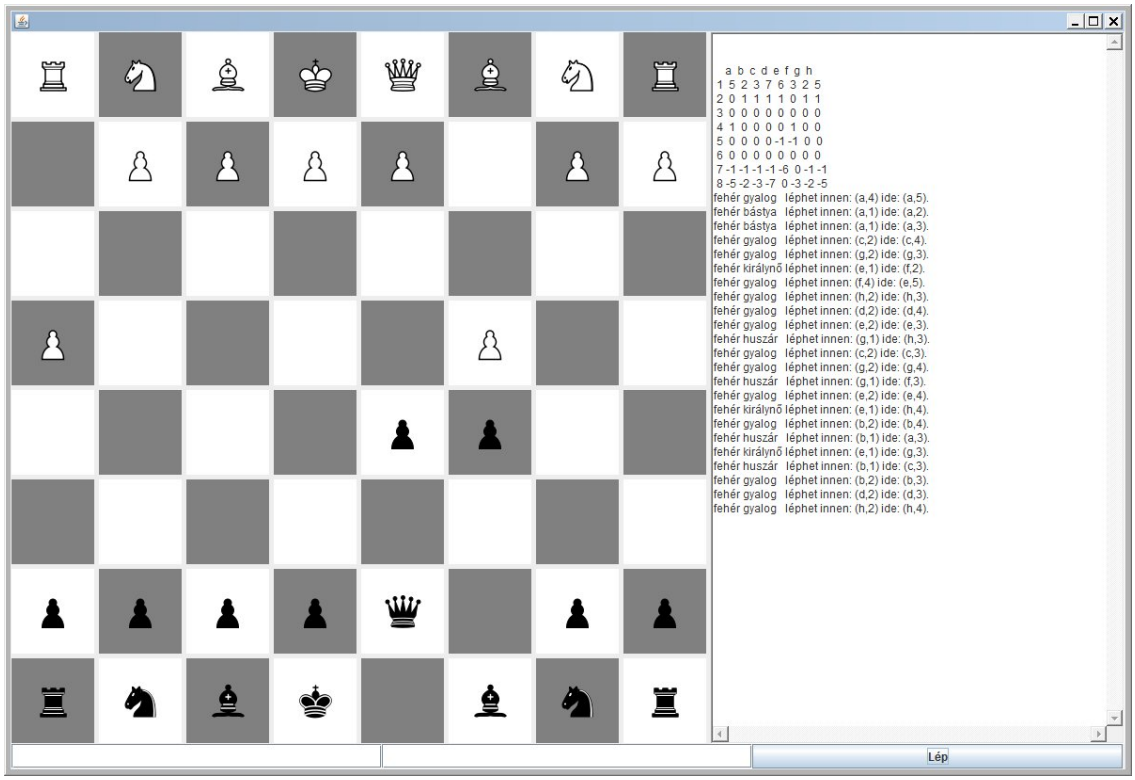
1.2 Kép a Dark Crazy House szabályairól.



1.3 Kép az egyéni beállítási lehetőségekről.



1.4 Kép a hagyományos sakk kezdőállapotáról.



1.5 Véletlenszerű kép a játék futásából.