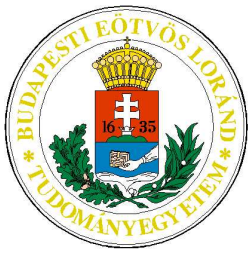


EÖTVÖS LORÁND TUDOMÁNYEGYETEM
INFORMATIKAI KAR
KOMPUTERALGEBRA TANSZÉK

Sakkprogram

szakdolgozat

Juhász Péter Károly

Témavezető:
Iványi Antal Miklós

Budapest, 2006.

Tartalomjegyzék

Előszó

A dolgozat egy Java nyelven írt sakkprogram dokumentációját tartalmazza, amit az ELTE programozó matematikusi szak szakdolgozataként írtam.

Egy pár szó a témaválasztásról. Miért adtam a fejemet sakkprogram írására? Ennek az az oka, hogy szeretem a játékokat, főleg az olyanokat, amelyek meggondolkoztatnak, aktív go-játékos vagyok. Mivel azonban a go-program megírása sokkal bonyolultabb feladat, mint a sakkprogram, és nem lehet ráhúzni egy a „Bevezetés a mesterséges intelligenciába" című tárgyból tanult algoritmust sem, ezért inkább a már sokak által tanulmányozott probléma, a sakkprogramozás felé fordítottam a fejemet. Másrészt, mert a mesterséges intelligencia is nagyon érdekel és az erős mesterséges intelligencia elkötelezett hívei közé sorolom magamat, továbbá már rég szerettem volna valami olyan programot írni, ami tudásában fölém emelkedik (bár csekélyke sakktudásomat tekintve ez nem nagy feladat). Egy szó, mint száz, már rég terveztem egy ilyen program megírását.

A sakk, mint mesterséges intelligencia probléma ideális arra, hogy programmal oldjuk meg. Nagy szerepe van ebben annak, hogy kicsi a tábla, csak 8x8-as[1], és a speciális lépésszabályok miatt egy-egy állásban kevés lehetséges lépés létezik (állítólag átlagosan 35 [MIKÖNYV]). Így a probléma tökéletes arra, hogy nyers erővel[2] keressük az ideális lépést, mint ezt láthattuk a Deep Blue esetében is, ami egy viszonylag primitív kiértékelőfüggvény használatával (majd, hogy nem csak az anyagi előnyt nézte) és irdatlanul sok számolással, 13 lépést[3] előregondolkodva megverte a világbajnokot.

A dolgozat első része a *Felhasználói dokumentáció,* ami főleg a program kezelését mutatja be, de kitér a sakk játékszabályaira is, ezen belül arra is, hogy mit, hogyan és miért alkalmaz, illetve nem alkalmaz ezek közül a program, továbbá a játékstílusáról is esik pár szó, és az emögött rejlő okokról is. Ez azért van, hogy a felhasználó egy kicsit beleláthasson abba, hogy mit és miért lép a program. A második rész a *Fejlesztői dokumentáció,* ami magában foglalja a programban felhasznált algoritmusokat, ezeket ábrákkal színesíti, valamint részletesen tárgyalja a program szerkezetét. A harmadik rész a *Tesztelés,* ami bemutatja a programozás közben felbukkanó hibákat és megoldásokat, a tesztelés módjait, a tapasztalatokat. Végül híres partikat, feladványokat vesz sorra, amikben a kulcslépést kerestettem meg a programmal. A negyedik, egyben utolsó rész, a *Továbbfejlesztési lehetőségek* címre hallgat. Azon módszereket mutatja be, amikkel növelni lehetne a program játékerejét.

A program megvalósította az összes általam kitűzött célt, azaz barátságos felhasználói felületű, több platformon is futtatható, és annak ellenére, hogy nem nagyon erős, de gyengébbek számára élvezhető erősségű sakkprogram született. Az elég primitív neuronhálóval – csak egy szintű (perceptron) –, eredményes tanulást mutatott a tesztelések alatt. A programozás folyamán sikerült szem előtt tartani az objektumelvűséget, amely lehetővé teszi a későbbi kiegészítéseket és fejlesztéseket.

[1] A go-tábla 19x19-es nagyságú.

[2] Mindennemű heurisztika mellőzésével, csak a nagy számítási kapacitásra támaszkodva, itt minél mélyebben kipróbálni az összes lépésvariációt, és úgy dönteni.

[3] A lépés továbbiakban mindig fél lépést jelent, angolul *ply*, ez bevett szokás ezen témakörben, mivel a program egy lépése az, hogy eggyel mélyíti a fát, ami a valós világban csak egy fél lépésnek felel meg.

1. fejezet

Felhasználói dokumentáció

1. A sakk játékszabályai

A következőkben leírom azon részeket a sakk nemzetközi játékszabályzatából, amelyek feltétlenül szükségesek ahhoz, hogy a programot használni tudjuk. Ha az olvasó tud már sakkozni, akkor nyugodtan átugorhatja ezt a részt.

1.1. A sakkozás meghatározása

A sakkjátékot egy négyzet alakú táblán (sakktábla) két ellenfél játssza a bábok mozgatásával.

1.2. A sakktábla és beosztása

1. A sakktábla 64 egyenlő nagyságú, négyzet alakú, váltakozva világos (fehér) és sötét (fekete) mezőből áll.

2. A sakktáblát úgy kell a játékosok közé elhelyezni, hogy a jobb kezük felőli sarokmező világos legyen.

3. A nyolc egymás feletti mezőt, amelyek a sakktáblának az egyik játékos felé néző szélétől az ellenfél oldalán lévő széléig függőlegesen futnak „vonalaknak” nevezzük.

4. A tábla egyik oldalától a másikig vízszintesen egymás mellett futó nyolc mezőt „sornak” nevezzük.

5. Az egymást sarkaikkal érintő, rézsútosan egymásba kapcsolódó mezőket „átlónak” nevezzük.

1.3. A bábok és elhelyezésük

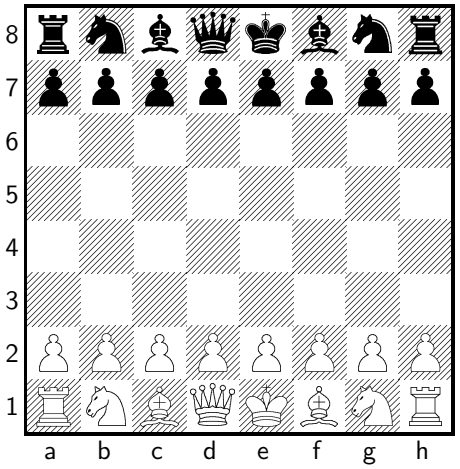
A játszma kezdetén az egyik játékos 16 világos (fehér), a másik 16 sötét (fekete) bábbal rendelkezik.

A bábok és ábrázolásuk az 1.1. ábrán látható.

egy király	♔
egy vezér	♕
két bástya	♖
két huszár	♘
két futó	♗
nyolc gyalog	♙

1.1. táblázat: A bábok és ábrázolásuk

A bábok alapállása a táblán a 1.1. ábrán látható.



1.1. ábra: A bábok alapállása a sakktáblán

1.4. A játék menete

Mindkét játékos felváltva lép oly módon, hogy minden alkalommal egy-egy lépést tesznek meg. A játszmát a világos bábokkal lévő játékos kezdi.

1.5. A lépés általános fogalma

1. Lépésnek nevezzük (a sáncolás kivételével) egy báb áthelyezését az egyik mezőről egy olyanra, amely vagy szabad, vagy az ellenfél bábja foglalja el.

2. A sáncolást végző bástyán és a huszáron kívül más báb nem ugorhat át olyan mezőt, amelyet már egy másik báb foglal el.

3. Ha valamely bábot ellenséges báb által elfoglalt mezőre teszünk, úgy ezzel a lépéssel az ott lévő bábot levesszük („leütjük"). Az ellenséges bábot azonnal le kell venni a tábláról. (Az „en passant" ütésre vonatkozólag lásd később.)

1.6. A bábok mozgása

A király A király saját mezőjéről (a sáncolás kivételével) bármely szomszédos mezőre léphet, amely nem áll ellenséges báb ütőkörében.

A sáncolás a király és a bástya együttes lépése. Ez egyetlen lépésnek, mégpedig király lépésnek számít, és a következőképpen történik: először a király elhagyja eredeti helyét, hogy ugyanabban a sorban a legközelebbi azonos színű mezőre lépjen, majd az a bástya, amely felé a király lépett, a királyt átugorva arra mezőre kerül, amelyet a király az imént átlépett.

Ha a király már lépett, a továbbiakban már egyik oldalra sem sáncolhat.

Éppen úgy nem lehet sáncolni az olyan bástyával sem, amelyik már lépett.

A sáncolás átmenetileg nem lehetséges:

1. ha a király eredeti helye, vagy az a mező, amelyen keresztül lépne, vagy az, amelyre lépni akar, ellenséges báb ütőkörében van;
2. ha báb vagy bábok állnak a király és a bástya között, amely felé a király lépni akar.

A vezér A vezér azon a soron, vonalon és átlókon mozoghat, amelyek találkozási mezején áll.

A bástya A bástya azon a soron és vonalon mozoghat, amelyek kereszteződésében áll.

A futó A futó azokon az átlókon mozoghat, amelyek keresztezésében áll.

A huszár A huszár mozgása két különböző lépésből tevődik össze. Egy lépést tesz a vonal vagy sor közvetlenül határos mezejére, majd utána a kiindulási mezőtől egyidejűleg távolodva egy lépést az átló közvetlenül határos mezejére.

A gyalog A gyalog csakis előre léphet.

1. Az ütés esetét kivéve, eredeti helyéről saját vonalán egy vagy két lépést tehet előre szabad mezőre, a továbbiakban pedig egy-egy lépést szintén szabad mezőre. Ütés esetén előrelép olyan mezőre, amely a sajátját átlósan érinti.

2. A gyalog, amelynek ütőkörén egy ellenséges gyalog kettős lépéssel átlépett, ütheti az ellenséges gyalogot úgy, mintha ez a gyalog csak egy lépést tett volna meg. Ez az ütés azonban csak a kettős gyaloglépésre közvetlen válaszlépésben lehetséges. Ezt az ütést hívjuk „en passant" (menet közbeni) ütésnek.
3. Minden olyan gyalogot, amely az utolsó sort elérte, azonnal át kell változtatni – mint ugyanezen lépés velejáróját – vezérré, bástyává, huszárrá, vagy futóvá, a játékos választása szerint, függetlenül a táblán lévő bábok számától. Ezt az eljárást nevezzük „átváltozásnak". Az átváltozott báb hatóereje azonnal életbe lép.

1.7. A sakkadás

1. A király sakkban áll, ha az általa elfoglalt mezőt ellenséges báb támadja. Ekkor mondják, hogy sakkot ad a királynak.
2. A sakkadást a közvetlenül rá következő lépéssel védeni kell. Ha a sakkadást védeni nem lehet (ellépni a sakkból vagy leütni a támadó bábot), úgy ezt mattnak nevezzük.
3. Az a báb, amely a saját királyának adott sakkot kivédi, egyidejűleg az ellenséges királynak ugyanakkor sakkot adhat.

1.8. A nyert játszma

1. A játszmát az a játékos nyeri, aki az ellenfél királyát mattolja.
2. Nyertes az a játékos, akinek ellenfele feladja a játszmát.

1.9. A döntetlen játszma

Eldöntetlen a játszma:

1. ha a lépésre következő játékos királya nem áll sakkban, de ez a játékos sem a királyával, sem más bábjával szabályos lépést nem tud tenni; erre az állásra mondják, hogy „patt";
2. a két játékos közös megegyezése alapján;
3. a játékosos egyikének kívánságára, ha ugyanaz az állás háromszor ismétlődik és ugyanaz a fél következik lépésre. Az állás akkor tekinthető azonosnak, ha ugyanazon fajtájú és színű bábok ismét ugyanazokon a mezőkön állnak, és ha a bábok lépésre vonatkozó lehetőségeik ugyancsak azonosak. (Ez a *háromszori állásismétléses döntetlen.*)

   Döntetlent kizárólag azon játékosnak van joga igényelni:

(a) akinek módjában áll olyan lépést tenni, amellyel az állás újbóli ismétlődését idézi elő, feltéve, hogy ezen lépés megtételére irányuló szándékát előzetesen bejelenti,

(b) aki soron következik olyan lépésre válaszolni, amellyel az ismételt állás újból létrejött.

ha a játékos már lépett anélkül, hogy döntetlen igényét a fentiek szerint szabályszerűen közölte volna, elvesztette jogát a döntetlen érvényesítésére. Ezt a jogot újból akkor nyeri el, ha az azonos állás ismét előáll és ugyancsak ő van lépésen.

4. ha a lépésre következő játékos kimutatja, hogy mindkét részről legalább 50 lépés történt ütés vagy gyaloglépés nélkül.

2. Hogyan viszonyul a program a fenti szabályokhoz?

Az előbbiekben bemutatott szabályokat majdnem teljes egészében betartja a program. Sajnálatos módon azonban van szabály, amelyeknek a betartásától el kellett tekinteni, mivel akkora számítási többletet jelentett volna, hogy ezáltal harmadára csökkent volna a program sebessége, és nem is gyakran kerül alkalmazásra. Ez a *háromszori állásismétléses döntetlen.*

Annak figyelése, hogy előfordult-e már a vizsgált állás valamikor a parti folyamán nem is olyan egyszerű, mint amilyennek látszik. Ennek az a magyarázata, hogy egy-egy bonyolultabb állásban a lépésgenerálás folyamán több 10.000 vagy akár 100.000-es nagyságrendű is lehet azon állások száma, amit a program megvizsgál, és ezen összes állásban nézni kellene a lépésismétléses döntetlent. A játékélmény – gyorsan válaszol az ellenfél – és a játékszínvonal (ugyanis a túl sok számolás miatt csökkenteni kell a megvizsgált lépésszámot, amit előregondolkodik) érdekében ennek a szabálynak a betartását mellőzöm.

Egy ötlet felmerült arra vonatkozóan, hogy ne a teljes eddigi partit nézzük, csak az utolsó 3 lépést, hogy elkerüljük azt a helyzetet, miszerint mindkét fél ide-oda lépdel egy-egy bábjával. Igaz, hogy ez nem zárná ki a tényleges állásismétlődést, de a fenti helyzetet megakadályozná, és nem is járna akkora számítási többlettel. Az ötletet azonban nem valósítottam meg a programban, mert nem akartam félmunkát végezni, egy szabályt, vagy betartunk, vagy nem. Valamint, ha nem tartjuk be, akkor ne végezzünk fölösleges számításokat a betartatása érdekében.

Még egy további dologban egy kicsit eltér a szabályoktól, bár ez nem is igazi eltérés, csak nem alkalmaz egy szabályt, azaz nem ajánl döntetlent, illetve nem fogadja el azt. A felhasználói felületen emiatt döntetlent ajánlani nincs is lehetőség.

3. A program használata

3.1. Rendszerkövetelmények

A program mind *GNU/Linux*, mind *Microsoft Windows* operációs rendszer alatt fut, valamint minden egyéb operációs rendszeren van esély a működésre, amit a *Java* támogat, illetve a jövőben támogatni fog.

A program feltételezi, hogy a gépen Java futtató környezet (JRE) van megfelelően telepítve, annak is legalább az 1.4-es verziója, mivel a program ezen verzió alatt lett írva, valamint tesztelve. Ez beszerezhető a http://java.sun.com weboldalról. A telepítést azonban a dokumentáció nem tartalmazza, feltételezi annak meglétét.

A futtatáshoz minél gyorsabb gépre van szükség, legalább 2 GHz-es processzor az ajánlatos. Bár ennél jóval lassabb gépeken is fut, de nagyon sokáig tart amire kiszámol egy-egy lépést, és ezzel élvezhetetlenné válik a játék. Ezen felül semmilyen extra igénye nincs a programnak.

3.2. A program telepítése

A program telepítést nem igényel, csak egy tetszőleges alkönyvtárba kell másolni a mellékelt CD-ről a file-okat. Ezt azért kell megtenni és azért nem futtatható CD-ről, mivel a program a tanuláshoz létre fog hozni egy file-t (`brain.dat`), amiben a megszerzett tudását tárolja, és mivel ezt ugyanabba az alkönyvtárba teszi, mint amiben futtatjuk, így annak írhatónak kell lennie.

3.3. A program indítása

A program indítására kétféle lehetőség is van, ezek közül az első a parancssori futtatás, a második a webböngészőben való futtatás applet-ként.

Továbbá lehetőség van *XBoard* [XBPROTO] kompatibilis felhasználói felület segítségével is használni a programot, a mellékelt `ChessXBoard.jar` file segítségével.

Parancssorból való futtatás esetén a programot a `java -jar Chess2.jar` parancs begépelésével indíthatjuk. Applet-ként való futtatáshoz irányítsuk a böngészőnket a mellékelt `index.html` file-ra.

Ahhoz, hogy külső felhasználói felületet használjunk, be kell állítani a külső programnak, hogy ezt a sakk motort használja. Ezt általában a futtatandó parancs nevével kell megadni, ami jelen esetben a `java -jar ChessXboard.jar` parancsot jelenti.

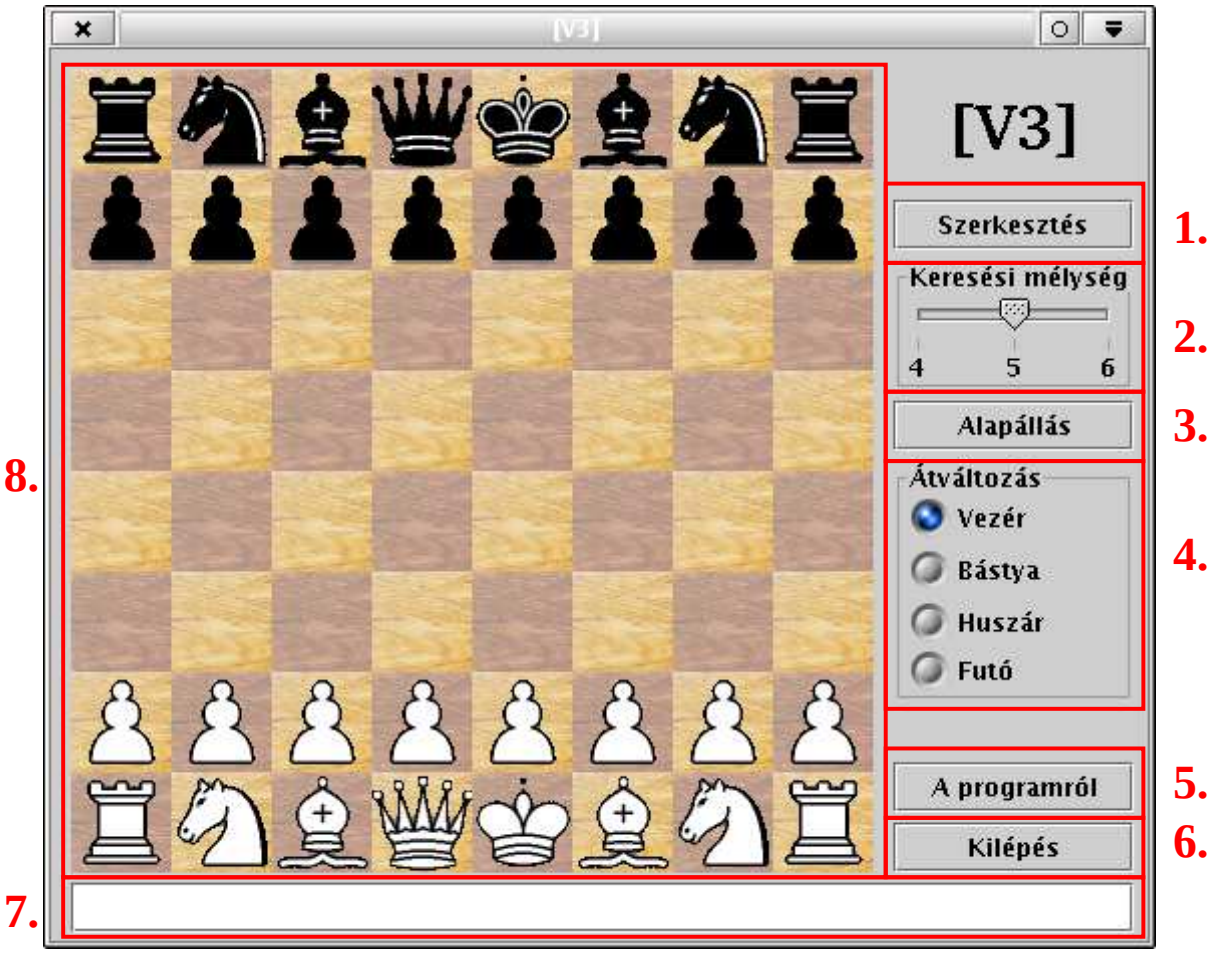
Ha szerencsénk van, akkor ezek után már tudunk is a programmal játszani, ha nincs, akkor még dolgoznunk kell egy kicsit. Két gyakori hibát szoktak elkövetni a sakkfelületet író programozók. Az egyik az, hogy csak 1 szót, az általuk használni kívánt program nevét hajlandók elfogadni. Mivel nekünk ez a Java program, amit felparaméterezünk, hogy a sakkprogramot futtassa, ezért ilyenkor egy kis trükköt kell alkalmaznunk. Egy script-et kell csinálni ami egyszerűen meghívja a fenti parancs segítségével a Java-t, majd ennek a script-nek a nevét kell megadni a programban. Másik gyakori hiba szokott lenni windows

operációs rendszeren, hogy automatikusan hozzáfűznek egy `.exe` kiterjesztést a névhez, amit beadunk. Ilyen esetben átalakíthatjuk a programot `.exe` formátumúvá, erre megvannak a megfelelő programok, de ezeknek az ismertetése nem tárgya ezen dokumentációnak.

3.4. A felhasználói felület

A *játék* képernyő

A program indítása után az 1.2. ábrán látható képernyő fogad minket. Ezen található meg minden, ami ahhoz kell, hogy rögtön játszhassunk. Alapesetben a felhasználó mozgatja a világos bábokat, a számítógép a sötéteket, de lehetőség van ennek a megváltoztatására a *szerkesztés* képernyőn. Azt a „Szerkesztés” nyomógombra (1) kattintva érhetjük el, de erről később lesz bővebben szó.



1.2. ábra: A *játék* képernyő

Most következzék az indulóképernyőn található kezelőszervek részletes leírása és a funk-

cióinak a bemutatása:

Szerkesztés nyomógomb (1) A szerkesztés képernyőre juthatunk a megnyomásával, ahol lehetőség van az állást, valamint a játékosokat is megváltoztatni. Ennek miként-jéről, később lesz szó.

Keresési mélység állító csúszka (2) Ezzel lehet beállítani, hogy a számítógép hány lépés mélyen értékelje ki az állást. Egy 2 GHz-es gépen 4 mélységnél körülbelül 1-2 másodperc egy lépés (a gép gondolkodási ideje), 5 mélységnél körülbelül 10-20 másodperc, 6 mélységnél körülbelül 3 perc. Ennek beállítására csak akkor van lehetőség, ha a felhasználó következik lépésre. Így ha egy lépést már elkezdett kiszámolni a program egy beállítással, akkor azt végigszámolja.

Alapállás nyomógomb (3) A megnyomása hatására a játék teljesen elölről kezdődik, mintha a felhasználó éppen most indította volna a programot. A bábok visszakerülnek az alaphelyzetükbe, a felhasználó mozgatja megint a világos bábokat, a számítógép a sötéteket (ha ezt esetleg átállította közben). A keresési mélység is visszaáll az alapértelmezett 5-ös értékre.

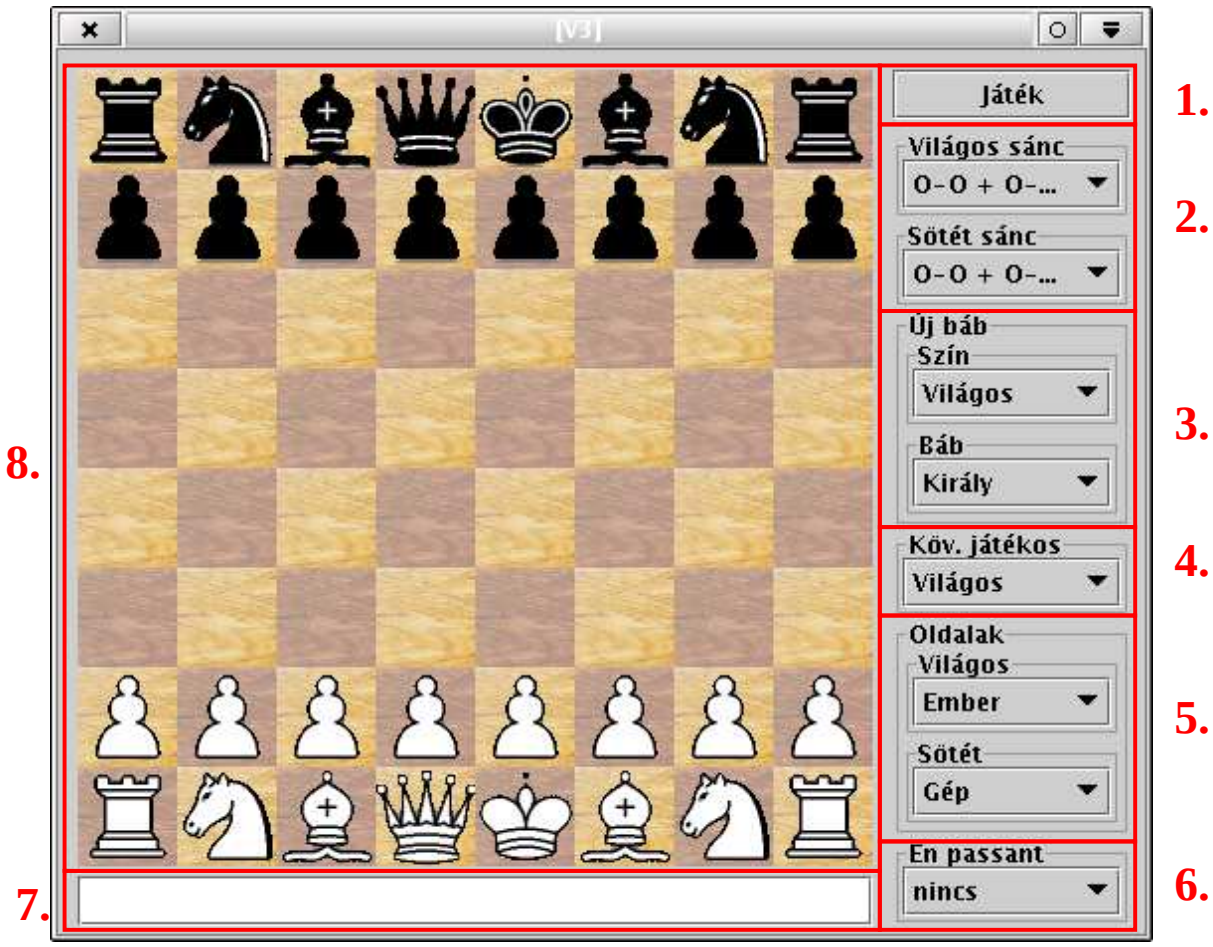
Átváltozás választógombok (4) Ezzel lehet megadni, hogy milyen bábot kapjon a felhasználó, amikor egy gyalogot az utolsó sorra bevisz. Fontos, hogy ezt még a báb lépése előtt meg kell tenni, mert utólag ezen már csak a szerkesztés képernyőn lehet változtatni.

A programról nyomógomb(5) A program nevét, verzióját és íróját lehet vele megnézni.

Kilépés nyomógomb (6) A program használatát lehet befejezni vele. Erre természetesen még az ablakot bezáró „×”-re kattintva is van lehetőség.

Állapotsor (7) A program ide írja ki az üzeneteit. Ezek közé tartoznak a statisztikák, amiket minden számítógépes lépésgenerálásról megkapunk, a hibás lépéseknél kapott figyelmeztetések, valamint a parti eredménye is ide kerül kiírásra.

A tábla (8) Itt látjuk a jelenlegi állást, valamint a lépések megtételére is itt van lehetőség. Ezt a következő módon tudjuk megtenni. Ha valahova kattintunk, akkor az a mező megjelölődik (kék keret), majd ha valahova máshova kattintunk, és az így kijelölt lépés szabályos, akkor az végrehajtódik. Ha nem szabályos a lépés, akkor kiírja azt. Az első mező megjelölését még egy rákattintással meg lehet szüntetni. A sáncolás királylépésnek számít, így itt is úgy kell megtenni (királlyal jobbra, vagy balra a legközelebbi hasonló színű mezőre lépni). Lépésvisszavonásra közvetlenül nincsen lehetőség, mert ahogy a mondás is tartja: „Nem pardonra játsszák a sakkot!”, de áttételesen, a szerkesztés képernyőn keresztül megoldható az is.



1.3. ábra: A *szerkesztés* képernyő

A *szerkesztés* képernyő

A játék képernyőn a „Szerkesztés” gombra kattintva az 1.3. ábrán látható képernyő fogad minket. Itt lehetőség van a bábokat átrendezni, beállítani a lehetséges sáncolásokat, az „en passant” mezőt, azt, hogy melyik színnel ki van, valamint azt, hogy melyik játékos következik a lépésre.

Az ábrán látható kezelőszervek pontos működése a következő:

Játék nyomógomb (1) A játék képernyőre juthatunk a segítségével, hogy az általunk megszerkesztett állásból folytathassuk a játszmát.

Sáncolást állító legördülő menük (2) Segítségükkel az állítható be, hogy melyik oldalra sáncolhatnak a játékosok. A választási lehetőségek a következők: `O-O` – rövid sánc (király oldalára), `O-O-O` – hosszú sánc (vezér oldalára), valamint ezeknek a kombinációi (-, `O-O`, `O-O-O` valamint `O-O + O-O-O`). Fontos, hogy a megfelelő bábok

a helyükön álljanak, mert különben hibát fog jelezni, amint megpróbálunk visszatérni a játék képernyőre.

Új báb kiválasztására szolgáló legördülő menük (3) Itt lehet beállítani azt, hogy a tábla valamely mezőjére kattintva, a program oda milyen bábot tegyen le. Az üres mezőt is a „Báb" legördülő menüben találhatjuk meg. Így ugyan nehézkes lehet sok komplex állást felrakni, de legalább mindent lehet állítani, nem úgy, mint egyes interfészekben, ahol sáncolást és „en passant"-ot nem lehet állítani[1].

Következő játékos legördülő menü (4) Ezzel lehet állítani, hogy melyik fél következik a lépéssel, a világos vagy a sötét.

Oldalak legördülő menük (5) Azt állíthatjuk be, hogy melyik színnel melyik fél legyen, a felhasználó, vagy a számítógép. Két számítógépet nem lehet egymás ellen beállítani, mivel ez problémákat okozna a tanulásában, így sajnálatos módon nincs alkalmunk úgymond „demót" nézni. Ha a felhasználó mozgatja a sötét bábokat, a tábla képe akkor sem fordul meg, a képernyőn ilyen esetben is alul van a világos és felül a sötét.

„En passant" legördülő menü (6) Azt a mezőt tudjuk vele kiválasztani, ahova az „en passant" lépés történhet. Fontos, hogy legyen a mezőhöz viszonyítva megfelelő helyen gyalog, ami a dupla lépést végezhette, mert különben hibát fog jelezni, amikor megpróbálunk a játék képernyőre visszatérni.

Állapotsor (7) A program az üzeneteket írja ki ide, hasonlóan, mint a játék képernyőn. Ez jelenleg egyetlen üzenet lehet, egy hibaüzenet, miszerint nem legális az állás. Ezt akkor kapjuk, ha a játék képernyőre akarunk visszatérni. Ilyenkor nem is enged tovább minket, amíg ki nem javítjuk a hibát. Ekkor a következő dolgok kerülnek ellenőrzésre:

1. Legyen két ellentétes színű király a táblán.
2. A két király ne álljon egymással szomszédos, illetve átlósan szomszédos mezőn.
3. Az utolsó és az első sorban ne álljon gyalog.
4. Csak olyan mezőre mutasson az „en passant" lépés ami alatt illetve felett (a színtől függően) áll gyalog, amitől a dupla lépés származhatott.
5. Csak akkor lehessen sáncolni, ha a megfelelő bábok azokon a mezőkön állnak, ahol kell.

A tábla (8) Az egyes mezőkre kattintva, az *Új báb* legördülő menükkel kiválasztott bábot helyezi oda, illetve üríti ki a mezőt.

[1]Erre a legtöbb programban, csak úgy van lehetőség, hogy alapállásból végiglépkedünk egy teljes játékot, a kívánt pozícióig, és ekkor tudja csak a program, hogy van-e lehetőségünk például sáncolni. Ha felrakunk egy állást, amiben elvileg lehetne sáncolni (a bábok állása miatt), de mi nem akarjuk ennek a lehetőségét megadni, akkor ezt általában nem tudjuk beállítani.

3.5. A gép játékstílusa

A program játékstílusával kapcsolatban több érdekességet, furcsaságot is felfedezhetünk. Ezeknek mindnek megvan a maga indoka. Ebben az alfejezetben ezeket írom le, hogy a felhasználó megértse, hogy miért lépte a számára néha értelmetlennek tűnhető lépést a gép, és, hogy így esetleg ne tekintse akkora őrültségnek azt.

Amit először észrevehetünk az az, hogy a program a kezdőlépésnél is nagyon sokat gondolkodik. Ez azért van, mert nem használ megnyitási könyvtárt, így az alapállást is teljesen ugyanúgy tekinti, mint egy bármely más állást. Így itt is meghatározott mélységig minden lépéslehetőséget kipróbál, aztán lép a gyakorlott sakkozó számára valamilyen szokatlan lépést, amit soha nem találnánk meg a megnyitási könyvtárakban. Itt meg kell jegyezni, hogy a program ezt a „gyermekbetegséget" a tanulás folyamán kinövi, és egyre inkább értelmes megnyitásokat kezd el használni, habár ezt a fejlődést csak nagyon hosszú idő után lehet megfigyelni, mivel az, hogy valami látványos javulást érjen el a tanulás folyamán több száz parti után következik be.

Másik két érdekességet akkor fedezhetünk fel, ha sikeresen a matt határára kényszerítjük a gépet. Ilyenkor két elég gyakori viselkedéssel szembesülhetünk. Az első az, hogy a gép teljesen értelmetlenül feláldozza sorban az összes bábját. Ezt azért teszi, mert a patt számára a nyeréssel egyenértékű. Mivel, ha rosszul áll, a legjobb eredményt akkor érheti el, ha pattot kap. Ennek érdekében mindenét leütteti, mert kevesebb bábbal könnyebben tud pattot kapni. Bár ez általában nem sikerül neki, ennek ellenére már többször is volt alkalmam a sikerét megfigyelni.

Másik gyakori viselkedés akkor következik be, ha a gép pár lépésre van a attól, hogy mattot kapjon. Ekkor teljesen értelmetlennek látszó lépéseket tesz a tábla bal oldalán. Ez azért történik, mivel, ha a gép már látja, hogy nem kerülheti el a mattot a meghatározott lépésen belül, akkor minden lépés ugyanolyan hasznossággal jár a számára és mivel a lépések bal alulról kezdve, jobbra felfele sorfolytonosan kerülnek generálásra, majd az első leghasznosabb (itt ugyanolyan haszontalan) lépést teszi meg.

Itt megjegyzem, hogy máskor is, az előbb említett oknál fogva, előnyben részesíti a lenti és bal oldali bábokkal való lépéseket. Ezt nem olyan könnyű észrevenni, mivel általában nem lesz minden lépés hasznossága közel egyenlő, de ha nagyon erős és megfontolt ellenfél ellen játszik, akkor bármelyik színnel is van általában a bal oldali gyalogszerkezete van jobban előretolva. Tipikusan, ha nem tud jobbat, akkor a baloldali bástyát tologatja jobbra-balra.

A következő furcsasággal akkor találkozhatunk, ha vesztésre állunk a programmal szemben, mert ilyenkor sokszor nem tud mattot adni. Mivel nem rendelkezik külön algoritmussal arra, hogy meghatározott bábukombinációval hogyan kell mattot adni, ezért itt is a szokásos, „nézzünk előre n lépést és lépjük a legjobbat" filozófia szerint gondolkodik. Ennek az az eredménye, hogy ha nem látja a mattadás lehetőségét, akkor teljesen buta lépéseket képes tenni akkor is, ha az ellenfelének már csak királya van, neki meg megvan a megfelelő bábja a mattadáshoz. Ilyenkor még az is előfordul, hogyha van gyalogja a táblán, akkor 49 lépésenként[2] tologatja eggyel előre, azért hogy elkerülje az 50 lépéses pattot, majd az így bejutott átváltozott gyaloggal esetleg sikeresen mattot ad.

[2]Itt most hétköznapi értelemben használom a lépés fogalmát.

Az utolsó gyakori furcsaság, hogy simán megcsinálja a háromszori állásismétléses döntetlent úgy, hogy ugyanazzal a bábuval lépked ide-oda némely helyzetekben. Ezt azért csinálja, mert ilyenkor látja, hogy más lépés esetén nagyon rossz irányba változna az állás értéke, és mivel nincs benne a háromszori állásismétlődéses döntetlen figyelése, így mindig a számára legjobb, de így döntetlenhez vezető lépést teszi meg akkor is, ha egyébként sokkal jobban áll. Csak akkor szakad ki ebből, ha közben az 50 lépéses pattot kell elkerülnie.

És még valami, ami nem furcsaság de mégis jellemzi a gép stílusát, az nem más, mint, hogy sohasem néz el semmit. Már sok magát jó sakkozónak tekintő játékost vert úgy meg, hogy valamit elnéztek és ilyenkor a gép kegyetlen precizitással büntette meg őket. Aztán általában kikapott tőlük a visszavágóban, amikor már komolyabban vették a partit, de ez már egy másik történet.

Az olvasó most megkérdezheti, hogy ezen hibákat, ha tudok róla, miért nem javítottam ki. Ennek az az oka, hogy ezeket nem olyan triviális kiküszöbölni, mivel ezek nagy része az *alfabéta* keresés – amin a program alapszik – sajátosságai közé tartozik és egy szakdolgozat kereteit meghaladó bonyolultságú feladat lenne ezeket mind kiküszöbölni. Tervezem, hogy a jövőben még továbbfejlesztem a programot, és akkor majd ezen „furcsaságokra” is lesz gondom.

2. fejezet

Fejlesztői dokumentáció

1. Fejlesztési környezet

A nyelv választása a hordozhatóság miatt, ami kiemelkedően fontos szempont volt, esett a Java-ra. Azon belül is úgy készítettem el, hogy mind önálló programként, mind applet-ként is futtatható legyen, az erre alkalmas böngészők segítségével.

A program megalkotása folyamán semmilyen fejlesztő környezetet nem használtam, mindent saját magam kézzel írtam az `mcedit`, valamint a `Kate` nevű szövegszerkesztővel. Fejlesztés alatt egy 2 GHz-es Pentium 4-es laptopot használtam 256 MB RAM-mal, amin Gentoo Linux operációs rendszer futott. A legtöbb tesztelést is ezen a gépen végeztem, ide értve az internetes tesztelést is, ami ADSL kapcsolaton keresztül jött létre.

A program fejlesztésében nagy segítségemre volt az internet, ahonnan nagyon sok ötletet és technikát merítettem a sakkprogramozáshoz, valamint az első számú hely volt ahova fordultam, ha bármiféle probléma merült fel a program készítése folyamán.

2. A program komponenseinek leírása

2.1. Áttekintés

A program, mivel a fejlesztés alatt nagy mértékben szem előtt lett tartva az objektum orientált szemlélet, jól elkülöníthető és egymáshoz előre átgondolt felületeken keresztül kapcsolódó komponensekből épül fel. A program fő osztálya a `Chess2`, amely inicializálja a sakktáblát reprezentáló `Table` osztályt, ami az éppen aktuális állásért felelős, beleértve az adott állásból lehetséges lépéseket is. Az alfabéta keresést végrehajtó osztályt, az `AlphaBeta`-t, is a `Chess2` hozza létre, átadva annak az aktuális táblát, amikor a gép következik lépésre. A három osztály között a kommunikáció segítésére szolgál a lépés osztály, `Move`, ami egy konkrét lépést reprezentál. Ezenfelül még van egy statisztikát előállító osztály, a `stat`, amit a program a számítógép lépésének generálásánál használ, és az alfabéta keresés statisztikáit tartalmazza. A `Brain` osztály a számítógép tanulásáért felelős. A táblát reprezentáló osztály, ennek segítségével értékelteti ki az állást, valamint ezen osztálynak egy metódusát

hívja meg a főprogram amikor vége van egy partinak. Ezt azért teszi, hogy a megtett lépések és a végeredmény tekintetében tanuljon a program.

A következőkben bemutatom a fent említett osztályok részletes leírását, függvényeit, és azok pontos működését.

2.2. Move osztály

Ez az osztály egy alapvető átjáró felületet valósít meg a program különböző komponensei között. A fő osztály, `Chess2`, ezen osztály egy példányaként kapja meg az alfabéta keresést megvalósító osztálytól a lépést, amit a számítógép meg fog lépni. Az `AlfaBeta` osztály is ezen osztály segítségével kommunikál a `Table` osztállyal. Sőt a `Table` osztály is ezt használja, a saját magán belüli kommunikációra. Más szóval ez a legkisebb, de az egyik legfontosabb osztálya a programnak, ami összefogja az egészet.

Adatok

int from A lépés kiinduló mezője.

int to A lépés cél mezője.

int what Ha gyaloglépésről van szó, és az utolsó sorba lép a gyalog, akkor azt a bábot jelöli, amivé a gyalog változni fog.

Függvények

Az osztály csak egy pár függvénnyel rendelkezik, és azok sem bonyolultak, csak beállítják illetve lekérdezik a fenti adatokat. Ezek a következők:

int from() Lekérdezi az `int from` változó értékét.

void setFrom(int) Beállítja az `int from` változó értékét.

int to() Visszaadja az `int to` változó értékét a felhasználónak.

void setTo(int) Beállítja az `int to` változót.

void setWath(int) Az átváltozást beállító függvény. A lehetséges értékeit lásd a `Table` osztály adattagjai között.

Move(int, int) Konstruktor, csak a honnan – hova paramétereket kapja meg.

Move(int, int, int) A másik konstruktor, aminek a harmadik paramétere azt a bábot jelenti, amivé a gyalog változni fog.

String toString() Az aktuális lépést adja vissza, embernek érthető formában, mint például: h7-h8q (ezen lépés jelentése, hogy egy gyalog lép a h7-ről a h8-ra és átváltozik vezérré).

A fenti függvények paraméterei, valamint visszatérési értékei egész számok, amik a tábla egy-egy mezőjét jelentik. A 0 az a1-es mező, a 7 az a8-as, a 8-as a b1, majd így tovább egészen 63-ig ami a h8-as mezőnek felel meg.

2.3. Table osztály

Ez az osztály írja le a konkrét állást, generálja le a lépéseket, majd elvégzi azokat, kiértékeli az állást a `Brain` osztály segítségével, ellenőrzi, hogy sakk van-e, valamint adatokkal látja el a program többi részét.

Adatok

int[] table Ez a 64 elemű numerikus tömb reprezentálja az állást. Azért választottam az egydimenziós tömböt, mert úgy gondoltam, hogy a moduló operátor segítségével nem lesznek bonyolultabb a műveletek, mint egy két dimenziós tömb esetében, és így nem kell bajlódni a két indexel. A tömb lehetséges elemei a 2.1. táblázatban szerepelnek.

0	üres	1	világos gyalog	-1	sötét gyalog
		2	világos bástya	-2	sötét bástya
		3	világos huszár	-3	sötét huszár
		4	világos futó	-4	sötét futó
		5	világos vezér	-5	sötét vezér
		6	világos király	-6	sötét király

2.1. táblázat: A bábokat reprezentáló számok

int next Ez a numerikus változó tárolja azt, hogy melyik játékos következik lépésre, ez lehet -1, ha a sötét a következő, és 1, ha a világos. Ez a választás azért is jó, mert az alfabéta keresésnél a minimalizálást, maximalizálást ugyanazzal a függvénnyel lehetett megoldani, ha ezzel az értékkel megszorozzuk a szükséges adatokat. (Erről bővebben az `AlphaBeta` osztály leírásánál lesz szó.)

int castle A változó alsó négy bitje tárolja azt, hogy ki milyet sáncolhat még, valamint a következő két bitje, hogy sáncolt-e már valaki. Ezen bitek érékét a 2.2. táblázat mutatja be.

Stack moves Ebbe a verembe generálom le a lépéseket. A verem használata nagyban megkönnyítette a programozást, mivel az előre definiált műveletekkel egyszerűen tudtam elérni az elemeit.

int enpassant Az „en passant” lépés célmezőjét tárolja, ennek értéke alapesetben -1, de ha valamelyik gyalog duplát lépett, akkor ezt az átugrott mező számára állítom be, majd ezt figyelem a lépésgenerálás során.

Helyi érték	Jelentés
1	világos sáncolhat rövidet
2	világos sáncolhat hosszút
4	sötét sáncolhat rövidet
8	sötét sáncolhat hosszút
16	világos sáncolt
32	sötét sáncolt

2.2. táblázat: Az `int castle` változó bitjeinek értéke

Brain b A `Brain` osztályt érhetjük el ezen a hivatkozáson keresztül. Erre azért van szükség, mivel nem a `Table` osztály végzi el a tényleges állásérték számolást, az csak az összegyűjtött adatokkal hívja meg a `Brain` osztály `float process(float[])` illetve egy másik bár hasonló nevű `float justProcess(float[])` függvényét, majd az onnan visszakapott értéket adja tovább a hívónak.

Move lastMove Az utolsó lépést tárolom benne. Azért van rá szükség, hogy az `AlphaBeta` osztály `int utogetes(int)` függvénye le tudja kérdezni, hogy mely mezőre történt az utolsó lépés, mert majd erre a mezőre tovább kell nézni az ütéseket, hogy *egyensúlyi helyzetet* érhessünk el, de erről majd az `AlphaBeta` osztály leírásában lesz bővebben szó.

boolean utes Az `AlphaBeta` osztálynak van rá szüksége, mert az `int utogetes(int)` függvény meghívását csak akkor végzi el, ha ütés volt az utolsó lépés, és ezt ezen változó tárolja. Ha nem volt ütés és a kiértékeléssel levélcsúcsba jutottunk, akkor egyensúlyi állás van.

int otven Ezen változó az ötven lépéses patthoz számolja a lépéseket, ami nálam akkor következik be, ha a változó értéke eléri a 100-at, mivel ahogy már említettem az Előszóban, féllépéseket számolok. A `void move(Move)` függvény növeli az értékét mindig 1-el, valamint nullázza azt, ha gyaloglépés vagy ütés történt.

Függvények

Table(Brain) Konstruktor. Egyetlen paramétere egy `Brain` osztályra mutató hivatkozás, aminek az értékét átveszi és bemásolja a saját `Brain b` változójába.

Table(Table) A copy-konstruktor. A paraméterként átadott tábla másolataként példányosítja önmagát. Átmásolja az `int otven`, a `boolean utes`, a `Move lastMove`, az `int next`, a `Barin b`, az `int castle` változók értékeit, valamint az `int table[]` tömb elemeit.

void init() Ez a függvény felelős a tábla alapállapotba hozásáért. Felpakolja a bábokat, beállítja, hogy a világos következik lépésre, azaz teszi meg a kezdő lépést. A sán-

colásért felelős integer felső négy bitjét 1-esre, a többit 0-ra állítja, valamint új vermet hoz létre a legenerált lépések számára.

String toString() A tábla ASCII képét adja vissza, a 2.1. ábrán látható formában.

```
   a b c d e f g h

8  r n b q k b n r  8
7  p p p p p p p p  7
6  . . . . . . . .  6          Next: White
5  . . . . . . . .  5
4  . . . . . . . .  4
3  . . . . . . . .  3
2  P P P P P P P P  2
1  R N B Q K B N R  1

   a b c d e f g h
```

2.1. ábra: A tábla ASCII képe

Ez nagyon hasznos hibakeresés közben, de végfelhasználói szempontból lényegtelen, mivel a grafikus megjelenítésnél ezt a függvényt nem használom. Egyébként is, ha valaki Java programot ír, akkor illik minden osztályhoz `String toString()` függvényt létrehozni.

void genMoves() Ez a függvény az egész osztály úgymond „lelke”. Ez generálja le az összes lehetséges lépést a soronkövetkező játékos számára az aktuális álláshoz. Természetesen azt is ellenőrzi, hogy a lépés következtében a gép ne lehessen sakkban, ezt a `boolean checkChess(Move m)` függvény hívogatásával ellenőrzi. Ha sakk lenne, akkor azt a lépést nem veszi figyelembe. A lépéseket a `Stack moves` verembe teszi. Ezután ezeket átrendezi úgy, hogy az olyan lépések kerüljenek felülre, ahol ütés történik. Ez azért hasznos, mert így a legenerált játékfában hamarabb jönnek elő az állások közti értékbeli különbségek, és így remélhetőleg többet és hamarabb vág majd a lépésgenerálás folyamán, ezzel gyorsítva azt.

void setBrain(Brain) Beállítja a `Brain b` változó értékét. Akkor van rá szükség, ha például megváltozik a számítógép által mozgatott bábok színe, – mert a felhasználó a szerkesztés képernyőn átállítja azt –, mivel ekkor új `Brain` osztályt kell példányosítani hozzá. Ezek után a táblával tudatni kell, hogy hogyan éri azt el.

boolean movesEmpty() Igazat vagy hamisat ad vissza attól függően, hogy üres-e a lépéseket tároló verem. Az `AlphaBeta` osztály szokta hívogatni, amikor végigmegy az összes lehetséges lépésen egy ciklus segítségével.

boolean validateMove(Move m) A paraméterként átadott lépés érvényességét ellenőrzi úgy, hogy legenerálja az össze lehetséges lépést, majd megnézi, hogy a megadott lépés közte van-e. Ezt a GUI[1] használja, a felhasználó által beadott lépés érvényességének az ellenőrzésére.

int next() Ez a függvény visszaad -1-et vagy 1-et, attól függően, hogy melyik játékos a következő. A már említett alfabéta keresés maximalizálásánál illetve minimalizálásánál van rá szükség. Ezen kívül a `Chess2` osztály is gyakran használja, például akkor is, amikor azt nézi meg, hogy mikor kell a gépnek elkezdeni gondolkodni. Ilyenkor a számítógép színét (-1 illetve 1) hasonlítja össze a függvény által visszaadott értékkel.

void setNext(int) Ha a felhasználói felületen megváltoztatják a soronkövetkező játékos színét, akkor ezzel a függvénnyel hozzuk azt a tábla tudomására.

Move movesNext() Kiveszi és visszaadja a verem tetején lévő – következő – lépést.

boolean utes() Az `boolean utes` változó lekérdezésére szolgál, ami azt mondja meg, hogy az utolsó lépés ütés volt-e vagy sem.

Move lastMove() A függvény az utolsó lépést, azaz a `Move lastMove` változó értékét adja vissza a hívónak.

void move(Move) Elvégzi a táblán a paraméterként átadott lépést. Növeli, illetve nullázza az `int otven` változó értékét a lépéstől függően, beállítja a `boolean utes` értékét. Ha sáncolás történt akkor frissíti az azt leíró változót. Továbbá az `int enpassant` változó állításáról is gondoskodik.

void setCastle(int) Az `int castle` változó értékét állítja be az átadott értékre. Akkor használatos, amikor a felhasználói felületen a Szerkesztés módban megváltoztatják a sáncolási lehetőségeket.

int getCastle() Az előző függvény által beállított változó lekérdezésére szolgál. Szintén a GUI használja.

void setEnpassant(int) Az „en passant” lépés célmezőjét állítja be vele, szintén a felhasználói felületen való állíthatóság miatt van rá szükség.

int getEnpassant(int) Mint az előző függvény csak ez a lekérdezést végzi, azaz visszaadja az `int enpassant` változó értékét.

float value() Az állás értékét adja vissza. Összegyűjti a bemenő paramétereket, jelenleg 802-et majd ezeket a `Brain` osztály `float justProcess(int[])` függvénye segítségével kiértékelteti. Végül $+/-5$ ezrednél kisebb hibát ad hozzá véletlenszerűen és visszaadja az eredményt a hívónak. A hibára azért van szükség, hogy ne mindig ugyanazt lépje a számítógép egy-egy állásban.

[1] Graphical User Interface - grafikus felhasználói felület

A jelenleg figyelt értékek a következők:

1. A különféle bábok darabszáma.
2. Minden mező – báb kombinációra, hogy az adott báb az adott mezőn van-e.
3. A sáncolások lehetőségét, illetve azt, hogy megtörtént-e már.
4. A dupla gyalogok számát.
5. A tisztekre, hogy a saját helyükön állnak-e.

Ezen értékek figyelésének az ötletét a *KnightCap* [KCAP] programból vettem, mivel ő is hasonló értékek figyelésével nagyon látványos fejlődésről és játékerőről tett tanúbizonyságot. Első ránézésre furcsának tűnhet, hogy például olyanokat figyelünk, hogy az *c4-en van-e gyalog*. A tesztelések folyamán azonban nagyon szépen megtanulta a program, hogy jó neki, ha az utolsó előtti soron állnak a gyalogok. Továbbá gyalogszerkezetek képe is ki-kirajzolódik az ábrán, ha megjelenítjük, hogy hol pozitív, illetve negatív a súly, ha ott gyalog áll.

Az ilyen viszonylag sok súllyal[2] az a baj, hogy elég lassan konvergálnak egy elfogadható értékre. Körülbelül 50-100 partit kell végigjátszani, hogy valamilyen javulás legyen megfigyelhető.

void learnFromMove() A működése teljesen megegyezik az előbb bemutatott `float value()` függvényével, kivéve a végén, ahol nem a `float justProcess(int[])` függvény segítségével adjuk át az összegyűjtött adatokat a `Brain` osztálynak, hanem a `void process(int[])` függvénnyel. A kettő között az a különbség, hogy míg az előbbi csak kiszámolja az értéket a bemenő adatok alapján, az utóbbi ezt el is tárolja, majd az így megkapott bemenet – kimenet párok alapján fog tanulni.

Erre a szétbontásra azért van szükség, hogy a lépésgenerálás folyamán a faépítés alatt ne tanuljunk minden egyes kipróbált lépésből csak abból, amit meg is lépünk. Így ezt a függvényt a főprogramnak kell meghívnia minden ténylegesen meglépett lépés után.

void setPiece(int,int) Az első paraméterként megadott mezőre felteszi, a második paraméterként megadott bábot.

int getPiece(int) A függvény a paraméterként adott mezőn álló báb numerikus reprezentációját adja vissza.

boolean otven() Akkor ad vissza igazat, ha az `int otven` változó értéke több, mint 100, azaz már 50 egész lépés óta nem történt ütés vagy gyaloglépés, azaz az 50 lépéses patt esete áll fenn.

[2]Ebben az esetben nem is olyan sok, mivel a már említett KnightCap több, mint 6000 súlyt tanul, eredményesen.

boolean checkChess(Move) Ellenőrzi, hogy a megadott lépés végrehajtása után sakk van-e. Ezt úgy végzi, hogy először a táblát reprezentáló tömböt lemásolja, majd ugyanúgy, mint a `void move(Move)` elvégzi rajta a lépést. Ezután a lépő játékos királyától kiindulva végignézi a mezőket, ahonnan sakkban lehetne. Azért csak a lépő játékos királyát ellenőrzi, és az ellenfélét nem, mivel ezt a függvényt egyedül a lépésgenerálásnál van használva, és ott a kérdés az, hogy legális-e a lépés.

boolean checkChess(int) A megadott színű (-1,1) játékosra ellenőrzi, hogy sakkban van-e. A matt illetve a patt eldöntésekor használja a főprogram, amikor kiíratásra kerül a játék eredménye, továbbá a számítógépnek a lépésgenerálás folyamán az `AlphaBeta` osztály `float kiertekel2(float,float)` függvényében is hasonló feladatokra van használva.

2.4. Stat osztály

Ez az osztály készíti az `AlphaBeta` osztály által végzett lépés-generáláshoz a statisztikát.

Adatok

int node A megvizsgált nem levélcsúcsok számát tartalmazza.

int leaf Ez a numerikus változó tárolja a megvizsgált levélcsúcsok számát.

int cut A vágások számát tartalmazza.

int mely A maximális mélységet tartalmazza, amilyen mélyen lement a keresés a fában.

long time Az osztály példányosításakor – a keresés kezdetekor – aktuális időpontot tartalmazza, 1970 január 1 00:00:00 óta eltelt ezredmásodpercek száma által reprezentált formában.

Függvények

Stat() Ez a konstruktor. Feladata, hogy a fenti változókat nullázza, valamint, hogy az aktuális időt elmentse a `long time` változóba.

void addNode() Az `int node` értékét növeli eggyel.

void addLeaf() Az `int leaf` változó értékét növeli eggyel.

void addCut() Eggyel növeli az `int cut` értékét.

void melyseg(int) Ha a paraméterként megkapott szám nagyobb, mint az `int mely` változó értéke, akkor az utóbbi értékét lecseréli az előzőére.

String toString() Egy karakterlánc formájában visszaadja a változók értékét. Ide értve a meghívás időpillanata és a `long time` által eltárolt idők közti különbséget, másodpercben.

2.5. AlphaBeta osztály

Ez a program „lelke". Ez az osztály generálja a gép következő lépését. Ennek az alapja az *alfabéta* keresés (erről később lesz szó bővebben), ami ki van egészítve egy előkereséssel. Ez fele mélységgel hajtja végre a keresést, és az így létrejött lépés – hasznosság sorrendjében végzi a tényleges optimális lépés keresést, ezzel is növelve a *vágás*ok (lásd később) esélyét. Ezenfelül még egy kis kiegészítés is került a lépésgenerálásba, miszerint ha levélcsúcsba érünk és nincs *egyensúlyi helyzet* (lásd később), akkor tovább építjük a játékfát, amíg el nem érjük az egyensúlyi helyzetet.

Az alfabéta keresés

Az alfabéta keresés a *minimax* algoritmus egy továbbfejlesztett változata. Így mielőtt ezt megnéznénk, nézzük meg a minimax algoritmust, és később azt, hogy az alfabéta keresés miben tér el ettől.

Hívjuk az egyik játékost – aki a célfüggvény értéke szerint a legkisebb értékű lépést akarja meglépni – MIN-nek, a másikat – aki a nagyobb értékűt akarja meglépni – MAX-nak (az algoritmus neve is innen ered). Generáljuk a játékfát[3], meghatározott lépés[4] mélységben, majd a levélcsúcsokra alkalmazzuk a kiértékelő függvényt, hogy megtudjuk annak az elért állapotnak az értékét. Minél nagyobb annál jobban kedvez MAX-nak mivel ő maximalizál, és minél kisebb annál jobb MIN-nek. A fa, nem levél csúcsaira „futtassuk fel" az értékeket olyan módon, hogy ha a MAX szintjén vagyunk, azaz MAX következik lépéssel, akkor vegyük a gyerekcsúcsai értékének a maximumát és az legyen az ő értéke, ha MIN szintjén vagyunk, akkor vegyük a gyerekcsúcsai értékének a minimumát. Ezt egészen addig folytatjuk, amíg el nem jutunk a fa gyökeréig. A gyökércsúcsnál, ha az MAX szintje, akkor kiválasztjuk azt a gyerekcsúcsot (lépést), ahonnan a legnagyobb értéket kaptuk. Ha több csúcstól is kaptuk a legnagyobbat akkor véletlenszerűen lehet választani[5]. Ha MIN szintje a gyökércsúcs, akkor a legkisebb értékkel rendelkező gyerekcsúcsnak megfelelő lépést kell kiválasztani.

A 2.2. ábra egy két lépésre legenerált játékfát mutat, a levélcsúcsok kiértékelése, és ezen értékek felfuttatása után. Az algoritmus a középső ágat fogja választani, mivel így legrosszabb esetben is 1 értékű csúcsba tud jutni, míg a többi választás esetén -1, sőt -3 értékűbe is juthatna MIN választása után.

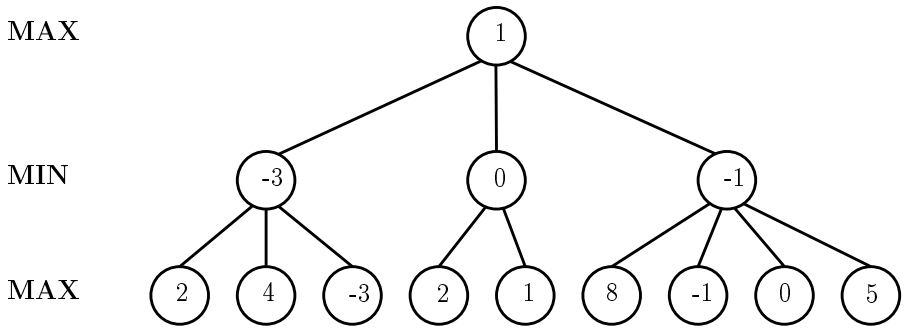
Az előbbiekben leírt minimax algoritmus, valamint az alfabéta keresés is rekurzív függvénnyel valósítható meg a legkönnyebben, ezért én is így valósítottam meg a programban. A konkrét megvalósítás a függvények leírásánál kerül ismertetésre.

Az alfabéta keresés a minimaxon túl számon tart egy α és egy β értéket, ami a MAX valamint MIN eddigi legjobb választását tárolja az éppen aktuális keresés folyamán. Ezeket

[3]Olyan körmentes irányítatlan gráf, aminek a csúcspontjaiban az állások vannak, az élei azokkal a lépésekkel vannak címkézve, amelyek a segítségével az egyik csúcsból az azt követőbe (a gyerekcsúcsába) lehet jutni. A játékfa gyökere az az állás, amihez keressük a számítógép számára legjobb lépést.

[4]A „lépés" egy fél lépést jelent.

[5]Általában a legelsőt szokás választani, mivel programozói szempontból az egyszerűbb és jobb is.



2.2. ábra: Minimax keresés

az értékeket az algoritmus folyamatosan frissíti, majd befejezi a rekurzív hívást, azaz levágja a részfát, ha az α nagyobb vagy egyenlő lesz, mint β.

Abban az esetben, ha MAX csúcsában vagyunk, akkor ezen csúcs alatti részfában MIN tudna számára jobb, MAX számára rosszabb levélcsúcsot elérni, mint MAX eddigi legjobb olyan csúcsa, amit eltud érni (α). Emiatt fölösleges azt a részfát tovább nézni. Hasonlóan, ha MIN szintjén vagyunk, akkor a csúcs alatti részfában MAX tudna elérni nagyobb értékű levélcsúcsot, mint amit MIN jelenleg el tud érni (β), így azt a részfát is fölösleges tovább nézni.

A 2.3. ábra egy három szintű játékfán elvégzett alfabéta keresést mutat. A szaggatott nyilak a rekurzív függvény által visszaadott értékek útját mutatják. Az olló azt mutatja, hogy hol történt vágás. Így a szaggatott vonallal levágott csúcsok nem kerülnek kiértékelésre. Annak ellenére, hogy ezen levélcsúcsok értékei is fel vannak tüntetve, az algoritmust azok nem befolyásolják.

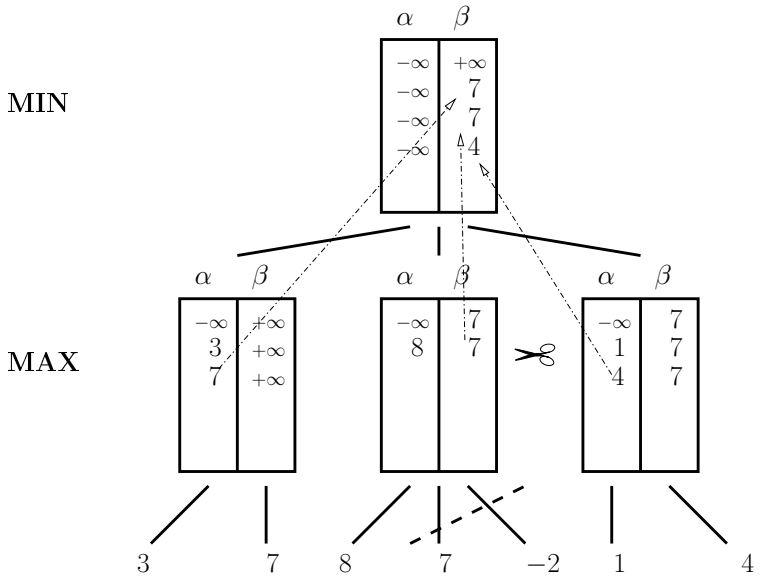
A baloldali részfa kiértékelése után látható, hogy ha MIN azt a részfát választja, akkor MAX választása után a legkisebb érték amit el tud érni az 7. A középső részfában az első levélcsúcs kiértékelése után már látszik, hogy ott MAX tudna olyan lépést választani, ami legalább 8-az értékű levélcsúcsba vezet. Ezért ez a részfa érdektelen MIN számára, mivel már tud jobbat, így levágjuk a további részeket.

Be lehet látni [MIKÖNYV], hogy a vágások segítségével, ugyanannyi idő alatt körülbelül kétszer olyan mélyen ki lehet értékelni a fát, mintha minimaxot használnánk, Mivel az eredmény ugyanaz mindkét esetben, így bár az elméletben nagy jelentőségű a minimax algoritmus, azonban a fentiek miatt a gyakorlatban nem szokás használni.

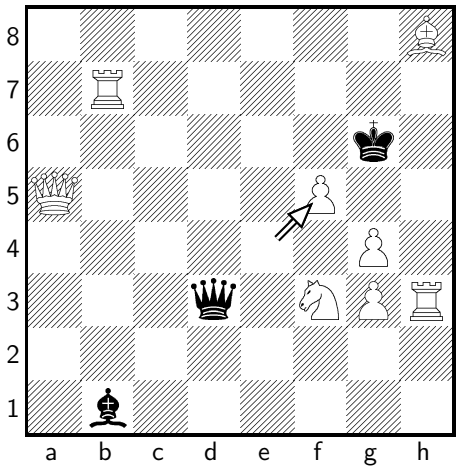
A nyugalmi helyzet

Nyugalmi helyzet akkor van, ha a keresési fa levélcsúcsában nem áll az a bábu ütésre, amelyik az utolsó lépésben ütött, mivel az arra érkező esetleges válasz meghamisíthatná a kiértékelés eredményét.

A nyugalmi helyzet elérésének fontosságát szemléltessük egy példával. Tekintsük a 2.4. ábrán látható állást. Fehér utolsó lépése a nyíllal jelzett **exf5** gyaloglépés volt, amivel egy,



2.3. ábra: Alfabéta keresés



2.4. ábra: Nem nyugalmi helyzet

az f5-ön álló fekete bábut ütött. Ha a szokásos, a 2.3 táblázatban látható, bábértékeket[6] tételezzük fel, akkor a jelenlegi állás értéke $16,5$.

Ebben az esetben azonban nincs nyugalmi helyzet. Nyilvánvaló, hogy feketének következő lépésben vezérrel ütni kell a gyalogot, hogy elodázza a mattot. Ezt természetesen a fehér gyaloggal üti, majd miután ezt a gyalogot is ütötte a sötét futóval, világos nem üti azt, mivel a fekete futóját már csak vezérrel tudná ütni, de azt leütné fekete a királlyal. Most

[6]Több szokásos bábérték is létezik, ez az egyik ezek közül.

Báb	Érték
gyalog	1
futó	3,5
huszár	3,5
bástya	5
vezér	9

2.3. táblázat: Egy elterjedt bábérték táblázat

már nyugalmi helyzet van, és az így kapott állás értéke 23,5, ami jócskán különböző, mint az előzőleg kiszámított. Gondoljunk csak bele, hogy egy-egy ilyen kis „elszámolás" mennyire kedvezőtlenül befolyásolhatja a parti kimenetelét, ha ezért veszíti el a gép például a vezérét.

Az ilyen helyzetek kezelésére jött létre a következő algoritmus[7]. Ha az utolsó lépés ütés volt, és van olyan ellenséges báb vagy bábok, amelyek ugyanarra a mezőre üthetnek, mint ahova az ütés történt, akkor a program a következőket teszi. Leüti a kérdéses mezőn lévő bábot felváltva mindig a legkisebb értékű bábbal egészen addig amíg van lehetséges ütés. Ezek után kiértékelésre kerül a kapott állás, majd annak az értéke visszaadásra kerül a szülőnek. Ha MAX szintjén van az algoritmus, akkor a saját és a kapott érték közül azt az értéket kell feladni a szülőnek ami a nagyobb, mivel ezen a szinten MAX dönthet, hogy elvégezi-e az ütést, vagy nem. Ha MIN szintjén van az algoritmus, akkor a kisebb értéket kell feladni a szülőnek. Ezen szabályok alól kivétel az, ha csak egy lépéslehetőség van, mivel ilyenkor sakkot kell elodázni, így feltétlenül kell ütnünk (ez látható volt az előző példában is). Az így visszaadott érték már reálisan tükrözi az állás valós értékét.

A 2.5. ábrán szemléltetem a fenti algoritmus működését. A csúcsok jobb oldalán található az aktuális állás értéke. Az átlósan felfelé mutató nyilak szemléltetik a rekurzió által visszaadott értékek útját. Látható, hogy a szaggatott vonal feletti csúcs elérésével MAX megáll az ütögetésben, mivel ez a számára elérhető legjobb érték.

A program tesztelése közben kiderült, hogy nagyon gyakran kerül alkalmazásra ez az algoritmus és átlagosan a maximális keresési mélység mintegy másfélszerese mélyen megy le a játékfában az ilyen esetekben. Így, bár csekély többlet számolási kapacitás ráfordításával, de sokkalta pontosabban sikerül megkapni a legjobb lépést.

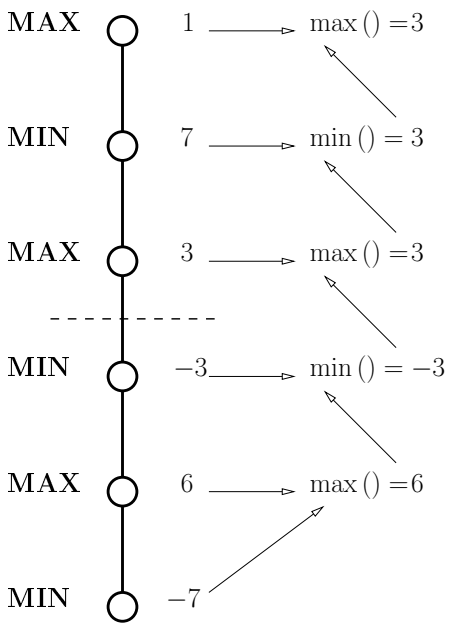
Most, hogy túl vagyunk az elméleti alapozáson, következzen a fent leírtaknak a tényleges megvalósítása.

Adatok

Table table Ezen a változó tárolja a csúcshoz tartozó állást, mivel (mint később kiderül) minden csúcshoz a fában új `AlphaBeta` osztály példányosódik. A szülő példánytól kapott állást másolja a változó által hivatkozott osztályba a konstruktor.

Stat stat A statisztikát készítő osztályra lehet rajta keresztül hivatkozni. Először csak

[7]Itt meg kell jegyeznem, hogy az algoritmus jóbarátomtól, Prorok Mártontól származik.



2.5. ábra: A nyugalmi helyzet keresése

pusztán tesztelési céllal készült a statisztika, de mivel sokan szeretik nézegetni az ilyen adatokat, ezért a végeredmény a felhasználói felületre is kikerült. A következő értékeket tartja számon: megvizsgált nem levélcsúcsok számát, kiértékelt levélcsúcsok számát, a maximális mélység, amennyire lementünk a fában valamint, hogy mennyi ideig tart a keresés.

int melyseg A numerikus változó tárolja azt, hogy az algoritmus milyen mélyen van a játékfában attól a ponttól ahonnan a keresést elindítottuk. Ezen változó értéke kerül ellenőrzésre, amikor el kell dönteni, hogy kell-e még tovább lemenni a fában, vagy ki kell értékelni az adott csúcsot.

int maxmelyseg A keresés alatt maximálisan elérhető mélység van benne. Kivételt képez ez alól, ha a levélcsúcsban még nem került elérésre a nyugalmi helyzet, ilyenkor a program további mélységben járja be a fát.

Beágyazott osztályok

MoveVal Ez az osztály kizárólag arra az egy célra való, hogy a `Move general2()` függvényben az összetartozó lépés – érték párokat kényelmesen lehessen kezelni, ezért ez a két adat publikusan[8] van benne, valamint egy konstruktor, ami beállítja ezen értékeket.

[8] A megvalósítás érdeme az egyszerűség, a lekérdezőfüggvények feleslegesen elbonyolították volna a programot.

Függvények

public AlphaBeta(Table t, int melyseg, int maxmelyseg, Stat stat) Ez a konstruktor. Feladata, hogy az átadott állást a saját osztály `Table table` változójába másolja, amit a `Table` osztály saját maga által paraméterezett konstruktorával teszi meg. Az `int melyseg` által leírt számot növeli eggyel, és azt állítja be a saját `int melyseg` változójába. A `Stat stat` változóját beállítja a megkapott értékre, ami nem új osztály, csak egy hivatkozás. Ennek az az oka, hogy az egész keresés alatt ugyanazt a statisztikai osztály példányt kell adatokkal ellátni azért, hogy a főprogram azt majd ki tudja írni. Végül: az átadott maximális mélységet is átveszi és beállítja vele az `int maxmelyseg` változóját.

public Move general2()[9] Ez a függvény valósítja meg az osztály és a külvilág közötti információátadó feladatokat. Ez a függvény adja vissza a számítógép számára a legjobb lépést, és a meghívásával lehet elindítani ennek a keresését.

A tényleges működés a következő: a függvény legeneráltatja a táblával a lehetséges lépéseket (`float genMoves()`). Minden lehetséges lépésre végrehajtat egy fele maximális mélységű alfabéta keresést. Ez úgy történik, hogy először létrehoz egy `ArrayList moves` listát[10], amiben a lépés – érték párok egy `MoveVal` osztályba burkolva helyezkednek el. Ezek után egy ciklussal sorba veszi a tábla által legenerált lépéseket, majd mindegyikhez példányosít egy új táblát a mostani másolataként, majd a lépést elvégzi rajta. Ezek után példányosít mindegyikhez egy új `AlphaBeta` osztályt fele keresési mélységgel, majd meghívja rá a `float kiértékel2(float,float)` függvényt, ezekután az általa visszaadott értékekkel és a lépésekkel a `MoveVal` osztály $1-1$ példányát példányosítja.

Ennek megtörténte után a kapott eredményeket rendezi érték szerint, majd ilyen sorrendben hajtja végre a rendezett lépésekre a tényleges keresést, ezzel növelve a vágások előfordulását.

Az igazi lépésgenerálás a következő módon történik: a program felvesz egy változót (`Move bestmove`) az eddig talált legjobb lépés tárolására, valamint egy másikat (`float bestertek`), amiben ennek az értékét tárolja, kezdetben a lépés az a1-a1, az értéke MAX esetén -100.000, MIN esetén 100.000. Továbbá beállítja az `float alpha` és `float beta` változókat, két extremális értékre (konkrétan ez -1.000 és 1.000)[11] .

[9] A névben a 2-es, a verzióra utal, mivel az első verzióban még nem volt nyugalmi helyzet keresés és lépés előrendezés.

[10] Azért esett erre az osztályra a választás, mert számomra kényelmes műveletekkel rendelkezik.

[11] Itt fontos, hogy ez az érték is és a `float bestertek` is a kiértékelő függvény által kiadott értéktartományon kívül essen, és az α és a β értéke kisebb legyen, mint a `float bestertek`. Ellenkező esetben előfordulhatna, hogy az algoritmus azt adja vissza, hogy nincs az a1-a1-nél jobb lépés (matt esetén adja vissza ezt az algoritmus), pedig a gép még nincs bemattolva, csak látja, hogy úgysem kerülheti el azt. Továbbá fontos az is, hogy a későbbiekben bemutatásra kerülő `float kiertekel2(float,float)` függvény által matt, illetve patt esetén visszaadott érték ezen két érték közé essen, mivel különben nem biztos, hogy észrevenné a mattot.

Majd jóság szerinti sorban az előbb kihozott lehetséges lépéseken végigmenve végrehajtja a következőket:

Első lépésként, készít egy másolatot a tábláról és végrehajtja rajta a lépést. Ezek után példányosít vele egy új `AlphaBeta` osztályt, majd erre meghívja, `float alpha` és `float beta` értékekkel, a `float kiertekel2(float,float)` függvényt. A függvény által visszaadott értéket összehasonlítja a jelenleg talált legjobbal. Itt mindkettő értéket megszorozza az `int table.next()` által visszaadott értékkel, mivel ez -1, ha sötét jön és 1, ha világos. Emiatt az összehasonlítást, nem kell két ágra bontani attól függően, hogy MAX vagy MIN szintjén van. Ha az eddigieknél jobb értéket kap, akkor lecseréli a legjobb lépést és annak az értékét az újra. Továbbá a `float alpha` és a `float beta` értékét is frissíti a következő módon. Ha világos szintjén van azaz maximalizál, akkor `float alpha` értékét cseréli le a visszaadott értékkel, ha az annál nagyobb. Ha sötét jön lépésre akkor `float beta` értékét cseréli le, ha a függvény által adott érték kisebb nála.

Amikor már nincs több lépés, amit megnézhetne a program, akkor visszaadja a legjobb lépést a meghívónak, ami általában a főprogram.

float kiertekel2(float alpha, float beta) Ez nagyon hasonlít a `Move general2()` függvényhez. Majdnem ugyanazt a feladatot végzi, azaz új gyerekcsúcsokkal bővíti a játékfát a bejárás folyamán. A különbség a két függvény között az, hogy az előbbi csak a fa legelső szintjén van jelen, míg ez a függvény a fa belső csúcsaiban hívódik meg. Ez azért van, mert az előző függvény a legjobb lépést adja vissza, míg ez a függvény a legjobb lépés értékét. További különbség, hogy itt már ellenőrzi, hogy milyen mélyen van, és, hogy elérte-e már a maximális mélységet.

A tényleges működés a következő lépésekből tevődik össze: először megnézi, hogy elérte-e már az 50 lépéses[12] patthoz szükséges határt. Ha igen, akkor rögtön visszatér 0-val, mint állásértékkel. Másodszorra ellenőrzi, hogy van-e legális lépés. Ha nincs, akkor vagy patt, vagy matt van. Ezt a lehetséges lépések legenerálásával ellenőrzi, és ha nem áll meg itt, akkor az itt legenerált lépéseket használja fel a későbbiekben is, emiatt nincs fölösleges számolgatás akkor sem, ha nincs matt vagy patt. Ha nincs lehetséges lépés, és sakkban van a játékos, akkor matt van. Ilyenkor 10.000-et vagy -10.000-at ad vissza, attól függően, hogy melyik szinten van (MIN/MAX). Ha nincs sakk, akkor pont az ellenkezőjét adja vissza, mivel patt van, és az általában annak jó, aki kapja. Ez azért van így, mivel lehetne akár 0-t is visszaadni ilyenkor, de az nem írná le igazából a nyereséget, amit azzal ér el, hogy pattot sikerült kiharcolnia vesztes állásban. A program korai szakaszában a pattot is olyan értékkel vettem, mintha kikapott volna a program, ezzel megpróbálva elkerülni, hogy pattot kapjon. A későbbiekben kiderült, hogy ez nem volt jó megoldás. Ennek az oka az, hogy a pattadást úgy értelmezte, hogy az az ellenfélnek rossz. Emiatt, ha nyerésre is állt, de a játékfában nem talált mattadási lehetőséget, de pattot igen, akkor azt adta be nagy örömmel.

[12] Ez még mindig 100 lépés.

Miután a program megnézte a pattot valamint a mattot, és egyikkel sem találkozott (nem tért vissza), akkor ellenőrzi, hogy nem ért-e már el a maximális mélységet. Ha igen, akkor megnézi, hogy ütés volt-e az utolsó lépés (a `Table boolean utes()` függvénye segítségével). Ha igen akkor meghívja az `float utogetes(int)` függvényt arra a mezőre, ahova az ütés történt (ennek értékét a `Table int lastMove()` függvénye adja meg). Ezután az általa visszaadott értékkel visszatér. Ha nem volt ütés, akkor az aktuális állást értékeli ki a tábla `float value()` függvénnyel és az így kapott értékkel tér vissza.

Ha még nem érte el a maximális mélységet (nem tért vissza), akkor az előzőekben már legenerált összes lehetséges lépésen sorban végigmenve mindegyikhez másol egy új példányt a tábláról, majd ezen végrehajtja a lépést. Példányosít hozzá egy `AlphaBeta` osztályt és meghívja a `float kiertekel2(float,float)` függvényt rá. Ha a következő játékos a világos, és a visszaadott érték nagyobb, mint a `float alpha` értéke, akkor egy eddigi legjobbnál is jobb állást talált, ezért a `float alpha`-t lecseréli az új értékkel, és ha a `float alpha` nagyobb vagy egyenlő, mint a `float beta`, akkor visszatér a `float beta` értékével, mivel vágás volt. Hasonlóan jár el, ha a következő játékos a sötét, és ha a `float beta` nagyobb, mint a visszaadott érték. Ekkor lecseréli a `float beta`-t az új értékre, mivel minden eddiginél jobbat talált a sötétnek. Ha a `float alpha` nagyobb vagy egyenlő, mint a `float beta`, akkor itt is visszatér a `float alpha` értékével.

Ha végignézte az összes lehetséges lépést, és még nem tért vissza, akkor attól függően, hogy melyik játékos a következő, tér vissza, vagy `float alpha`-val vagy `float beta`-val. Az előbbivel akkor, ha világos a következő lépéssel, az utóbbival pedig, ha a sötét.

float utogetes(int to) Ez a függvény végzi a levélcsúcsokban az előzőekben bemutatott algoritmus szerint a nyugalmi helyzet keresését.

Első lépésben megnézi, hogy mennyi az állás jelenlegi értéke, és tárolja azt. Majd generálja a lehetséges lépéseket. Ezek közül kiszűri azokat, amik a megadott mezőre ütnek és elvégzi a legkisebb értékű[13] bábbal való ütést. Ha nincs már ilyen báb, akkor visszatér az aktuális állás értékével. Ezen a ponton lép ki a rekurzióból. Ha van ilyen báb, akkor példányosít az új álláshoz egy `AlphaBeta` osztályt és arra is meghívja a `float utogetes(int)` függvényt. Ezek után, ha olyan szinten van, ahol a világos lép (azaz maximalizál), akkor a saját állás szerinti érték és aközül, amit a gyerekcsúcstól kap a nagyobbat kell feladni. Ha a sötét szintjén van (azaz minimalizál), akkor a kisebb értéket kell feladni a szülőnek. Akkor is a kapott értéket kell feladni, ha csak egy lehetséges lépés van az adott szinten, mivel akkor sakk miatti kényszerítő lépésről van szó.

[13]Itt a következő bábértékeket veszem alapul: gyalog – 1, futó – 4, huszár – 4, bástya – 5, vezér – 9 valamint király – 10. A konkrét értékeknek nincs is jelentősége csak a sorrendnek.

2.6. Brain osztály

Ha az `AlphaBeta` osztály a program „lelke”, akkor ez, mint a neve is mutatja, a program „agya”. Ez az osztály végzi a `Table` osztály `float value()` függvénye számára a neki átadott bemenő paraméterek alapján az állás értékének kiszámolását. Továbbá ez az osztály végzi a játszma végén a parti (állás – lépés párok) alapján a tanulást, ami a bemenő paraméterekhez rendelt súlyok módosítása. Ezen súly-változtatásokat a $TD(\lambda)$ (*temporal difference*, időbeli eltérés) algoritmus alapján végzi.

Most következzen ennek az algoritmusnak a bemutatása, inkább matematikai szempontból, mint programozóiból, a korrektebb specifikálhatóság miatt.

A $TD(\lambda)$ algoritmus [KCAP]

Az összes lehetséges állást jelölje az S halmaz. Tekintsük a játékot úgy, mint lépések sorozatát. Az ágens a t-edik ($t = 1, 2, \ldots$) időpillanatban legyen az $x_t \in S$ állapotban, és A_x halmazban lévő lehetséges lépések közül választhat. Ha egy $a \in A_x$ lépést választ, akkor $P(x_t, x_{t+1}, a)$ valószínűséggel átkerül az x_{t+1} állapotba. Az x_{t+1} állapot az ágens és az ellenfele lépése után van. A játék végén az ágens egy visszajelzést, jutalmat (ez lehet negatív is, ami a büntetést jelöli) kap az eredménytől függően. Ez 1, ha győzött, -1, ha vesztett és 0, ha döntetlent játszott.

Legyen N a játszma hossza, így x_N jelöli az utolsó állást, és $r(x_N)$ jelölje a jutalmat, amit kap a játszma végén. Így a jelenlegi $x \in S$ állapotban a várható jutalom értéke

$$J^*(x) := E_{x_N|x} r(x_N).$$

Itt figyelembe kell venni a $P(x_t, x_{t+1}, a(x_t))$ valószínűségeket is.

S nagy méretére való tekintettel nem lehet eltárolni az összes lehetséges értékét a $J^*(x)$ függvénynek, ezért meg kell próbálni közelíteni egy $\tilde{J} : S \times \mathbb{R}^k \to \mathbb{R}$ függvénnyel. Ezt a $\tilde{J}(\cdot, w)$-ot tételezzük fel parciálisan deriválhatónak a $w = (w_1, w_1, \ldots, w_k)$ paraméterek szerint. A cél az, hogy találjunk egy $w \in \mathbb{R}^k$ paramétervektort, ami minimalizálja az eltérést a $\tilde{J}(\cdot, w)$ és $J^*(\cdot)$ között. A $TD(\lambda)$ algoritmus pontosan ezt csinálja.

Tegyük fel, hogy $x_1, x_2, \ldots, x_N$ egy játék állapotainak a sorozata. Jelöljük az x-edik és $x+1$-edik állapotban várható jutalmak közti különbséget d_t-vel, azaz legyen

$$d_t := \tilde{J}(x_{t+1}, w) - \tilde{J}(x_t, w).$$

Feltehetjük, hogy $\tilde{J}(x_N, w) = r(x_N)$. Így

$$d_{N-1} = \tilde{J}(x_N, w) - \tilde{J}(x_{N-1}, w) = r(x_N) - \tilde{J}(x_{N-1}, w).$$

Az igazi eltérés a két állapotban várható jutalom között természetesen 0.

$$E_{x_{t+1}|x_t}[J^*(x_{t+1}) - J^*(x_t)] = 0.$$

Így a $\tilde{J}(\cdot, w)$ akkor jó közelítése $J^*(\cdot)$-nak, ha $E_{x_{t+1}|x_t} d_t$ közelít a 0-hoz. A játék végén az algoritmus a következő formula szerint frissíti a súlyokat tartalmazó vektort:

$$w := w + \alpha \sum_{t=1}^{N-1} \partial \tilde{J}(x_t, w) \left[\sum_{j=t}^{N-1} \lambda^{j-t} d_t \right],$$

ahol $\partial \tilde{J}(x_t, w)$ a $\tilde{J}$ w paraméterek szerinti parciális derivált vektort jelenti. Az α paraméter a tanulási rátát jelenti, általában 0 és 1 között van, és az idő folyamán közelít a nullához. Ez azért kell, hogy az elején gyorsan, de durván tanuljon, és az idő előrehaladtával ugyan lassabban, de finomabban tanuljon. A $\lambda \in [0, 1]$ paraméter azt befolyásolja, hogy mely időbeli különbségeket terjesszük vissza az időben. Ez annyit tesz, hogy például $\lambda = 0$ esetén a t-edik időpillanatban lévő $\tilde{J}$ értékét a $(t+1)$-edik értékhez közelítjük, $\lambda = 1$ esetén a végső jutalom értékéhez közelítjük.

Megmutatható, hogy a fent bemutatott algoritmus egy közel optimális súlyvektorhoz konvergál.

Adatok

Stack s Ebben a veremben tárolódik, a későbbiekben leírt adatszerkezetben, minden lépésről az alábbi 3 adat: bemenő paraméterek, kimenő érték, lépésszám.

int lepes A numerikus változó tárolja az aktuális lépésszámot. Kezdő értéke 0. Értékét a `float process(float[])` függvény növeli minden meghíváskor 1-el.

int color Értéke $\{-1, 1\}$, ami a szokásos sötét, világos helyett áll. Azért van szükség azt tudni, hogy milyen színnel van, mivel mind sötétnek, mind világosnak külön súlyokat tanul a program. Ez azért van, mert például a program szempontjából nem biztos, hogy világossal is akkora az értéke egy sötét gyalognak, mint ha a sötéttel van.

float[] weights Mint a neve is mutatja, ezen tömb tárolja a súlyokat. Az értékeket a konstruktor tölti be a `void load()` függvény segítségével, majd a `void learn(float)` menti el a `void save()` függvény meghívásával.

int inputs Ezen változóban van tárolva, hogy mennyi bemenő paraméter van. Akkor van rá szükség, amikor a sötét játékos súlyaihoz fér hozzá, mert akkor a súlyok indexét éppen ennyivel kell megnövelni.

Beágyazott osztályok

Rec Ez az előbbiekben (a `Stack s` változó leírásánál) említett adatszerkezet. Belső adatai a következők: `float[] in`, `float out` és `int lepes`. Ezen adatok mind publikusak[14]. Ezenfelül az osztály csak egy konstruktort tartalmaz, amivel ezen adatoknak lehet értéket adni.

[14] Itt is az egyszerűség van a stílusosság elé helyezve, gondolok itt az adatokra való hivatkozásra.

Függvények

Brain(int,int,boolean) A konstruktor. Első paramétere a súlyok száma, második paramétere a $\{1, -1\}$ halmazból kerül ki attól függően, hogy világossal vagy sötéttel vagyunk. A harmadik logikai paraméter igaz, ha applet-ként futtatjuk a programot, különben hamis. Erre azért van szükség, mivel applet-ként nincs írásjogunk a filerendszerhez, és ezért ilyenkor nem olvassuk be a súlyokat tartamazó `brain.dat`-ot és nem is írjuk ki azt. Ilyen esetben minden súly 0, kivételek ezalól a bábértékekhez rendelt súlyok, amik a 28. oldalon lévő 2.3. táblázatban lévő értéket veszik fel.

Ha a `brain.dat` létezik, akkor a program betölti az előző `void load()` függvény segítségével, ha nem létezik, akkor minden súlyt nullára állít, kivéve a bábok értékeit, amit a standard számítógépes értékekre állítja, amik a 28. oldalon lévő 2.3. táblázatban láthatók. Végső lépésként példányosítja a vermet is.

void load() A `brain.dat` file-ból a súlyokat tölti be. Fontos, hogy nem végez ellenőrzést, hogy a file-ban lévő és a tényleges súlyszám megegyezik-e.

void save() A súlyokat menti el a `brain.dat` file-ba.

float process(float[]) Meghívja a `float justProcess(float[])` függvényt a saját bemenő paramétereivel, majd a visszatérési értéket, valamint a lépésszámot és a bemenő paramétereket a `Rec` által megvalósított adatszerkezetbe teszi és bepakolja a verembe. A lépésszámot növeli eggyel, majd a `float justProcess(float[])` által visszaadott értékkel visszatér. Azért van szükség külön `float process(float[])` és `float justProcess(float[])` függvényekre, mert a program, csak a ténylegesen meglépett lépés alapján tanul, nem a lépésgenerálás során összes kipróbált lépés alapján.

float justProcess(float[]) Kiszámolja a bemenetek súlyozott összegét, majd annak a tizedrészének veszi a tangens hiperbólikuszát ($tanh\,(x) = \frac{e^{2x}-1}{e^{2x}+1}$), így az érték a $[-1, 1]$ intervallumba fog esni. Ezután ezzel az értékkel visszatér.

void learn(float) Az előzőekben leírt képlet alapján frissíti a súlyokat. Először kiszámolja a d_x-ek értékét, majd azoknak a segítségével végzi a frissítést. Ha a gép a feketével van, akkor minden súlyra hivatkozó indexet eltol az `int inputs` értékével. Végül a `void save()` függvény segítségével menti a súlyokat.

2.7. Chess2 osztály

Ez a program főosztálya. Ez tartalmazza a `main()` függvényt illetve az `init()` függvényt. Az előbbi akkor hívódik meg, ha parancssorból indítják a programot, a második, ha appletként futtatják böngészőből.

Ezen osztály végzi a grafikus felhasználói felület felépítését, valamint kezelését is. Ezt az előző változatban külön osztály végezte, de mivel a két osztály nagyon össze volt fonódva,

ezért a program jelenlegi változatában összeolvasztásra kerültek, így elkerülhető lett az adatok ide-oda adogatása. Ezzel elveszett annak az esélye, hogy könnyen lehessen a programhoz több felhasználói felületet írni. Az *XBoard* protokoll [XBPROTO] implementálásával nyílt lehetőség ennek a hátránynak a kiküszöbölésére.

Mivel applet-ként is lehet indítani a programot, az osztály a `JApplet` osztály egy leszármazottja, ami maga is az `Applet` osztály leszármazottja. Egyedül az `init()` függvény került definiálásra, ezenfelül minden más változatlan.

Adatok

int vilagos Ez a numerikus változó tárolja azt, hogy a világos színnel a felhasználó, vagy a számítógép van-e. Az értékeit a $\{0, 1\}$ halmazból veszi fel. A 0 jelentése az, hogy a világos bábokat a felhasználó mozgatja, az 1 jelentése, hogy ezt a számítógép teszi.

int sotet Mint az előbb az **int vilagos**-nál, csak itt a sötét bábokról van szó.

int mode Azt tárolja, hogy a felhasználói felület milyen állapotban van. Ez lehet: 0 – játék, 1 – szerkesztés. A módot a felhasználói felületen lévő *Játék* illetve *Szerkesztés* gombokkal lehet váltani (ezek közül mindig csak az ellentétes mód gombja látható).

A módtól függően a felhasználó által elérhető kezelőszervek változnak.

int promotion Ez változó adja meg, hogy milyen bábot kap a játékos az átváltozott gyalogért cserébe. Az értékei a következők lehetnek: 2 – bástya, 3 – huszár, 4 – futó, illetve 5 – vezér. Alapesetben az értéke 5 – vezér. A változó értékét Játék módban lehet váltogatni egy választógombbal.

int piececolor A szerkesztés módban a táblára kerülő báb színét adja meg, lehetséges értékei: 1 – világos, -1 – sötét. Az értékét egy legördülő menüvel lehet szabályozni, ha Szerkesztés módban vagyunk.

int piecetype A szerkesztés módban a táblára kerülő báb típusát adja meg, az értékek a következők lehetnek: 1 – gyalog, 2 – bástya, 3 – huszár, 4 – futó, 5 – vezér, illetve 6 – király.

Image[] img A tábla kirajzolásához szükséges képeket tárolja benne a program. A képek betöltését a `void init()` függvény végzi el.

Ezeken felül már csak a különböző alosztályok elérésére használt hivatkozások vannak adatként az osztályban, amelyeknek felsorolását mellőzöm, mivel nincs sok jelentőségük.

Beágyazott osztályok

BoardPanel Ez a játékteret megvalósító osztály, a `JPanel` leszármazottja. Két adatot tartalmaz csupán, amelyek egy honnan – hova párt alkotnak. Ezek mondják meg, hogy mely mezőket kell kéken bekeretezni, szemléltetve azt, hogy mi volt az utolsó

lépés. Az osztály csak három függvénnyel rendelkezik, ezek közül az első kettő (`void markFrom(int), void markTo(int)`) a fenti két változó beállítását végzi, a harmadik (`void paingComponent(Graphics)` a tábla kirajzolását. Ez utóbbi a `JPanel`-től örökölt függvény felüldefiniálása, így a szokásos `reapint()` függvény meghívásakor ez a függvény kerül meghívásra, és rajzolja ki a táblát. Ez átlátszó hátterű `.gif` fileok segítségével van megoldva, amiket a Java képes egymás tetejére helyezés után helyesen megjeleníteni.

bpanelListener A játéktér eseményeire figyelő osztály. A `MouseListener` sablon implementációja. Egyedül a `void mouseClicked (MouseEvent)` függvény van megvalósítva, így figyelve azt, hogy a felhasználó kijelöli a mozgatandó bábot és a mozgás célmezőjét. Amikor a felhasználó megjelöl egy mezőt, azt a függvény egy `Move` osztály példányát használva tárolja el. Ha még nem volt beállítva a kezdeti mező, akkor azt beállítja. Ha be volt állítva a kezdeti mező és a felhasználó megint azt jelöli meg, akkor törli a mező kijelölését. Ha már be volt jelölve e kezdeti mező és mást jelöl meg a felhasználó, akkor az a „hova" mezőt jelenti, és beállítja azt is. Ezek után megnézi, hogy gyaloggal léptünk-e (ezt a `Table` osztálytól kérdezi meg a `int getPiece(int)` függvény segítségével) és, hogy az utolsó sorra léptünk-e. Ha igen, akkor beállítja az átváltozás után a táblára felkerülő bábot is. Ezek után ellenőrzi, hogy legális-e a lépés. Ezt a `Table` osztály `boolean validateMove(Move)` függvényével teszi. Ha igazat kap vissza, akkor elvégezteti a lépést a `Table` osztállyal, valamint beállítja a bekeretezendő mezőket a játéktéren, és újrarajzoltatja azt.

AIThread A `Thread` osztály egy leszármazottja, így ez egy külön programszálat képvisel. A szálat az `void init()` függvény indítja el. A feladata az, hogy végtelen ideig fut, és tizedmásodpercenként felébredve megnézi, hogy játék módban vagyunk-e, valamint, hogy a számítógép következik-e lépéssel. Ha igen, akkor generáltat egy lépést az `AlphaBeta` osztállyal és végre is hajtja azt a táblával. Azt is ellenőrzi ezek után, hogy matt, esetleg patt lett-e a játszma vége. Ha igen, akkor ezt kiírja, ellenkező esetben csak a lépésgenerálás statisztikáit írja ki.

ExitListener A `WindowAdapter` leszármazottja. Egy `void windowClosing(WindowEvent)` függvényt tartalmaz, ami akkor hívódik meg, ha a felhasználó az ablakzáró ikonra klikkel. Ha ez megtörténik, akkor kilép a programból.

PromoteListener Az `ActionListener` osztály leszármazottja. Egy függvényt tartalmaz. Ez a `void actionPerformed(ActionEvent)`. Akkor hívódik meg, ha a felhasználó a felületen átállította, hogy milyen bábot kapjon az átváltozott gyalogért cserébe. A függvény egyedül az ezt leíró változót (`int promotion`) állítja be.

PieceColorListener Hasonlóan az előző függvényhez, ez is az `ActionListener` osztály leszármazottja. Ha szerkesztés módban változtatjuk a bábszínt, akkor hívódik meg és állítja be az `int piececolor` változót.

WhiteCastleListener Ez is az `ActionListener` osztály leszármazottja, amelynek az egyedüli `void actionPerformed(ActionEvent)` függvénye akkor hívódik meg, ha az állásszerkesztő nézetben a felhasználó a fehér sáncolását beállító legördülőmenün állít. Ilyenkor lekéri a táblától a sáncolást leíró numerikus változót (a `int getCastle()` függvénnyel), és attól függően, hogy mit állított a felhasználó, megváltoztatja az alsó két bitjét, majd visszaküldi a táblának a `void setCastle(int)` függvény segítségével.

BlackCastleListener Mint a `WhiteCastleListener` csak a sötét bábokra alkalmazva..

PlayerColorListener Ezen osztálynak a `void actionPerformed(ActionEvent)` függvénye, akkor hívódik meg, ha a Szerkesztés képernyőn a felhasználó a soron következő játékost beállító legördülőmenü állapotán változtat. Ekkor a `void setNext(int)` függvény segítségével beállítja ezt a `Table` osztályban.

WhiteSideListener Ha a világos színű bábok mozgatóját átállítja a Szerkesztés képernyőn a felhasználó, akkor hívódik meg a `void actionPerformed(ActionEvent)` függvénye ennek az osztálynak, ami az `int vilagos` változó értékét állítja át. Értékei a következők lehetnek: 0 – felhasználó, 1 – számítógép a fehér.

BlackSideListener Mint a `WhiteSideListener`, csak az `int sotet` változó értékét állítja be.

EnpassantListener Annak a mezőnek a beállítására szolgáló legördülő menü megváltozása esetén hívódik meg, amelyet az „en passant" lépés céljaként jelöl meg a felhasználó. Az érték átadására a táblának a `void setEnpassant(int)` függvényét használja.

PieceListener A felhelyezni kívánt báb típusát választó legördülő menü megváltozásakor hívódik meg a `void actionPerformed(ActionEvent)` függvénye. Feladata az `int piecetype` változó beállítása, valamint a bábszínt kiválasztó legördülő menü kattinthatóságának az állítása. Ezen utóbbira akkor van szükség, ha a bábtípusnak „Üres"-et választott a felhasználó, ilyenkor letiltja a bábszínt választó menüt.

EditListener Csak `void actionPerformed(ActionEvent)` metódusa van az osztálynak, mint az előbbiekben látható minden olyan alosztálynak, ami egy eseményt figyel. Akkor hívódik meg, ha a Szerkesztés nyomógombra kattintunk. Első lépésként átállítja az `int mode` változó értékét 1-re, ezzel jelezve, hogy váltás történt. Ezek után lekéri a táblától a sáncolási lehetőségeket tároló változó értékét, majd ennek megfelelően beállítja a legördülő menük állapotát. Ugyanezt elvégzi az „en passant" lépést mutató legördülő menüvel is. Ezután a szükségtelen gombokat és egyéb felületi elemeket eltünteti, és a szükségeseket megjeleníti.

PlayListener Hasonló, mint az `EditListener`, de annál egy kicsit több feladatot valósít meg. A legfontosabb ezek közül, hogy ellenőrzi a felpakolt állás érvényességét, és addig nem tér vissza a Játék módba, amíg az nem legális. A következőket ellenőrzi:

1. Legyen két ellentétes színű király a táblán.
2. A két király ne álljon egymás mellett, se szomszédos, se átlós mezőn.
3. Az utolsó és első sorban ne álljon gyalog.
4. Csak olyan mezőre mutasson az „en passant" ami alatt illetve felett (a színtől függően) áll gyalog, amitől a dupla lépés származhatott.
5. Csak akkor lehessen sáncolni, ha a megfelelő bábok azokon a mezőkön állnak, ahol kell.

ClearListener A Játék módban látható *Alapállás* nyomógomb megnyomására hívódik meg ennek az osztálynak a `void actionPerformed(ActionEvent)` függvénye. Inicializálja a táblát a `void init()` függvény segítségével. Továbbá törli az esetleges jelöléseket a táblán, amelyek az utolsó lépés honnan és hova mezőjét jelölték.

Függvények

void init() Ez a függvény végzi a felhasználói felület felépítését, valamint a program azon szálának elindítását, ami a számítógép gondolkodásáért a felelős. Ezt egy külön program szál végzi azért, hogy amikor elindításra kerül a számítógép lépésgenerálása, akkor ne „fagyjon le" az egész program addig, amíg ki nincs számolva a lépés.

Ez a függvény végzi a különféle programkomponensek inicializálását is (tábla példányosítását, alapállásba állítását; a lépést leíró `Move` osztály példányosítását), valamint a felülethez szükséges képek betöltését is.

void main(String[]) Akkor hívódik meg amikor a programot parancssorból indítjuk. Nincs is más feladata mint, hogy példányosít egy `JApplet` osztályt saját magából, beállítja az ablak méretét, majd meghívja a `void init()` függvényét. Innentől nem különbözik az applet-ként való, valamint a parancssori indítás.

2.8. ChessXboard osztály

A másik főprogram, amely *XBoard* protokoll [XBPROTO] kommunikációra teszi képessé a programot. Főként tesztelési célokkal jött létre, mivel ennek segítségével, valamint egy másik programot (Henrik Oesterlund Gram által írt icsDrone) használva interfésznek, képes az internetes játékra. A segítségével arra is lehetőség van, hogy más felhasználói felülettel használjuk a programot, mint például az *XBoard* illetve az *eboard*. Így lehetőség nyílik arra, hogy más XBoard kompatibilis programok ellen játszhasson..

A protokollból az összes kötelező parancs meg van valósítva és egy-két opcionális is. A parancsok listája, amit érdemien kezel a program a következő: xboard, protover, new, quit, force, go, move, playother, white, black, time, usermove, ping, draw, edit, hint, bk. Ezenfelül még több parancsot is elfogad, de azokat figyelmen kívül hagyja. Itt megjegyzem, hogy ez nem baj, mert vannak olyan parancsok, amikre nem kell válaszolni, hanem csak a program működésén kell változtatni, és ezek olyan működésbeli változások, amiket a program felszólítás nélkül is megcsinál.

Adatok

int sd A keresési mélységet tárolja ebben a numerikus változóban a program, amelynek az értéke a még megmaradt gondolkodási időtől függ.

Table table A táblára lehet hivatkozni vele. A `Move stringToMove(String)` fügvénynek, ami egy karakterláncként megadott lépést alakít át egy `Move` osztály által leírt lépésre, van rá szüksége. Mivel a sáncolást jelölő utasítás nem hordoz információt arról, hogy melyik játékos tette azt meg („O-O-O” illetve „O-O”), így a táblától kérdezi meg a következő játékos színét.

Függvények

Move stringToMove(String) Az átadott karakterláncból példányosít egy `Move` osztályt. Az átadott karakterláncok a következő tipikus formákat vehetik fel: „e2e4”, „h7h8q”, „O-O”, „O-O-O”, illetve egy-két elfajzott változata az utóbbi kettőnek: „OOO”, „OO”, „o”, „o-”. Ezen utóbbiakra azért van szükség mert az internetes szerver és a program között interfészt megvalósító külső program néha helytelenül a fenti formában adja át a sáncolásra vonatkozó lépéseket.

void main(String[]) A konkrét főprogram. A standard bemenetről fogad karakterláncokat, azokat feldolgozva a standard kimenetre előállítja a program válaszát: speciális válaszokat illetve hibaüzeneteket.

A tényleges működése a következő: példányosítja a szükséges osztályokat valamint inicializálja a változókat. Ezek után egy végtelen ciklusba lép be, ami a következőket csinálja: olvas egy sort a bemenetről, majd ezt összehasonlítja azokkal az utasításokkal amiket megért. Ha egyezést talál akkor az annak megfelelő programrészletet végrehajtja. Ezek a következők:

new Inicializálja a táblát, beállítja, hogy a számítógép a sötét bábokat mozgatja. Ezt egy numerikus változó tárolja (`int computer`). Újrapéldányosít egy `Brain` osztályt a sötéttel, mint a számítógép színével, majd ezt beállítja a `Table` osztálynak a `void setBrain(int)` függvénye segítségével.

go A parancs arra szolgál, hogy bármelyik szín is következik, a számítógép azzal kezd el játszani. Első lépésnek beállítja, hogy a számítógép a következő színnel van. Ezt a táblától az `int next()` függvény segítségével tudja meg. Itt is új `Brain` osztályt példányosít és a táblának is beállítja azt. Ezután létrehoz egy új `AlphaBeta` osztályt és annak a `Move general2()` függvénye segítségével generál egy lépést. A számítógép meglépi azt és tanul is belőle, mert meghívja rá a `Table void learnFromMove()` függvényét. Legvégül kiírja a standard kimenetre a lépést, ezzel átadva azt az interfésznek.

playother A parancs azt jelenti, hogy a bármelyik játékos is van soron lépéssel, az lesz a felhasználó, és a másik a számítógép. Az `int computer` változót beállítja arra az értékre, ami nem a most soron következő játékost írja le.

white Azt jelenti, hogy a lépésre következő játékos a világos és a számítógép a sötéttel van. Ennek megfelelően újra példányosít egy `Brain` osztályt és átadja azt a `Table` osztálynak. Beállítja a `Table` osztályt, úgy hogy a világos jön lépésre. Majd az `int computer` változót is megváltoztatja -1-re.

black Mint az előző parancs, csak a színek megcserélődnek.

time TIME A program hátralévő idejét adja meg századmásodpercben a segítségével az interfész. Ha több, mint 10 perc van még akkor a keresési mélységet (`int sd`) 6-ra állítja, ilyenkor körülbelül 3 perc alatt lép egyet a program. (A teszteléskor használt 2 GHz-es processzorral felszerelt gépen.) Ha 2 és 10 perc közötti idő áll a rendelkezésére akkor 5-ös keresési mélységet használ. Ekkor egy lépés körülbelül 40 másodperc. Ha 2 percnél kevesebb idő van hátra, akkor 4-es mélységet állít be, ekkor egy lépés körülbelül 1-2 másodperc.

usermove MOVE A megkapott lépést ellenőrzi, hogy legális-e és ha igen végrehajtja.

ping N Egy „pong N”-el válaszol erre a program. Az interfészek szokták ezzel ellenőrizni, hogy fut-e még a program.

draw Az ellenfél ennek segítségével ajánl döntetlent. Ezt a program az „offerdraw” utasítással elfogadja, amennyiben a jelenlegi állás értéke az ellenfélnek kedvez.

result X-X Az interfész így tudatja, hogy mi lett a parti végeredménye. A program ellenőrzi, hogy a számítógép időtúllépéssel vagy tényleges mattal nyert illetve vesztett-e. Ettől függően felparaméterezve hívja meg a `Brain` `void learn(float)` függvényét. Ezek lehetnek: 0 – patt; 1 – világossal nyert, illetve sötéttel kikapott; -1 – sötéttel nyert, illetve világossal kikapott; valamint bármilyen kettő közötti érték, ha idővel nyert. Ez utóbbi esetben az adott állás értékét adjuk át. Ennek az az oka, hogy nem lenne jó, ha rosszat tanulna a program abból, ha teljesen vesztett állással nyert úgy, hogy az ellenfélnek lejárt az ideje.

edit Egy logikai változót átbillent igazra, ami azt jelöli, hogy állásszerkesztés folyik. Ekkor különleges parancsokat kell elfogadnia, amikkel az állást lehet beállítani.

hint Tanácsot lehet kérni a programtól. Válaszként megadja, hogy mit lépne.

Ha szerkesztés mód van, akkor a következő parancsokat fogadja el:

.	Kilépés a szerkesztés módból.
#	Kiüríti a táblát.
c	A felhelyezendő báb színét változtatja ellenkezőjére, alapesetben ez világos.

Továbbá, ha ebben a módban van, és olyan 3 karakter hosszú utasításokat kap, amelyeknek első karaktere egy betű „a” és „h” között, második karaktere „1” és „8” közti szám, harmadik karaktere a következő lehet: „x”, „P”, „B”, „N”, „R”, „Q”, „K” (az „x” jelentése az üres mező), akkor a megfelelő bábot teszi a megjelölt mezőre.

Ha nem szerkesztés mód van, és a fentiektől különböző parancsot kap és azt sikeresen át tudjuk alakítani a `Move stringToMove(String)` függvénnyel, akkor az ellenfél lépését kapta meg. Ilyenkor elvégzi azt. Ezek után, ha a számítógép következik (lehet, hogy nem, mivel ha a `int computer` változó értéke 0, ilyenkor nincs egyik színnel sem), akkor generál egy lépést és meglépi a szokásos módon. Végül ellenőrzi, hogy vége van-e a partinak. Ha igen, akkor azt kiírja, mint például: „0-1 {Black mates}”.

3. fejezet

Tesztelés

A tesztelést több fázisban és több módon is végeztem. A program kezdeti szakaszaiban általában saját magam teszteltem úgy, hogy játszottam a program ellen. Ha hibára bukkantam, akkor célzottan a hibás helyzetet tartalmazó állásokat elemeztettem vele, hogy kiderüljön a hiba oka. A kezdeti hibák kijavítása után kitettem a programot az internetre és az ismerőseim segítségével teszteltem. Így is sikerült egy-két hibát kiküszöbölnöm. A végső szakaszban „bedobtam a mélyvízbe", és a http://freechess.org-on lévő ingyenes internetes sakkszerveren játszattam a programot, tesztelve a játékerejét és a tanulási képességét. Itt is sok érdekes tapasztalattal lettem gazdagabb. A későbbiekben ismertetem a tapasztalt hibákat.

1. Korai hibák

A legtöbb korai hiba a lépésgenerálásban és a sakkellenőrzésben jelentkezett. Mivel a táblát egy egydimenziós tömb reprezentálja, ahol a mezők sorfolytonosan helyezkednek el, így a horizontális és átlós lépéseknél osztási maradék segítségével figyelem, nehogy „lecsússzon" a báb a tábla széléről. A korai hibák ebből adódtak, mivel többször is előfordult, hogy a futó vagy a vezér átlósan átcsúszott a tábla ellenkező oldalára, vagy hasonló módon adott sakkot. Természetesen ezt a program helyesnek gondolta, és támaszkodott is rá, sokszor adott mattot a fenti módon.

Az alfabéta keresésben is adódtak hibák, de ezek csak a rosszul megválasztott relációs jeleknek voltak köszönhetőek. Voltak olyan időszakok, amikor egy-egy ilyen elírás miatt nagyon zseniálisan játszott a *francia sakkhoz* hasonló játékot, azaz olyan gyorsan kapott ki, ahogy csak tudott.

2. Későbbi hibák

Az `AlphaBeta` osztály leírásánál említett *patt* hiba is nagy fejtörést okozott egy időben. A pattra a kiértékelés olyan értéket adott vissza, mintha matt lett volna, ezzel próbálván elkerülni a pattot. Ez nemvárt eredményt hozott, mivel a program emiatt gyakran adott

pattot, mivel az ő szemszögéből az ellenfélnek a patt ugyanolyan rossz volt, mint a matt. Ezt úgy küszöböltem ki, hogy a kiértékelés a pattra olyan értéket ad vissza, mintha nyert volna az illető játékos, mivel ha pattot kap valaki az neki jó, mert úgyis rosszul állt.

Még többször bajlódtam az alfabéta kereséssel. Volt olyan eset, hogy ha a gép látta, hogy biztosan mattot fog kapni, bármit is lép, akkor nem adott vissza legális lépést. Ezt úgy lehetett orvosolni, hogy a kiértékelésnél a legjobb lépés kezdőértékét extremálisra állítottam, hogy annál még a matt is jobb legyen. Bár ezt fel lehetett volna úgy is használni, hogy a gép feladja ilyen esetekben a partit, de ezt azért nem teszi, mert nem biztos, hogy az ellenfél is tudja, hogy hogyan kell beadni a mattot. Mivel az időnek is szerepe van, főleg az interneten játszott partikban. Ha az ellenfél kifut az időből, miközben gondolkodik azon, hogyan adjon mattot, az is szabályos győzelem, bár nem olyan elegáns, mintha normálisan nyert volna.

3. Az internetes tesztelés

Az internetes teszteléshez implementáltam az *XBoard* protokollt [XBPROTO], egy alternatív főosztálynak a programhoz és az így elkészült programot Henrik Oesterlund Gram által írt icsDrone program segítségével kapcsoltam a freechess.org-hoz. (A program *veeharom* azonosítóval játszik a szerveren.)

Természetesen itt sem ment minden zökkenőmentesen, mivel néha nem a protokoll szerint kapta meg a program a lépéseket. Főként a sáncolás okozott galibát, de miután megtaláltam a hibát, máris tudott játszani. Már csak az idő volt a probléma. A programot `10 0` idővel játszattam, ami azt jelenti, hogy 10 perc van gondolkodásra és nincs lépésenként időnövekedés. A program az elején egész jól is boldogult, de ahogy kezdett tanulni, a lépésgenerálás is lassult, mivel az egyes állások közt a különbség kezdett árnyaltabb lenni. A probléma megoldására a következő dolgot találtam ki. Ha még több, mint 10 perce van (erre azért van szükség mert a kihívásokat elfogadja a program és akkor nem csak az nekem tetsző idő szerint játszik, hanem az ellenfél által választott idő szerint, ami lehet több, mint 10 perc), akkor 6 mélységig generálja a fát, ha 2 és 10 perc közti ideje van, akkor 5 mélységig keres, ha annál kevesebb akkor csak 4-ig. Így elértem, hogy szinte sohasem fut ki az időből. Sajnálatos tényként jegyzem meg, hogy amíg az 5 mélységű keresés kb. 30 másodpercig tart, a 6 mélységű, van úgy, hogy több, mint 4 percig. Így, ha a gép pont a 10 percbe lépés előtt kezd el gondolkodni, valószínüleg időzavarba kerül a végén.

Most pár szó a program eredményeiről. Az első internetes tesztelés alkalmával a következő eredményt érte el a program: amikor még nem volt bekapcsolva a tanulás, akkor körülbelül 1150 ELO pontos[1] volt. Amikor bekapcsoltam a tanulást, akkor körülbelül 300 parti alatt 1351 pontig tornászta fel magát. Ekkor sajnálatos módon valamit elrontottam a programban és megint visszaesett, majd javítás után megint nőtt, de aztán megint visszaesett. A második tesztelés alkalmával `5 0` idővel játszattam a programot, a gyorsabb eredmény érdekében. Ez alkalommal 1070 pontról indítottam, teljesen új súlyvektorral.

[1]Ez nem ugyanaz az ELO, mint amit a FIDE használ, a legjobb a szerveren körülbelül 2800 körüli és a legrosszabb, amit láttam, 500 körül volt.

Az elért ELO pontnövekedést a 3.1. ábrán lehet látni, a piros körök az ellenfelek erejét jelölik. A legnagyobb megfigyelt pontérték ezen esetben 1392 volt. Jelenleg több, mint 1200 lejátszott partin van túl.

Az internetes teszteléssel kapcsolatban több tapasztalatom is van. Először is az biztos, hogy jobb, mint saját maga ellen vagy bármiféle más program ellen játszatni. Ennek az az oka, hogy nem korcsosul el a tudása, azaz nem csak egy bizonyos ellenfél ellen tud majd jól játszani, kihasználva annak a típushibáit, hanem a nagy többség ellen is megtanul játszani. Bár az interneten is volt olyan eset, hogy egy ember egymásután tizenötször hívta[2] ki és kapott ki tőle, utána láthatóan sok értelmetlen lépést tett a program. A másik baj az, hogy általában a gyengébbek játszottak vele[3], és az ő megverésükkel nem sokat tanult, valamint csak kevés pontot kapott. Ezután, ha összekerült egy nála erősebbel, vagy hasonló erősségűvel és kikapott tőle, akkor meg sokat esett vissza a pontszáma. Így eléggé lassan tudott pontokat gyűjteni.

Az interneten általában magyar idő szerint reggel 8 és este 10 között játszott. Lehet, hogy jobb lett volna, ha állandóan fut, mert akkor több emberrel hozhatta volna össze a sors. Ezt sajnos nem állt módomban megtenni.

4. Híres partikkal való tesztelés

A játékerősséget úgy is teszteltem, hogy feladványokat, híres partikból vett állásokat adtam be a programnak, hogy megnézzem, megtalálja-e azt a lépést, amit a nagymesterek léptek. Az itt következő példákból nem akarok messzemenő következtetéseket levonni, mivel nem tudok annyira sakkozni, inkább csak érdekességképpen vannak itt, valamint azok számára, akik elég jól tudnak sakkozni és látják a lépések mögött rejlő gondolatokat.

4.1. Első példa

A 3.2. ábrán látható állás Conrad Bayer és Ernst Falkbeer 1852-ben Bécsben játszott partijából való, világos utolsó lépése **Vxa8** volt, amivel egy sötét bástyát ütött. Az állást kiértékeltettem a programmal és 6-os keresési mélységgel 35, 147 másodperces gondolkodás, és 254.821 állás megvizsgálása után, miközben a fában legmélyebben 11 lépést tett meg a nyugalmi helyzet keresése érdekében. A **He2†** lépést találta meg, pont azt amit Falkbeer,

[2] A második internetes tesztelés alkamával sikerült megoldanom, hogy a program ne engedje, hogy egymás után 2-nél többször ugyanaz az ember fogadja el a kihívását. Bár néha így is volt, aki többször egymás után játszott vele, de ez nem vitte annyira tévútra a programot.

[3] Amíg nem volt számítógépként regisztrálva a szerveren, csak úgy, mintha élő ember lett volna, akkor az erősebbek is előszeretettel játszottak vele. Sajnos ez nem volt legális, így kitiltottak a szerverről. Amikor több hetes hercehurca után sikerült most már számítógépként regisztrálnom magamat, akkor már általában csak a gyengébbek játszottak a program ellen. Ennek szerintem az az oka, hogy a gyengébb sakkprogramokat meg lehet verni, ha lecseréltetünk vele minden tisztet, mivel a végjátékban általában az emberek az ügyesebbek. Ezt tudva, a gyorsabb fejlődés érdekében inkább a kezdők játszanak a programok ellen. A másik ok az lehet, hogy a programok néha lépnek „hülyeségeket" és a jobb játékosok ezt nem tolerálják annyira, mint a gyengébbek.

majd mattot adott világosnak, mivel a világosnak el kell lépnie a sakkból. Ezek után sötét **Vxh2†**, a sakkot világos, csak úgy tudja védeni, ha a királya üti a vezért, **Kxh2**, de ezek után **Bh4‡** mattot ad.

4.2. Második példa [BCPS]

A 3.3. ábrán látható feladványt F. Matousek alkotta 1936-ban. A feladat: fehér lép és 3 lépésben[4] mattot ad.

A feladvány kulcslépése az **Fc5**, mivel ha ezek után sötét üti a d2 bástyát, **Kxd2** akkor világos **Vh6†** lépéssel sakkot ad, ezek után sötétnek két lehetséges lépése van: **Kc3**, illetve **Ke1**, azért, hogy elodázza a sakkot, de mindkét esetben **Vc1‡** lépése matt. Ha **Fc5** lépés után sötét **Kf1**-et lépi, akkor **Vh1‡** matt. Ha nem a királyával lép, akkor vagy az e2 gyaloggal vagy bármi mással léphet, ha az e2 gyaloggal, akkor **Vc1‡** matt, ha az e2 gyalog a helyén marad, akkor a **Vc1** után még elléphet a sakkból **Kf2**-vel, de ezek után **Hd1‡** ad mattot.

A fenti matthoz a kulcslépést a program 67, 895 másodperc alatt 578.376 állás megvizsgálása után találta meg, majd a fent leírtak szerint mattot adott[5].

4.3. Harmadik példa [BCPS]

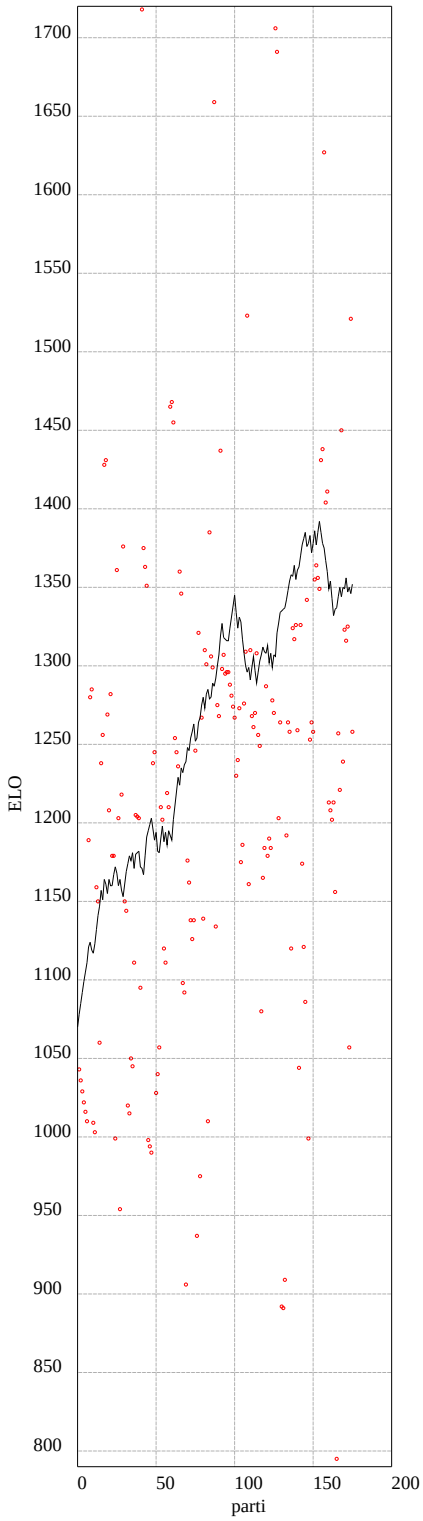
A 3.4. ábrán látható problémát I. A. Schiffmann alkotta 1929-ben. A feladat világossal mattot adni 3 lépésben.

A kulcslépés a **g6** gyaloglépés, amivel az **xf7**, majd **Bxd8‡** lépést készíti elő. Világos próbálhat ellenállni úgy, hogy ellép a bástyával, hogy majd a vezérével h4-re léphessen. Ha sötét a **Bd4**-et lépi, akkor világos **Ka5**-öt lépi, hogy majd a **Ha6‡** lépéssel mattot adhasson, mivel a d2 bástya akadályozza a **Ve1** sakkot. Ha azonban sötét a nem **Bd2**-et, hanem **Be2**-őt lépi, akkor világos **Kxb5**-öt lépi és így megint nem tud sakkot kapni, **Vf1** lépéstől. Míg, ha a sötét a **Bf2**-t lépi, akkor világos a **b7** gyaloglépése a helyes válasz, amit a **Fa7‡** fog követni. Utolsó lehetőség, ha sötét **Bg2**-t lépi, ekkor **Fb7** a helyes válasz, amit a **Hc6‡** huszárlépés követ, mivel ha fekete el is lépne a d5-ön álló gyaloggal akkor is útban van a g2 bástya.

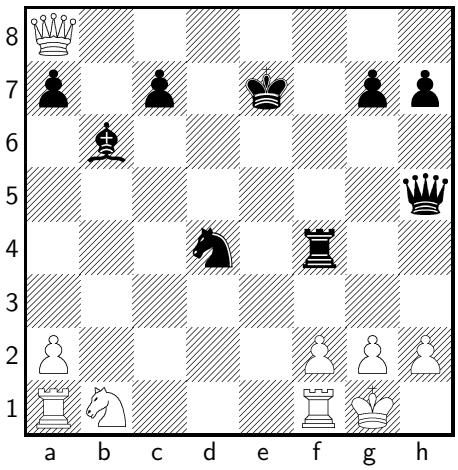
A fenti megoldást a program sikeresen megtalálja és végigjátssza. A kulcslépést 69, 92 másodperc alatt 482.227 állás megvizsgálása után találta meg.

[4]Itt ez a hétköznapi értelemben vett lépést (lépésváltást) jelenti.

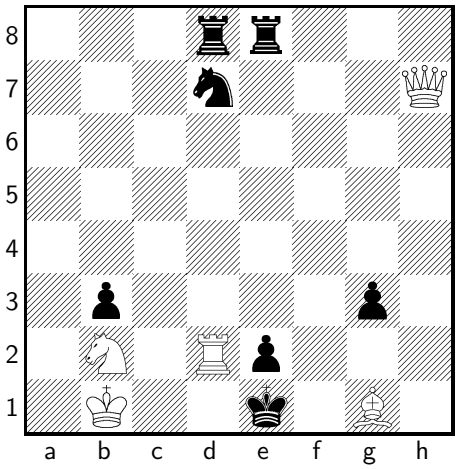
[5]Itt megjegyzem, hogy bár a program megoldja ezen feladványokat, de nem optimálisan jár el, mivel azt nem lehet beállítani, hogy hány lépésben adjon mattot, így a keresés folyamán a megadott korlátnál mélyebben is nézi a játékfát.



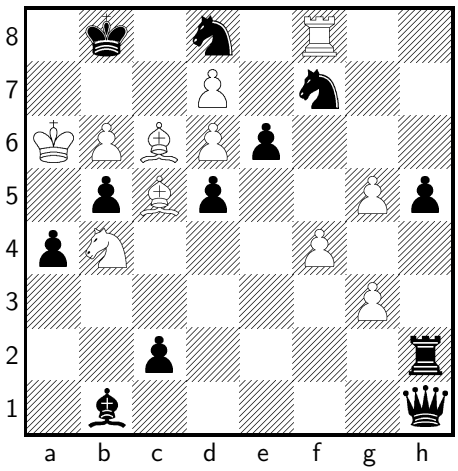
3.1. ábra: ELO pont grafikon



3.2. ábra: Első példa



3.3. ábra: Második példa



3.4. ábra: Harmadik példa

4. fejezet

Továbbfejlesztési lehetőségek

Ebben a fejezetben leírom, hogyan lehet a programot a továbbiakban fejleszteni, a játékerősségét javítani.

1. Játékerő javítás

Mivel a program képességeinek két dolog szab határt, ezeknek a fejlesztésére kell a legfőbb hangsúlyt helyezni. Ezek közül az első a számítási kapacitás, a második a kiértékelőfüggvény minősége. Mint már bebizonyosodott, elég a számítási kapacitást növelni ahhoz, hogy egy sakkprogram legyőzze a világbajnokot, de sajnos ez a normál földi halandók számára nem megoldható, ezért inkább a másik irányt kell javítani. Ennek ellenére megemlítek egy olyan módszert is, amivel a számítási kapacitás növelhető oly módon, amire még a hétköznapi embereknek is lehetősége nyílik.

1.1. Osztott számítás több processzor segítségével

Mint a bevezetőben is említettem, van mód költségtakarékos módon is növelni a program számítási kapacitását. Ez véleményem szerint a PVM [PVM], valamint az MPI [MPI] program könyvtárak filozófiáját követve oldható meg a legkönnyebben. Ez úgy valósítható meg, hogy a programot kibővítjük főprogram és csak a számítást végző segédprogramokra. A Internet Protokoll segítségével megoldható, hogy ezen programok különböző gépeken futva egymással együttműködve tudják a program számítási teljesítményét növelni. Olyan módon téve mindezt, hogy a keresési fát valamely gyökér közeli csúcsnál szétvágva, az egyes részfákat különböző gépeknek adva értékeltessük ki azokat.

Mivel a részfa kiértékelése nagyon hosszadalmas feladat, ezért a kommunikáció elhanyagolható időnek tekinthető, főleg, hogy egy versenypartiban egy lépésre átlagosan 3 perc áll a játékos rendelkezésére. Ezen okoknál fogva, nem csak helyi hálózatokat, hanem az Interneten szétszórt gépeket is lehet erre használni. Mivel a program Java nyelven íródott, még az sem szab határt, hogy melyik gépen milyen operációs rendszer fut.

Itt megjegyzem, hogy már van olyan kezdeményezés, ami a fenti ötleten alapul: a neve

ChessBrain. A http://www.chessbrain.net címen található meg az oldala, bár a program még nem játszik jól, de a fenti módon megvalósított osztott számítási módszerrel már Guiness rekordot ért el abban, hogy hány számítógép gondolkozik egyszerre egy sakkpartin. Szerintem ez a megoldás a leginkább eredménnyel kecsegtető, főleg, hogy a mai világban egyre nagyobb sávszélesség és megbízhatóbb internet áll az emberek rendelkezésére.

Egyedüli probléma az lehet, hogy egy részfát a többinél lényegesen lassúbb gép kap meg kiértékelésre, és a főprogram bár már rajta kívül minden klienstől megkapta a kiértékelés eredményét, így kénytelen mégis várni. Erre több lehetséges megoldás is kínálkozik. Például, hogy minden részfát több kliensnek küldünk el kiértékelésre, ezzel a redundanciával csökkentve a fenti probléma előfordulásának az esélyét. Másik megoldás lehet, hogy a program indításakor minden klienssel *teszt kiértékelést* hajtatunk végre, hogy megmérhessük, hogy ki milyen gyors, és csak a leggyorsabb klienseket használjuk. Az első megoldás ellenére is előjöhet a fenti probléma, míg a másik sem küszöböli ki azt teljesen, mert mi van, ha valaki kikapcsolódik közben, vagy elindít egy processzort leterhelő számítási folyamatot és így a nekünk kellő számítás kerül hátrányba.

Szóval a fent említett megoldásnak bár vannak hibái és buktatói, de szerintem a jövő ebben van.

1.2. Több kiértékelő függvény

Ha a számítási kapacitást már nem tudjuk tovább növelni, lehet a kiértékelő függvényt *okosítani*. Ezt a dolgot általában úgy szokás megoldani, hogy a játékmenetet 3 részre osztják: megnyitás, középjáték, végjáték, valamint minden játékrészhez különböző kiértékelőfüggvényeket alkalmaznak.

Ennek az az előnye, hogy egy függvénnyel nehézkes leírni az egész játékot, mivel a kezdésnél a legfontosabb, hogy minél inkább kinyíljunk és a táblát minél jobban az irányításunk alá vonjuk. Ilyenkor még nem érdemes olyan aspektusokat belevenni a kiértékelő függvénybe, mint az ellenséges király által lehetséges lépésszám. Annak majd csak a végjátékban, a mattadásnál van jelentősége inkább.

A kiértékelőfüggvények meghatározása már egy másik lapra tartozik. Felhasználhatjuk saját, valamint nálunk jobb sakkozók ötleteit, de akár más programok által figyelt értékeket is használhatunk.

Tegyük fel, hogy megvannak a kiértékelő függvények amiket alkalmazni akartunk. A kérdés már csak az, hogy hol vannak a határok a játékrészek között. Erre a legelterjedtebb megoldás az, hogy a megnyitást és a középjátékot egy meghatározott lépésszám határolja, a középjátékot és a végjátékot egy meghatározott bábérték alá csökkenés jelzi. Ezeknek az értékeknek meghatározására nincs jobb megoldás, mint a próbálkozás.

Irodalomjegyzék

[BCPS] British Chess Problem Society, http://www.bcps.knightsfield.co.uk/ (2005.05.25)

[KCAP] Jonathan Baxter, Andrew Tridgell, Lex Weaver: *KnightCap: A chess program that learns by combining $TD(\lambda)$ with game-tree search*, http://arp.anu.edu.au/ftp/papers/jon/knightcap.ps.gz (2005.05.10)

[MIKÖNYV] Stuart J. Russell, Peter Norvig: *Mesterséges Intelligencia - Modern megközelítésben*, Panem Könyvkiadó, 2000, [1093], ISBN 963-545-241-1

[MPI] http://www-unix.mcs.anl.gov/mpi/ (2005.05.10)

[PVM] http://www.csm.ornl.gov/pvm/pvm_home.html (2005.05.10)

[SAKKPROMI] Kovács P. Attila: *Sakkprogramozásról mindenkinek*, Novotrade, [240], ISBN 963-02-4691-0

[SZASAKK] Bartel, Kraas, Schrüfer: *Számítógép és sakk*, Data Becker, Novotrade, [333], ISBN 963-02-4692-9

[XBPROTO] Tim Mann: *Chess Engine Communication Protocol*, http://www.tim-mann.org/xboard/engine-intf.html (2005.05.10)