Szakdolgozat

Szabó István

Debrecen
2010

Debreceni Egyetem
Informatikai Kar

Sakkvariánsok vizsgálata

Témavezető:
Kósa Márk
egyetemi tanársegéd

Készítette:
Szabó István
programtervező informatikus

Debrecen
2010

Tartalomjegyzék

Bevezetés

Hatéves korom óta játszom sakkot kisebb-nagyobb kihagyásokkal, több-kevesebb sikerrel. Figyelemmel kísérem az aktuális világversenyeket, egy-egy helyi versenyen, vagy az interneten pedig néha én is megmérettetem magam. Napjainkban az alábbi módokon találhatok magamnak ellenfelet:

- játék másik személy ellen élőben
- játék másik személy ellen interneten keresztül
- játék PC-n futó sakkprogram (számítógép) ellen
- játék sakkprocesszorral rendelkező számítógép (sakkgép) ellen

Magam részéről nem vagyok válogatós az ellenfelek tekintetében, szívesen játszom sakkprogram, sakkgép ellen is, persze a kedvenc ellenfél mindig az élő személy marad. Egy gép ugyanolyan monoton akárhányszor is játszom vele, mindig a legjobb lépésre törekszik, több játék után némileg ki is ismerhető. Nincs arca, nem olvasható le róla, hogy mire gondol, nem lehet zavarba hozni egy kellemetlen lépéssel. Persze megverhető, de kisebb a sikerélmény.

A sakkgépek olyan számítógépek, melyekben beépített sakkprocesszor van. Egy lépés számukra egyetlen utasításnak felel meg, ezáltal nagyobb számítási sebesség érhető el, mint a hagyományos PC-k esetén, ahol például a huszár lépéséhez legalább három utasításra van szükség. Ezek a sakkgépek komoly fejlődésen mentek át, óriási világszenzáció volt 1997-ben, amikor az IBM sakkprocesszorral rendelkező szuperszámítógépe először megverte az aktuális sakkvilágbajnokot[1]. Ma számukra a legnagyobb sakkozó sem jelent kihívást. Sajnos nincs mindenkinek lehetősége ilyen szekrénynyi gépekkel játszani, a piacon elérhető sakkgépek pedig hiába találják meg a legjobb lépéseket, gyengébb teljesítményük miatt hosszú perceket, sokszor akár órákat kell várni egyetlen lépésre. Élvezhetetlenné válik a játék.

Online sakkot játszani rizikós. Egyrészt nem tudhatjuk, hogy ki van a sakktábla másik oldalán, sokszor elkezdődik a játszma, és a második lépés során kiderül, hogy az ellenfél nagyon gyenge. Ilyenkor nem szívesen sérti meg az ember, végigjátssza a meccset, de megint csak élvezhetetlenné válik a játék. Másrészt lehet, hogy a másik oldalon lévő játékos nem megengedett számítógépes segítséget használ, hiszen úgyse látja senki.

[1] IBM Deep Blue – Garry Kasparov 3,5-2,5 http://www.research.ibm.com/deepblue/home/html/b.html

Mai rohanó világunkban élő ellenfelet találni nem könnyű feladat. Vannak barátok, osztálytársak, versenytársak, akikkel lehet egy jót játszani, de velük nem találkozhat az ember akármikor, amikor ideje engedi.

Aki mindig kapható egy gyors partira az a számítógép. Időm nagy részét a szakmám és a munkám miatt számítógép előtt töltöm és egy tíz perces játszma kikapcsolódásként szinte bármikor beiktatható. Rengeteget játszom különböző sakkszoftverek ellen. Régen az volt a kihívás, hogy egyáltalán eredményt tudjak elérni, hiszen ezek a szoftverek szintén könnyedén megverik az embert. Mostanában azonban már nem csak az eredmény hajt, hanem hogy legyen nekem is egy saját sakkprogramom.

Talán jól látszik, hogy nem kellett sokat gondolkoznom szakdolgozatom témaválasztásakor. Szinte biztos voltam benne, hogy valamilyen sakkal kapcsolatos témát választok. Az interneten körülnézve azt látom, hogy, rengeteg ilyen szoftver elérhető: különböző erősségűek, grafikusak, parancssorosak, pár megabájtos szoftverektől egészen a több DVD-s rendszerekig. A Linux rendszerek már régóta tartalmaznak beépített sakkjátékot és néhány éve a Windows Vistába is bekerült egy ilyen program.

Pontosan ez a rengeteg létező sakkprogram ösztönzött arra, hogy valami újat próbáljak meg készíteni, egy olyan sakkprogramot, ami némiképp különbözik az eddigiektől. A hangsúlyt így nem csak a hagyományos sakkra helyeztem, hanem a különböző sakkvariánsokra is.

A sakk és az informatika

Nem is gondolná az ember, hogy egy komoly sakkprogram az informatikának mennyi területét használja:

- adatbázis kezelést: megnyitás és végjáték adatbázisokban. Egy jó sakkprogram az első 10-15 lépésen nem gondolkodik, hanem az adatbázisa alapján szinte azonnal lép. Itt mindenképp fontos, hogy gyorsan tudjunk keresni az adatbázisban.
- mesterséges intelligenciát: a lehetséges lépések közül a lehető legjobb megtalálása. A programnak el kell tudnia döntenie, hogy a huszárral lép, vagy a bástyával, hogy leveszi-e a felkínált csalit, vagy nem dől be a cselnek. Ez egy roppant időigényes feladat, hiszen olykor több mint száz lehetséges lépésből kell a lehető legjobbat, de legalább egy elég jót megtalálni.
- grafikus eszközöket: sakknál elvárható a színes, grafikus felület, a méretükben és színükben paraméterezhető tábla és bábuk. A nagymesterek gyakran panaszkodnak egy számítógép ellen vívott játszma során, hogy nem látják úgy át a táblát, mint amikor egy asztal mellett ülnek. Így a komolyabb szoftverekben általában 3D-s táblák és figurák is rendelkezésre állnak. Az pedig, hogy a bábukat egérrel lehessen mozgatni, egy teljesen természetes igény.
- kommunikációs eszközöket: a sakkprogramok általában felkínálják a lehetőséget, hogy élő személy játsszon élő személy ellen, a program ilyenkor mindössze a szabályok betartatásáért felelős. Mivel a sakkjáték során mindig felváltva lépnek a felek, így az egész lebonyolítható egy számítógépen, először én lépek, majd az ellenfelem. De gyakran megesik, hogy a két fél nincs közös helyen, akár más-más országban is lehetnek, nem tudják ugyanazt a gépet használni, mégis szívesen játszanának egymás ellen. Különböző hálózati interfészek segítségével ilyenkor megoldható, hogy a program ne parancssorból vagy egér által kapja meg a lépést, hanem egy távoli számítógépről.
- fájlkezelést: gyakran megesik, hogy az elkezdett játék félbeszakad. Nem sikerült időben befejezni a játszmát és dolgozni, iskolába kell menni, vagy egyszerűen másra kell a futtató számítógép és ki kell lépni a programunkból. Elvárható, hogy ha egy játszmát elkezdtünk, akkor azt később, akár hetek múlva is folytatni tudjuk. Ehhez azonban a megkezdett játékot háttértárra kell írni. Azt gondolhatnánk, hogy ez egy

egyszerű folyamat, kiírjuk, hogy hol állnak a bábuk, ki következik a lépésben és készen vagyunk. De jobban belegondolva ez így kevés. Például azt is számon kellene tartani, hogy sáncolhatok-e még, tehát hogy léptem-e már a királlyal. Ez csak egy felállított táblából ránézésre ugyanis nem mindig derül ki. Gyakran lehetőség van lépések visszavonására is, sokszor egészen az elejéig visszavonunk egy elrontott játszmát. Ezt pedig csak úgy tudjuk megtenni, ha nem csak a kilépéskor fennálló hadállást mentjük ki, hanem az első lépéstől kezdve az egész játékmenetet. Tehát mint látható teljes, minden eseményre kiterjedő játéktörténet elmentése szükséges a sakkprogram megfelelő használhatóságának és működésének érdekében.

Dolgozatomban a hangsúlyt a mesterséges intelligencia kapta, tehát a sakkprogramnak az a része, amely a „gondolkodást” végzi. A program alkalmaz grafikus elemeket, több bábukészlet is rendelkezésre áll, az egér használható, a tábla azon mezői, ahova szabályosan léphetünk pedig külön színnel vannak kiemelve. A fájlkezelés is támogatva van, és mivel a sakkszoftverek egy szabványos formátumba mentenek, más szoftverben kimentett játszmák is betölthetőek. Az adatbázis kezelés és a hálózati kommunikáció megvalósítása egyelőre nem került be a programba. Terveim szerint azok a szakdolgozatom megírását követő hónapokban kerülnek kifejlesztésre.

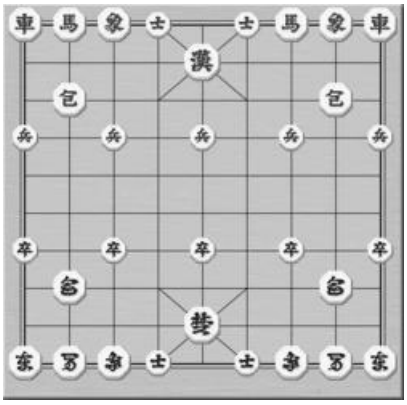
Sakk és sakkvariánsok

A sakkot a legtöbb ember ismeri, azt azonban már kevesebben tudják, hogy a sakknak több ezer változata létezik. A legelterjedtebb változat, amit Európában is játszanak, a modern sakk (hagyományos vagy európai sakk) névre hallgat. Ennek története a legendák világába nyúlik vissza.

A hagyományos sakk szabályait[2] a világon szinte mindenütt ismerik, ezen alapszik a Nemzetközi Sakkszövetség (FIDE[3]), ennek alapján indulnak a világbajnokságok és olimpiák. Magyarországon a hagyományos sakkon kívül más sakkvariáns alig ismert, ez azonban nem mindenhol van így. A legtöbb (főleg ázsiai) országnak van saját sakkvariációja és ezekből komoly versenyeket, olykor nemzetközi tornákat is bonyolítanak. Vizsgáljuk meg, miben különbözhetnek egymástól az egyes variációk. Ehhez először nézzük meg, hogy mi kell a sakkjáték játszásához. Mindenképpen kell tábla, bábuk, ellenfél, tudnunk kell, hogy az egyes bábuk hogyan léphetnek, kell egy cél, ismernünk kell a bábuk alaphelyzetét, azaz a tábla kezdeti felállítását, tehát összességében ismernünk kell a játék legfontosabb szabályait.

A modern sakkban ezekre a kérdésekre a választ mindenki jól ismeri, 8x8-as a táblán, hat fajta bábu sorakozik fel, a játékot ketten játsszák egymás ellen, a cél az ellenfél királyának mattot adni. Az egyes sakkvariánsokban ezekből a paraméterekből egy, vagy akár több is változhat. A Fischer és a random sakk a bábuk alaphelyzetében különbözik a modern sakktól, a francia sakkban a játék célja más. Léteznek variánsok, ahol már a tábla sem 8x8-as, sőt olyan is van ahol nem is négyszög alakú. A dinamikus sakkban a tábla mérete kezdetben csupán egyetlen mező, és a játék során ez dinamikusan bővül. Az Alice sakkot egyszerre két táblán játsszák, van az alaptábla, és egy úgynevezett tükörtábla. Egyes variációkban megjelennek új figurák, új lépésekkel. Az indiai változatokban nagyon gyakran szerepel elefánt a huszár helyett, ami nem képes „L” alakban lépni, de bármerre léphet egy vagy két mezőt. Sok variációban kap szerepet a kancellár, ami a huszár és a vezér lépéseit ötvözi és ezáltal hatalmas erővel bír.

[2] http://www.fide.com/component/handbook/?id=124&view=article
[3] FIDE - Fédération Internationale des Échecs, az 1924-ben alapított Nemzetközi Sakkszövetség rövidítése.

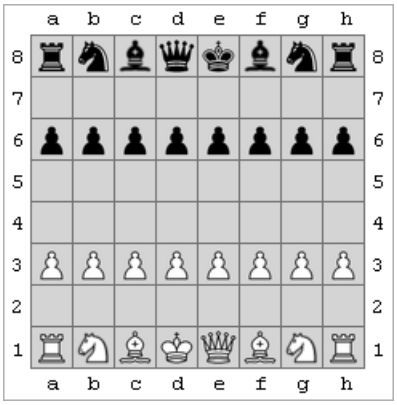


1. ábra A koreai sakk (Janggi) alapállása

A koreai sakkban más sakkvariációktól eltérően nem a tábla mezőin, hanem a négyzetháló pontjain állnak a bábuk. A figurák közül a bástya, huszár és király hasonló szerepet tölt be, mint az európai sakkban, de a testőr, ágyú és elefánt jelenléte mégis egy teljesen más variációt eredményez.

2. ábra Háromszemélyes sakktábla

Mint látható a háromszemélyes sakktábla hatszög alakú. A játék szabályai megegyeznek a kétszemélyes sakkéval, annyi a különbség, hogy nem egy, hanem két ellenfélnek kell mattot adni a győzelemhez.



3. ábra A thai sakk (Makruk) alapállása

A ma is játszott sakkváltozatok közül talán a makruk hasonlít leginkább a sakkváltozatok közös őséhez. Ezt az is bizonyítja, hogy itt még nem jelentek meg a fekete-fehérre színezett mezők, minden mező színe ugyanolyan. Vlagyimir Kramnyik volt sakkvilágbajnok szerint *„a thai makruk stratégiaibb, mint a nemzetközi sakk”*.

Az általam írt program a hagyományos sakk mellett a Fischer-féle random sakkot és a random sakkot valósítja meg.

Fischer-féle random sakk

A Fischer-féle random sakk (vagy 960-as sakk) Bobby Fischer amerikai sakknagymester, sakkvilágbajnok nevéhez fűződik. A legfontosabb különbség a modern sakkhoz képest, hogy a tisztek[4] kezdőhelyét az alapsoron sorsolják, bizonyos szabályokat figyelembe véve. A 960-as elnevezés is erre utal, ugyanis ennyi különböző kezdőállás lehet. Minden egyes felálláshoz egy szám van rendelve 0-től 959-ig, játék előtt egyszerűen húzunk egy számot, és annak megfelelően állítjuk fel a táblát. Célja a sakk megújítása, itt ugyanis a betanult megnyitáselméleti változatok elvesztik értelmüket.

A figurák elhelyezésének szabályai:

- a tiszteket az alapsoron véletlenszerűen helyezzük el ügyelve a következőkre:
    - a futók különböző színű mezőkön álljanak
    - a két bástya bárhol állhat, de a királynak a kettő között kell lennie
    - a sötét és világos figurák szimmetrikusan helyezkedjenek el
- a gyalogok a hagyományos sakknak megfelelően a második illetve a hetedeik soron állnak.

Több algoritmus is ismert arra vonatkozóan, hogy hogyan rendelhetünk egy számhoz egy felállítást. Reinhard Scharnagl több nagysikerű sakk-könyv szerzője a következő megoldást javasolja:

- véletlenszerűen válasszunk egy számot az intervallumból, majd osszuk el néggyel. Az osztás maradéka fogja kijelölni a világos futó helyét: a maradék lehet 0, 1, 2 vagy 3, és ez rendre a *b*, *d*, *f* vagy *h* oszlopot jelöli ki.
- az előzőleg kapott eredményt osszuk el megint néggyel, ennek maradéka adja meg a sötét futó helyét: 0 maradékhoz az *a* oszlop, 1-hez a *c*, 2-höz az *e*, míg 3-hoz a *g* oszlop tartozik.
- osszuk tovább az előzőleg kapott eredményt hat részre. Az osztás maradéka most megadja a vezér helyét a következőképpen: balról számoljunk le maradéknyi + 1-et a már felállított futókat kihagyva.
- az előző osztás eredménye egy 0 és 9 közötti szám. Ezt a számot kikeressük a KRN-táblázat (King-Rook-kNight, azaz király-bástya-huszár) első oszlopából, és a második oszlop alapján balról-jobbra feltöltjük a még üres helyeket.

[4] a tiszt a vezér, bástya, futó és huszár összefoglaló neve

KRN-kód	Pozíció
0	NNRKR
1	NRNKR
2	NRKNR
3	NRKRN
4	RNNKR
5	RNKNR
6	RNKRN
7	RKNNR
8	RKNRN
9	RKRNN

4. ábra KRN-táblázat

A hagyományos sakk alapfelállását az 518-as kód adja.

518 / 4 = 129, maradék a 2, a világos futó tehát az *f* oszlopra kerül.

129 / 4 = 32, maradék az 1, így a sötét futó a *c* oszlopon lesz.

32 / 6 = 5, maradék a 2. A vezér balról a harmadik (maradék+1) üres oszlopra, azaz *d*-re kerül.

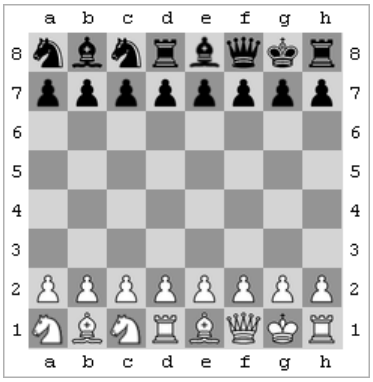
A többi tiszt helyzetét a KRN-táblázat ötödik sora adja, azaz RNKNR. Ebből R (bástya) megy az *a* oszlopra, N (huszár) a *b*-re, *c*-n és *d*-n már áll bábu, K (király) a következő üres *e* oszlopra kerül, majd végül N és R a *g* és *h* oszlopra.

A 960 alapállás a következő módon alakul ki. A futók különböző színekre kerülnek, így mindkettő 4-4 helyen állhat. A vezér a nyolc oszlopból hatra kerülhet, arra a hatra, amelyre előzőleg nem tettünk futót. Ez eddig $4 \cdot 4 \cdot 6 = 96$ lehetőség. A következő bábu egy huszár, ami a fennmaradó öt helyre kerülhet, majd a másik huszár már csak négy helyre. A két bástya és a király elhelyezése a maradék három helyre már csak egyféleképpen történhet, hiszen a király áll középen. Így $4 \cdot 4 \cdot 6 \cdot 5 \cdot 4 \cdot 1 = 1920$ különböző eset jöhet létre. A fenti levezetésben azonban a két huszárt különbözőnek tekintettük. Mivel a két huszár a játék folyamán megkülönböztethetetlen, azaz ha kicserélnénk őket semmilyen változás nem történne, így az eredményt el kell osztani kettővel, ezzel pedig megkaptuk a 960 alapállást.

Fontos kérdés, hogy hogyan oldjuk meg a sáncolást, hiszen most a bástya és a király helyzete eltér(het) a megszokott felállításhoz képest. A sáncolás megvalósítására nincs általános szabály, csupán ajánlás, így a sáncolás kérdését minden verseny előtt pontosan tisztázni kell. Az egyik gyakori megvalósítás - amit a program is alkalmaz - úgy végzi a sáncolást, mintha hagyományos sakkot játszanánk. Tehát bármi is a király és a bástyák helyzete, rövid sánc után a király mindig a *g* oszlopra, hosszú sánc esetén pedig mindig a *c* oszlopra kerül, a bástya pedig rövid sánc esetén az *f*, míg hosszú rosálás után a *d* oszlopon áll majd. Előfordulhat tehát, hogy sáncoláskor a király és a bástya közül csak az egyik változtat helyet, ami a hagyományos sakkhoz képest teljesen szokatlan, sőt olyan lépést is tehet a király, amely során nem egyértelmű, hogy egyet lép, vagy sáncolni akar.

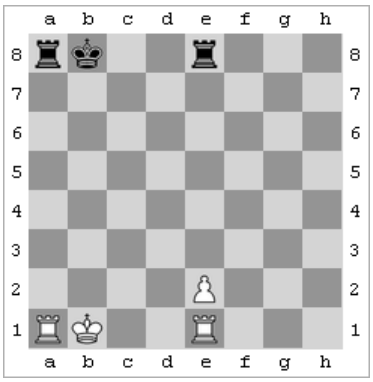
Természetesen sáncolás előtt teljesülnie kell a „három u” szabálynak: unmoved, unattacked és unimpeded, azaz:

- nem léptek még idáig a sáncolásban résztvevő bábuk,
- a király nem áll sakkban, sáncolás közben a királynak nem kell ellenséges bábu által támadott mezőn átlépnie, valamint sáncolás után a király nem lesz sakkban (az utóbbi szabálynak minden lépésre teljesülnie kell),
- a király és a sáncolásban résztvevő bástya között nem áll sem saját, sem ellenséges bábu.

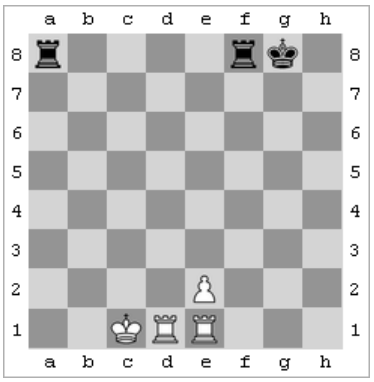


5. ábra Egy alapállás

Egy lehetséges alapállás. A futók különböző színeken, király a bástyák között, és a tisztek egymással szemben állnak. Látható, hogy a megszokott, sokszor 15-20 lépés mélységig betanult megnyitások helyett teljesen új stratégiát kell kidolgozni.



6. ábra Sáncolás előtt



7. ábra Sáncolás után

Sáncolás előtt és után. Sötét rövid váncot hajtott végre, tehát királya a *g* oszlopra került, a bástya pedig *f*-re. Világos hosszú váncot választott, így most királya a *c*, míg bástyája a *d* oszlopon áll. Az *e2* gyalog tette lehetővé, sötét rosálását, ha nem állna ott, akkor a sötét király nem mehetett volna át az *e8* mezőn *e1* bástya miatt. Míg hagyományos sakk esetén a király sáncoláskor minden esetben kettőt lép, addig Fischer sakkban léphet egytől ötig a felállítástól függően. Esetünkben világos mindössze egyet, sötét pedig ötöt lép királyával. A

sáncolás mindig a király lépése, valódi sakktáblán a királyt kell először megfogni, lépni vele, majd utána következhet a bástya. Ha a bástyát fogjuk meg először, akkor a fogott bábu lép szabály miatt a bástyával kell lépnünk, és mivel a sáncolás nem a bástya lépése, így a királyt már nem mozgathatjuk utána, mert két lépést tennénk. Egy sakkprogram hagyományos sakknál sáncoláskor a bástyával automatikusan lép, hiszen a szoftver felismeri, hogy a király kettőt lép, ami csak sáncolás esetén fordulhat elő. De most nézzük meg mi a helyzet Fischer-sakk esetén. Ha sáncoláskor csak egyet lép a király, akkor a lépés nem egyértelmű. A 6. ábrán látható állásban világos lép, és a *Kc1* lépést választja. A program nem tudja eldönteni, hogy a király *c1*-re akar lépni, vagy a játékos hosszú sáncot szeretne végrehajtani.

A random sakk

A random-sakkban szintén véletlenszerűen állítjuk fel a világos tisztsort (a sötétet pedig ezzel szimmetrikusan), de nem vesszük figyelembe a Fischer-féle sakknál lévő többi felállítási szabályt. Így olyan speciális kezdőállások is létrejöhetnek, ahol a király a sarokban áll, vagy egy játékosnak mindkét futója azonos színű, ami azért érdekes, mert ellenfelének ilyenkor két ellentétes színű futója lesz. A sáncolás ebben a variációban szintén értelmezés kérdése, itt azonban sokszor nincs lehetőség rá. Mivel a tisztsor felállítására semmilyen megkötés nincs, így egyszerű permutációval kiszámolható az összes lehetséges alaphadállás:
A $\{K, Q, B, B, N, N, R, R\}$ halmaz elemeit hányféleképpen lehet sorba rendezni? Mivel bástyából, huszárból és futóból kettő-kettő van és ezek megkülönböztethetetlenek, ezért ismétléses permutációt kell alkalmaznunk.

$$P_n^{k_K,k_Q,k_B,k_N,k_R} = \frac{n!}{k_K!\,k_Q!\,k_B!\,k_N!\,k_R!} = \frac{8!}{1!\cdot 1!\cdot 2!\cdot 2!\cdot 2!} = \frac{8!}{8} = 7! = 5040,$$

ahol n a sorba rendezendő halmaz elemeinek száma, k_i pedig az egyes megegyező figurák száma $(i = 1, \dots, 5)$.

A sakkprogramok felépítése és működése

A korszerű sakkprogramok sakklogikát megvalósító része három fő egységre bontható. Megnyitás és végjáték adatbázist használnak, a harmadik összetevő pedig a mesterséges intelligencia, amely az egész sakkprogram lelke.

A megnyitás adatbázis

A megnyitás adatbázis nem más, mint megfelelő erősségű játékosok által már legalább egyszer lejátszott játszmák gyűjteménye. Tehát nem csak megnyitásokat tartalmaz, de mint látni fogjuk, csak megnyitásokban válik hasznunkra, innen ered az elnevezés. A megfelelő erő azt jelenti, hogy olyan játékosok játszották, akik megbízható, jó játékosok, a lépésük átgondolt, valószínűleg alapos elemzés eredménye. Például 2500 Élő-pont[5] feletti játékosok játszmái elég erős adatbázist adnak, hiszen e szintet csak a játék legjobbjai érik el.

Egy megnyitás adatbázis annál jobb, minél több megfelelően erős játékos által játszott játszmát tartalmaz, és mivel a játszmák rögzítése időigényes feladat, ezért vannak külön erre a feladatra specializálódott programok, sőt megnyitás adatbázist külön lehet vásárolni. A Shredder több világbajnok sakkprogram fejlesztője például külön kínálja megvásárlásra a több mint 16 millió lépést tartalmazó adatbázisát. Ez méretét tekintve is hatalmas, több mint 1 GB, egy részlete interneten elérhető és ingyen kipróbálható[6].

Move	Value	Games	Perc.	Elo	Perf.	Year	Wins	Draws	Losses
1.e4	-	242221	54.5%	2507	2539	1990	83248	97815	61158
1.d4	-	204646	56.3%	2515	2555	1990	72648	85419	46579
1.Nf3	-	59276	55.7%	2509	2546	1990	19990	26061	13225
1.c4	-	45320	56.3%	2510	2548	1987	15914	19274	10132
1.g3	-	5864	55%	2501	2532	1989	2037	2382	1445
1.b3	-	1505	52.9%	2498	2515	1990	526	542	437
1.f4	-	1078	46.1%	2472	2457	1988	314	366	398
1.Nc3	-	464	45.6%	2470	2461	1992	135	154	175
1.b4	-	266	49.2%	2482	2491	1987	85	92	89
1.e3	-	108	48.6%	2456	2480	1988	38	29	41
1.d3	-	99	47.9%	2469	2461	1987	27	41	31
1.c3	-	52	49%	2493	2489	1990	18	15	19

8. ábra Részlet a Shredder megnyitás adatbázisából

[5] Élő Árpád, magyar származású fizikus által kifejlesztett pontrendszer. Jelenleg ezt használja a FIDE. http://www.chessbase.com/newsdetail.asp?newsid=4326

[6] http://www.shredderchess.com/online-chess/online-databases/opening-database.html

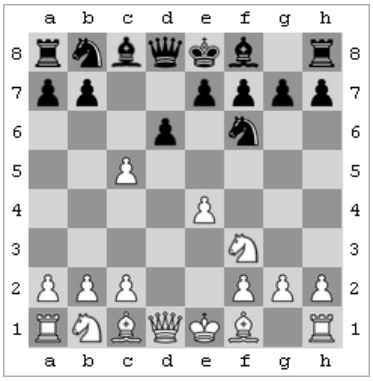
A 8. ábrán lévő táblázat a megnyitás adatbázis egy részlete, éppen világos első lépését vizsgáljuk. Egy-egy sor egy-egy lépésnek felel meg. Látható, hogy *e4* kezdést 242221 játszma tartalmaz, és e kezdés esetén világos 83248 játszmában nyert, 97815 esetben pedig döntetlent ért el, tehát a 242221 játékból szerezhető 242221 maximális pontból pontosan 83248+97815/2 pontot szerzett (győzelemért 1 pont, míg döntetlenért fél pont jár). Ez 54,5%, ami azt jelenti, hogy érdemes világosnak ezt a megnyitást választani, hisz akik eddig ezt választották többször nyertek, mint vesztettek. Lépjük is meg ezt az *e4* lépést. Ekkor sötét lépése következik, a táblázatban most az ő lehetséges lépései jelennek meg.

Move	Value	Games	Perc.	Elo	Perf.	Year	Wins	Draws	Losses
1...c5	-	115998	46.4%	2511	2481	1991	31218	45441	39339
1...e5	-	49163	44.7%	2511	2478	1989	11069	21837	16257
1...e6	-	31007	44.3%	2505	2468	1991	7470	12548	10989
1...c6	-	16320	44.7%	2508	2473	1991	3708	7181	5431
1...d6	-	9509	44.1%	2503	2463	1991	2425	3553	3531
1...g6	-	8499	44.9%	2499	2460	1990	2335	2977	3187
1...Nf6	-	5823	43.4%	2494	2450	1988	1444	2169	2210
1...d5	-	4280	42.6%	2488	2446	1994	1020	1611	1649
1...Nc6	-	1268	44.6%	2491	2457	1990	369	395	504
1...b6	-	273	45.9%	2498	2470	1993	82	87	104
1...a6	-	60	29.1%	2466	2333	1992	12	11	37
1...g5	-	10	45%	2434	2455	1987	4	1	5
1...h6	-	7	21.4%	2440	2259	1993	0	3	4
1...Na6	-	3	83.3%	2438	2714	1998	2	1	0
1...f6	-	1	0%	2460	2150	1995	0	0	1

9. ábra Sötét válaszának statisztikái világos e4 nyitására

A számítógépnek azt célszerű választania, ami mellett a legnagyobb % áll, tehát amely lépés esetén a legnagyobb esélye van a nyerésre, vagy pontszerzésre. Így sötétnek a *Na6* lépést kellene lépnie, hiszen ez eddig az összes megszerezhető pont 83,3%-át adta (2 győzelem, 1 döntetlen, 0 vereség). De ezen lépés mögött nincs elég játszma, mindösszesen három ilyen játszma szerepel az adatbázisban, amiből így nem vonhatunk le kellő biztonsággal következtetést. A számítógép nem is ezt fogja választani, hanem a második legnagyobb értéket, amit *c5* képvisel, és már több mint 115 ezer játszma alapján kellő biztonsággal választhat. Meg kell határozni tehát egy olyan minimális játszmaszámot, amely mellett biztonsággal támaszkodhatunk az adatbázisra. Sötét számára még *b6* is elég jónak tűnik, ez a harmadik legmagasabb érték, és 273 játszmában szerepel.

A magam részéről csak azokat a lépéseket venném számításba, amiket legalább 200 játszmában már játszottak, pontosabban, amely lépéshez az adatbázis tartalmaz legalább 200 lejátszott játszmát. Az adatbázis egy faszerkezettel reprezentálható, van egy állás, amelyből több állás következhet a lehetséges lépéseinktől függően. Így egy mélységet meghaladva már annyi águnk lesz, hogy egyik se fog 200-nál több játszmát tartalmazni, akkor pedig nem hagyatkozhatunk már az adatbázisra. Ekkor jön el a tényleges számítás, és ezzel a mesterséges intelligencia ideje. Ez a mélység változó. A 1. *d4 d5*, úgynevezett elfogadott vezércsel megnyitás iszonyú alaposan ki van elemezve, szinte minden változatát végigjátszották több százszor. Így itt az adatbázis még 20 lépéspár után is tud olyan lehetőségeket adni, amelyek közül biztonsággal választhatunk. Az 10. ábrán bemutatott játszma esetén azonban a mélység mindössze négy lépéspár, sötét negyedik lépését már a mesterséges intelligenciának kell megadnia.



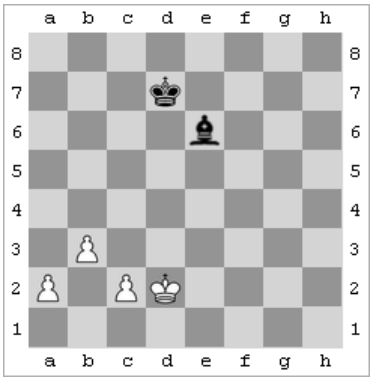
10. ábra Szicíliai védelem egy változata

Az 1. *e4* *c5*
2. *Nf3* *d6*
3. *d4* *Nf6*
4. *dxc5*

lépések után a 10. ábrán látható hadállás alakul ki. A Shredder adatbázisa erre az esetre két lépéssel áll elő, *Qa5+* vagy *Nxe4,* de mivel mindkettőt csupán 30 játszmában játszották, nem fogathatjuk el teljes biztonsággal az ajánlást, így hosszas számolás veszi majd kezdetét.

A végjáték adatbázis

Beszélünk 3, 4, 5 és 6 bábus végjáték adatbázisról. A számok azt adják meg, hogy hány még táblán lévő figura esetén működik a keresés az adatbázisban. Ha egy táblán bármilyen felállításban már csak hat (vagy annál kevesebb) bábu szerepel, akkor a (hat bábus végjáték adatbázishoz tartozó) kereső azonnal megadja az összes lehetséges lejátszást. Ez a nagy különbség a megnyitás adatbázishoz képest, hogy itt bármely hat bábu bármely felállítása az adatbázisban szerepel, az összes lehetséges lejátszással együtt, ugyanis ez nem valós játszmák alapján készült, hanem számítógéppel generálták, minden helyzetet figyelembe véve. Tehát, ha egy játszma folyamán olyan állás alakul ki egy ütés után, hogy már csak hat bábu maradt, akkor a számítógép azonnal tudja, hogy ebből az állásból minimum hány lépésben adhat mattot, vagy vesztes helyzetben mit kell lépnie a döntetlen kiharcolásához. Egy példán keresztül szemléltetve ez a következőt jelenti:



11. ábra Egy döntetlen szagú végjáték

A 11. ábrán látható állásban összesen már csak hat bábu maradt, tegyük fel, hogy világos lép. Ránézésre döntetlen gyanús az állás, de talán azt gondolnánk, hogy világos a három gyalogjából legalább egyet be tudni vinni a nyolcadik sorra, és akkor megnyeri a játszmát. Sötét célja már csak a döntetlen lehet, ha világos nem csinál akkora baklövést, hogy saját gyalogjai megfojtsák[7], akkor sötét nem tud neki mattot adni. A Shredder végjáték adatbázisa szintén kipróbálható[8]. A 11. ábrán lévő állás FEN kódja: 8/3k4/4b3/8/8/1P6/P1PK4/8, pipáljuk még ki, hogy világos következzen (White to move). A kereső azonnal kiírja az összes lehetséges lépést, a lehetséges legjobb kimenetekkel:

[7] megfojtást jelent, ha a király a saját bábui miatt nem tud elmenekülni egy sakkból
[8] http://www.shredderchess.com/online-chess/online-databases/endgame-database.html

Draw

Move	Value
a2-a3	Draw
a2-a4	Draw
c2-c3	Draw
c2-c4	Draw
Kd2-e2	Draw
Kd2-d1	Draw
Kd2-d3	Draw
Kd2-c1	Draw
Kd2-e1	Draw
Kd2-c3	Draw
Kd2-e3	Draw
b3-b4	Draw

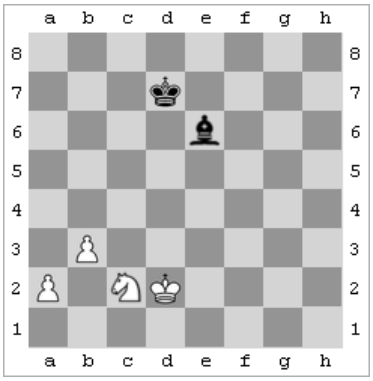
12. ábra Világos lépéseinek lehetséges kimenetele

Draw

Move	Value
Be6-d5	Draw
Be6-f5	Draw
Be6-g4	Draw
Be6-h3	Draw
Be6-f7	Draw
Be6-g8	Draw
Kd7-c7	Draw
Kd7-e7	Draw
Kd7-d6	Draw
Kd7-d8	Draw
Kd7-c6	Draw
Kd7-c8	Draw
Kd7-e8	Draw
Be6-c4	Lose in 17
Be6xb3	Lose in 17

13. ábra a3 után sötét lépéseinek lehetséges kimenetele

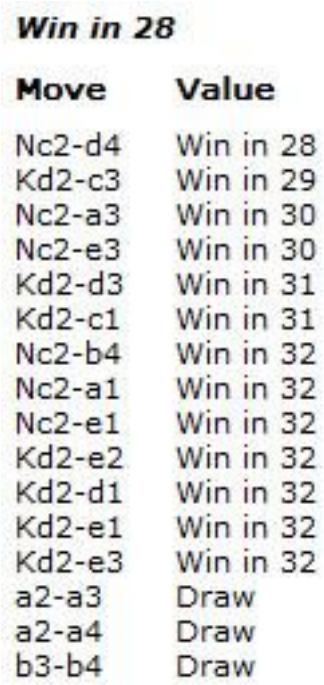
A 12. ábrán azt látjuk, hogy világos a lehetséges 12 lépése közül bármelyiket is lépi, sötét ki tudja harcolni a döntetlent, ha nem hibázik. Válasszuk ki az *a3* lépést. Sötét erre 15 szabályos lépést léphet (13. ábra), amiből 13 döntetlenhez vezet, kettő esetben pedig 17 lépésben veszítene. Sötét persze a számára legkedvezőbbet lépi, tehát valamelyik döntetlen ágat választja. Mindkét játékos legjobb lépéseinek sorozata döntetlenhez vezet. A következő ábrán már szomorúbb sötét helyzete:



14. ábra Egy másik végjáték

A 14. ábra FEN-kódja: 8/3k4/4b3/8/8/1P6/P1NK4/8, szintén világos van soron. A lehetséges 16 lépése mindegyikének eredménye azonnal megjelenik a táblázatban (15. ábra). Van három döntetlenre vezető lépés, a többiben világos nyer 28-32 lépésen belül. Világos persze a legjobbat választja, ami jelen esetben a legrövidebb út a nyerésig, azaz *Nd4*. Ha sötét ezután minden lépésében a számára lehető legjobbat lépi, akkor is mattot kap 27 lépés múlva (persze ha világos követi az ajánlott lejátszást). Hatalmas segítség ez a sakkszoftvernek. Hiszen az adatbázis leírja a következő 28 lépését egy pillanat alatt! Ezt a 28 lépést természetesen a mesterséges intelligencia is megtalálná, de ha lépésenként 1 perc

gondolkodással számolunk (ami egy nagyon jó idő lenne), akkor még 28 percre lenne szüksége a mattadáshoz. Ennyi idő pedig ritkán marad a játszma végére.



Win in 28

Move	Value
Nc2-d4	Win in 28
Kd2-c3	Win in 29
Nc2-a3	Win in 30
Nc2-e3	Win in 30
Kd2-d3	Win in 31
Kd2-c1	Win in 31
Nc2-b4	Win in 32
Nc2-a1	Win in 32
Nc2-e1	Win in 32
Kd2-e2	Win in 32
Kd2-d1	Win in 32
Kd2-e1	Win in 32
Kd2-e3	Win in 32
a2-a3	Draw
a2-a4	Draw
b3-b4	Draw

15. ábra Világos mattot ad 28 lépés múlva

Jelenleg a hat bábus végjáték adatbázis a legbővebb, ami létezik és kapható. Nyilvánvaló azonban, hogy az a szoftvergyártó, aki elsőként legenerálja és sakkprogramjába beleépíti a teljes hét bábus adatbázist óriási lépést tesz a világbajnokság megnyeréséhez vezető úton. A feladat persze nem olyan egyszerű, hiszen hét bábunak már csak a felállítására több millió variáció létezik, arról nem is beszélve, hogy a hét bábu közül öt bármi lehet (akár lehet mind az öt vezér, vagy mind az öt huszár).

Az adatbázisok hatalmas segítséget nyújtanak a sakkprogramnak, hiszen egy adatbázisban történő keresés nagyságrendekkel gyorsabb, mint a következőkben tárgyalásra kerülő mesterséges intelligenciát megvalósító algoritmusok. Ma már a legtöbb nagymester csak akkor áll ki számítógép ellen, ha a gép nem használhat semmilyen adatbázist.

Az általam írt program nem használ adatbázisokat, ugyanis ingyenesen elérhető teljes megnyitás és végjáték adatbázist - érthető módon - a mai napig nem találtam.

Mesterséges intelligencia

Elérkeztünk a sakkprogramok legfontosabb összetevőjéhez. A mesterséges intelligencia (MI) akkor avatkozik be a játékba, amikor a megnyitás adatbázisból nem nyerhető már ki több információ. Az MI feladata tehát a két adatbázis összekötése, magyarul a megnyitás adatbázisból utoljára kinyert állásból lépések sorozatán át eljutni addig, amíg már csak hat bábu marad a táblán, és innen a végjáték adatbázisra támaszkodva már gyorsan befejezhető a mérkőzés. Senki nem tudhatja, hogy ez hány lépés generálását jelenti, hiszen húsztól akár száznál is több lépésig tartó játszmák is akadnak. Persze a játszma akár úgy is véget érhet, hogy mind a 32 bábu a táblán van, ekkor a végjáték adatbázis nem kerül kihasználásra a győzelem elérésének érdekében. Mivel programom semmilyen adatbázist nem használ, így már az első lépés kiválasztása is a MI feladata.

A MI megvalósításához először egy lehetséges állapottér reprezentációt mutatok be a sok közül, majd a kiértékelő függvényeket veszem górcső alá és különböző heurisztikákat vizsgálok meg a kiszámolási idő és az optimálisság szemszögéből. Végezetül a lépésajánló algoritmusok működését mutatom be.

Állapottér reprezentáció

Egy állapot pontos reprezentálásához – mint arra már utaltam – nem elég csupán a sakktáblát leírni. Olyan a játék szempontjából fontos tényezőket is számon kell tartani, hogy ki sáncolhat még, vagy mik voltak a megelőző lépések. Ne feledkezzünk meg arról sem, hogy egy sakktáblára ránézve legtöbbször nem tudjuk, ki következik a lépésben, tehát ezt is nyilván kell tartanunk. Számos olyan adat is tartozhat egy-egy állapothoz, aminek látszólag kevés köze van magához a játékhoz, de bevezetésük később nagyban elősegíti az egyes állások kiértékelését, megkönnyítve ezzel a gyorsabb döntést a felmerülő lehetséges lépések közül. Ilyen lehet annak a tárolása, hogy egy állapot célállapot-e, vagy éppen sakkban áll-e a lépni következő játékos királya, de ide sorolható annak az eltárolása is, hogy egy állapot által reprezentált állás végjáték-e.

```
protected char[][] table = new char[9][9];
protected int[][] whiteAttack = new int[9][9];
protected int[][] blackAttack = new int[9][9];
protected Vector<Figure> figures = new Vector<Figure>();
protected Vector<Move> possibleMoves = new Vector<Move>();
protected Vector<Step> possibleStep = new Vector<Step>();
protected Vector<Step> prevStep = new Vector<Step>();
protected boolean wscastle = true;
protected boolean bscastle = true;
protected boolean wlcastle = true;
protected boolean blcastle = true;
protected boolean isChess = false;
protected boolean isTerminal = false;
protected boolean isEndGame = false;
protected int next = 0;
```

Futtatáskor a legtöbb időt a heurisztikus kiértékelő függvény kiszámolása mellett a példányosítás igényli. Minden gépi lépést megelőzően több száz, akár több ezer állapot jöhet létre a memóriában, így rendkívül fontos, hogy egy állapotot minél kevesebb adattal lehessen jellemezni. A sáncolások vizsgálatára megoldható lenne akár egyetlen `byte` típusú változóval, a program mégis négy `boolean` típusú változót használ. Sokszor célszerű a futtatás gyorsaságának rovására inkább olvashatóbb, könnyebben érthető kódot írni.

A `table` a sakktáblát reprezentálja, karakterekkel kerül feltöltésre, minden karakter egy figura típusát jelöli, az angol elnevezés kezdőbetűje alapján. A figuraobjektumokat a `figures` nevű kollekcióban tárolja a program. Felmerülhet a kérdés, hogy miért nem töltjük fel a `table`-t figuraobjektumokkal, csökkentve így az adattagok számát. Magyarul a

```
protected char[][] table = new char[9][9];
protected Vector<Figure> figures = new Vector<Figure>();
```

adattag deklarációk helyett miért nem alkalmazzuk inkább a rövidebb

```
protected char[][] Figure = new Figure[9][9];
```

deklarációt. Azért választottam az első megoldást, mert a programnak rendkívül sokszor van szüksége egy állapotot vizsgálva csak a figurákra. Ilyen eset például a figurák képének

lekérdezése kirajzoláshoz, egyes heurisztikák kiszámolása, vagy egy-egy figura lehetséges lépéseinek vizsgálata. A második esetben mindig újra és újra be kellene járni a mátrixot, valahányszor a figurákra lenne szükség és a példányosításkor nyert idő itt elveszne.

A `table` minden variáns esetén egy 9x9-es mátrix, melynek az első oszlopát és sorát nem vesszük figyelembe a programban sehol. Így érjük el, hogy a reprezentáció során alkalmazott indexek megegyezzenek a valóságnak, hiszen a Java nullától kezdve számolja az indexeket és a sakktáblán nincs nulladik sor, vagy nulladik oszlop.

Minden állapot rendelkezik egy `prevStep` nevű kollekcióval, amelyben a megelőző lépések vannak letárolva, egészen az első lépéstől kezdve. Ennek szerepe egyrészt a lépések visszavonásában lehet, hiszen ilyenkor az utolsó benne szereplő operátor hatását kell semmissé tenni. Másrészt vannak olyan speciális lépések, a sáncolás és az en Passant, melyek csak akkor tehetők meg, ha a megelőző lépésekre teljesülnek bizonyos feltételek, tehát mindenképp fontos az összes megelőző lépés tárolása.

A `next` numerikus adattag 0 és 1 értéket vehet fel, ennek alapján tudjuk, hogy egy állapotban világos vagy sötét lép-e. Ezt megkaphatjuk a `prevStep` kollekció vizsgálatával is, ugyanis a játékszabályok szerint mindig világos lépi az elsőt. Ha tehát páros számú elem van `prevStep`-ben, akkor világos következik, ha páratlan, akkor sötét. De célszerűbb külön változóban eltárolni a lépő játékos rövidítését, mint minden állapotnál lekérni egy akár százelemű kollekció méretét, majd ezt osztani maradékosan kettővel.

Az `isTerminal` adattag annak tárolására hivatott, hogy az adott állapot célállapot-e, az `isChess` pedig megmondja, hogy a `next` által kijelölt lépni következő játékos királya sakkban áll-e. Ennek a két adattagnak a segítségével megállapítható a játék végeredménye.

Az `isEndGame` adattag értéke akkor `true`, ha végjáték az adott állapot által reprezentált állás. Ennek hasznát a heurisztikus kiértékelő függvényben vesszük, ahol az egyes figurák értékei változhatnak attól függően, hogy végjátékról van-e szó, vagy nem.

A `whiteAttack` és `blackAttack` mátrixok a `table`-hoz hasonlóan egy-egy sakktáblát reprezentálnak, de itt az egyes mezőkön nem a rajta álló figura szerepel, hanem az, hogy az adott mezőt hányan támadják `whiteAttack` esetén a világos, míg `blackAttack` esetén a sötét bábuk közül. Segítségével könnyű eldönteni, hogy egy állapot által reprezentált állásban a király sakkban áll-e.

A lehetséges lépések a `possibleMoves` kollekcióban, míg a lehetséges lépések alapján előállított alkalmazható operátorok a `possibleStep` nevű kollekcióban tárolódnak.

A két vektor létének oka, hogy vannak olyan operátorok, melyeket nem lehet egyetlen lépéssel leírni. Például a sáncolás két lépést igényel, először a királlyal, majd a bástyával lépünk. A másik ok, hogy a `possibleMoves`-ban minden olyan lépés szerepel, ami nem vezet le a tábláról, vagy nem áll a célmezőn saját figura. A `possibleStep`-ben viszont már csak azok az operátorok kaptak helyet, amelyek ténylegesen megtehetőek, tehát meglépésük esetén nem lesz sakkban a király.

Kezdőállapot

Kezdetben `table` minden mezője ’0’ karakterekkel kerül feltöltésre, bármely variáns esetén. A ’0’ karakter azt jelöli, hogy azon a mezőn nincs bábu. Ezután a második sor minden mezőjébe csupa ’p’-t, míg a hetedik sor minden mezőjébe ’P’-t ír a kezdőállapotot létrehozó konstruktor, ezzel jelölve, hogy a második soron sötét, a hetediken pedig világos gyalogok állnak. Az első és nyolcadik sor már az adott variáns függvényében kerül kitöltésre, ezt a felhasználótól kért `gameType` változó értéke alapján hajtja végre a `start()` metódus. A `gameType` értékei hagyományos-, Fischer- és random sakk esetén rendre 1, 2 és 3. A `start()` metódus random sakknál véletlenszám generátort használ, míg Fischer-sakknál a fentebb vázolt KRN-algoritmus segítségével véglegesíti a tisztsorokat.

Ha kialakult `table` végső helyzete, akkor bejárva a mátrixot példányosítással létrehozza a program a figurákat, amiket a `figures` vektorban tárol.

Bármely variáns esetén a játékot világos kezdi, így a `next` változó értéke kezdetben mindig 0 (0 világos következik, 1 sötét következik).

A sáncolást vizsgáló `boolean` típusú adattagok kezdeti értéke hagyományos és Fischer-sakk esetén mindig `true`, random sakknál pedig mindig `false`, mert itt a sáncolás nincs értelmezve.

A lépések nyilvántartására létrehozott lista kezdetben üres, hiszen az első lépés következik, valamint egyik fél sem állhat sakkban (`isChess` = `false`).

Az `isTerminal` logikai típusú változó a célállapotok esetén vesz fel `true` értéket, így kezdőállapotban értéke `false`.

A `whiteAttack` és `blackAttack` mátrix kezdetben üres, később kerülnek feltöltésre, miután meghatároztuk az összes figura lehetséges lépését.

Célállapot

A célállapot vizsgálatát az `isTerminal()` logikai függvény végzi. Ez minden állapothoz kiértékelődik, és amíg visszatérési értéke `false`, addig folytatható a játék. A következő esetekben vált `true`-ra:

- a soron következő játékos nem tud szabályosat lépni, magyarul a lehetséges operátorokat tartalmazó vektor üres.
- egyik fél sem rendelkezik mattadó erővel. Lehetetlen mattot adni, ha már csak két király van a táblán, két király és mindkét félnek egy-egy könnyűtisztje van a táblán, vagy az egyik játékosnak csak királya, a másiknak pedig a király mellett két huszárja maradt. A sakkzsargon erre az esetre használja a „reménytelenül döntetlen” vagy „holt remi” kifejezést.
- az utolsó 50 lépésben nem történt ütés vagy gyaloglépés. Ennek vizsgálatára leginkább egy numerikus típusú számláló használható, amelynek értéke nulláról indulva minden lépés során nő eggyel, ha pedig ütés vagy gyaloglépés történik, akkor nullázódik. Ha a számláló eléri az 50-et, a program felajánlja a döntetlent.

```
public boolean isTerminal() {
	int i = 0;
	if ( possibleStep.isEmpty() ) return true;
	if ( lastStepWithPawn == 50 ) return true;
	if ( figures.size() == 2 ) return true;
	if ( figures.size() == 3 )
		for ( Figure f: figures )
			if ( f.getType() == 'b' || f.getType() == 'B' ||
				f.getType() == 'n' || f.getType() == 'N' )
					return true;
	if ( figures.size() == 4 )
		for ( Figure f: figures )
			if ( f.getType() == 'n' || f.getType() == 'N' ) i++;
	return i==2 ? true : false;
}
```

Miután az `isTerminal()` metódus megállapítja a játék végét, az `isWinner()` metódus megmondja a végeredményt:

```
public int getWinner(){
      if ( possibleStep.isEmpty() ) {
            if ( isChess ) return next == 0 ? 1 : 0;
            else return -1;
      }else
            return -1;
}
```

Ha nincs lehetséges lépés, és a király sakkban van a vizsgált állásban, akkor a lépni következő játékos vesztett, tehát a másik játékos kódjával kell visszatérni. Minden más esetben döntetlen az eredmény, melynek visszatérési értéke -1.

Sokszor előfordul, hogy egy játékos komoly anyagi fölénnyel kezdi a végjátékot. Ilyenkor élő játékosok esetén szinte mindig feladja a vesztésre álló fél, ritka kincs egy sakkozónak, amikor mattot tud adni. Ez persze nem azért teszik, hogy megfosszák ellenfelüket a mattadás örömétől, sokkal inkább az unalmas, kilátástalan végjáték felesleges lejátszását kerülik el vele. Komolyabb sakkszoftverek rendelkeznek ilyen „önkritikával”, és biztosan vesztes helyzet esetén feladják a mérkőzést. Más programokban nincs ilyen komponens, de statisztikát vezetnek a nyert és vesztett játszmákról, és ez az embert szinte rákényszeríti a játszma mattig való lejátszásához, holott ebben már csak kevés élmény van. A számítógép még hosszú percekig számolva megtalálja a legjobb lépést, de az eredményt már borítékolni lehet.

A számítógép célállapotnak tekinthetné a kilátástalan állapotokat is, például azokat a végjátékokat, amelyekben már nincs gyalogja a gép által irányított félnek, és ellenfele legalább egy könnyűtiszt előny mellett rendelkezik még mattadó erővel.

Ennek vizsgálatára tett kísérleteim nem jártak különösebb sikerrel. Fő okai a sikertelenségnek a csaliként bevetett áldozatok. A sakktörténelemben számos olyan játszma van, amelyben valaki vezért áldoz, de lépések múlva ez az áldozat meghozza a gyümölcsét, és megnyeri a játszmát. Ha ez az áldozat a program keresési mélységen túl válik javunkra, akkor a számítógép tévesen állítaná egy állapotról, hogy vesztes.

Operátorok

A sakkban háromfajta lépés jelenik meg, amelyek megvalósításához három operátort használ a program:

- hagyományos lépés, amikor a lépő bábu egy üres, vagy egy az ellenfél által birtokolt mezőre lép. Utóbbi esetben a lépést ütésnek hívjuk, de mindkét variáció kezelhető a `Step` operátorral.

  Az operátor hatása: `table`-nak annak a mezőjére, ahova a lépés vezet, beírjuk a lépő figura típusát, míg a kiinduló mezőre ’0’-t írunk, jelezve, hogy ott már nem áll figura.

  Az operátor alkalmazási előfeltétele: ha egy operátor már létrejött, akkor biztos, hogy a lépés nem vezet le a tábláról és hogy a célmezőn nem áll saját figura, valamint hogy a lépő figura a saját specifikus lépését végrehajtva a célmezőre szabályosan léphet. Az egyetlen dolog, amit ellenőrizni kell, hogy az alkalmazás után ne legyen sakkban a lépő fél királya.
- menet közbeni ütés (en Passant[9]). Ez a gyalog lépése. Ekkor mindig üres mezőre lépünk, de mégis leütünk egy ellenséges bábut. Az operátor neve `EnPassant`.

  Az operátor hatása: Legyen egy világos gyalog pozíciója (x,y). Az operátort erre a gyalogra alkalmazva, a `table` (x-1,z) mezőjére ’P’, a `table` (x,z) mezőjére pedig ’0’ kerül, ahol z mindkét helyen vagy y-1, vagy y+1. Sötét esetén a hatás hasonló.

  Alkalmazási előfeltétel: természetesen itt is vizsgálni kell, hogy az alkalmazás után ne legyen sakkban a lépő fél királya. Ezenkívül, egy (x,y) mezőn álló világos gyalog esetén teljesülnie kell annak az előfeltételnek, hogy az ellenfél a megelőző lépése során vagy az (x,y-1), vagy az (x,y+1) mezőre lépett gyalogjával, méghozzá dupla lépést téve.
- sáncolás: A sáncolás azért különleges lépés és azért igényel új operátort, mert alkalmazásakor nemcsak egy figura helyzete változik meg. Ezt végzi a `Castle` operátor.

  Az operátor hatása: a világos király *c1* (vagy *g1*) mezőre kerül, míg az *a1* (*h1*) mezőn álló bástya a *d1* (*f1*) mezőre. Sötét esetén a hatás hasonló.

  Alkalmazási előfeltétel: a sáncolás végrehajthatóságáról a 11. és 12. oldalon már esett szó.

[9] http://www.fide.com/component/handbook/?id=124&view=article 3.7-es szabály, d. pont

Állapotok értékelése

A heurisztikus kiértékelő függvény minden állapothoz egy numerikus értéket rendel. Ezt az értéket az állapotok minősítésére használhatjuk, segítségével az állapotok között kisebb, nagyobb vagy akár egyenlő relációkat értelmezhetünk. Ennek alapján a lentebb tárgyalt keresőalgoritmusok könnyebben megtalálhatják a mindenkori legjobb, vagy legjobbnak vélt állapotot, és így azt az operátort választják ki a sok közül, ami ennek a jónak vélt állapotnak a létrehozásához vezet. A kiértékelő függvény azért heurisztikus, mert nem pusztán matematikai számolások alapján dönt egy állapot értékéről, hanem tapasztalatok, ökölszabályok sorát használja fel. Programomban a heurisztikus kiértékelő függvény olyan, hogy ha egy állapotra nullával tér vissza, akkor az állapot által reprezentált állás vizsgálatunk szempontjából döntetlen. Minél nagyobb ez a szám, annál jobb az adott állás világos számára, sötétnek pedig a negatív értékek kedveznek.

A heurisztikus függvényt két, egymásnak némiképp ellentmondó metrika jellemzi. Az egyik a pontosság, azaz hogy az aktuális állapothoz adott érték mennyire felel meg a valóságnak, mennyire írja le jól az állapotot. Például ha világosnak bástyaelőnye van, akkor valószínűleg ő áll jobban, így a heurisztikus függvénynek pozitív értékkel kellene visszatérnie, de kisebbel mintha vezér, és nagyobbal mintha huszárelőnye lenne.

A másik jellemző a kiszámíthatósági idő. Ha a játékfában öt szint mélységig keresünk előre, akkor olykor 3-5000 állapotot is meg kell vizsgálnunk. Ha a minden állapothoz kiszámítandó heurisztikus függvény csak egy tized másodperccel lassabb egy másiknál, akkor akár percekkel kell többet várnunk a legjobb operátor megtalálásához.

Nyilvánvaló, hogy minél pontosabban jellemezzük az állapotot, annál bonyolultabb heurisztikára, és így több kiértékelési időre van szükségünk. Kompromisszumot kell kötnünk.

Becslési szempontok

Meg kell vizsgálni, milyen metrikákkal jellemezhető jól egy hadállás. Ezek között lesz olyan, amelyre könnyen írhatunk kiértékelő függvényt, például a táblán lévő figurák anyagi értékeinek összege. Ekkor a még táblán lévő bábuk anyagi értékét kell összeadni, ez viszonylag gyorsan megvalósítható, és nagyban befolyásolja az állás összértékét. Van azonban olyan metrika, aminek megvalósítása nehéz, és nem biztos, hogy megéri. Ilyen lehet a gyalogok struktúrája, amin akár a győzelem is múlhat, de szinte lehetetlen hatékonyan vizsgálni. A heurisztikus kiértékelő függvény valójában egy súlyozott lineáris függvény, felírható a következő alakban:

$$\mathcal{H}(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

Ez azt jelenti, hogy egy *s* állapothoz rendelt összérték több súlyozott érték összegeként áll elő. Legnagyobb súlyt a legfontosabb szempont kapja, majd a súlyok egyre csökkenek. Végezetül elérkezünk egy olyan kis súllyal rendelkező taghoz, aminek kiszámolása már több erőforrást igényelne, mint amennyire a kiszámolt érték befolyásolná az összértéket, így nem éri meg foglalkozni vele.

Anyag, tér, idő

Ezzel a hármassal jellemezhető talán legjobban egy hadállás értéke. Az anyag a táblán lévő figurák anyagi értékének összege, a tér a megtehető lépések száma, az idő pedig a hátralévő idő. Mindegyik nagyon fontos jellemző, és egyiket se bonyolult megvalósítani.

Az egyes figurák anyagi értékét általánosságban az határozza meg, hogy a sakktáblán mekkora területre fejtenek ki hatást, hány különböző mezőre tudnak egy bizonyos helyről lépni, illetve ütni. A király más elbírálás alá esik, hiszen elvesztése a játszma végét jelenti, így egy abszolút értékű figura. Állítsunk fel egy bábut egy teljesen üres sakktábla egy mezőjére, nézzük meg, hány lépést tehet. Majd sorban az összes mezőre ismételjük meg a vizsgálatot. A kapott eredményeknek pedig egyszerűen vegyük a számtani közepét. A bástya az egyetlen bábu, amely a sakktábla bármely mezőjéről ugyanannyi helyre léphet, a többi figura miatt szükséges, hogy ne egyetlen mező, hanem az egész sakktábla átlagát számoljunk. Nézzük meg ennek alapján az egyes figurák értékét: a bástya minden mezőről 14 mezőre léphet, így a

bástya értéke 64 * 14 / 64 = 14. Van olyan mező, amelyről a vezér 27 helyre léphet, de egy sarokmezőről csupán 21-re, az átlag 22,75.

vezér	22,75
bástya	14,00
futó	8,75
huszár	5,25
gyalog	2,91

1. táblázat A figurák anyagi értéke

Ezt a teljesen elméleti számítást a sakkgyakorlat nagyjából igazolta, kivéve a futó és a huszár értéke között mutatkozó különbséget. A huszár azon értékét, hogy váltakozva tud világos és sötét mezőkre lépni, illetve hogy teli táblán is tud lépni (nem állhatják el az útját, mint egy futónak) számtanilag nem tudjuk kifejezni. Ezt figyelembe véve a program a huszár esetében 7,25 értékkel számol.

Az anyagi előny csak egy segítőeszköz a játék megnyerése érdekében, mely lehetővé teszi a sokszor döntőbb térelőny megszerzését és biztosítását. Hiszen mattadáshoz még vezérelőny esetében is az ellenfél királyának sokszor teljes leszorítására van szükség, a tágabb térben mozgó figurák együttműködése pedig anyagi előnyszerzést tesz lehetővé. Így a nagyobb terület birtoklására irányuló törekvésnek az egész játszma során a szemünk előtt kell lebegnie. Ha tehát két olyan állás közül kell választani, ahol az anyagi érték egyforma, de az egyik esetben a rendelkezésre álló tér nagyobb, akkor azt kell választanunk.

17. ábra Capablanca-Treybal

A 17. ábrán Capablanca Treybal elleni játszmájának egy állása látható. (1929, Karlsbad). Sötétnek ugyan nincs anyagi hátránya, de egyszerűen nem tud mit lépni. Futója és huszárja teljesen be van zárva, bástyáit a vezér rossz helyzete miatt nem tudja az *a* vonalra vinni, jelenlegi helyzetükben pedig teljesen inaktívak. Világos a bástyáit néhány lépésen belül felsorakoztatja az *a* vonalon, majd *a7*-en megtámadja a *b7* gyalogot, amit sötét nem tud megvédeni. A térelőny így hamarosan anyagi előnnyé fog válni.

A tér számolására jó módszernek tűnik, ha világos megtehető lépéseinek számából kivonjuk sötét megtehető lépéseinek számát. Ekkor nem vizsgáljuk, hogy sakkban áll-e valamelyik fél, hiszen abból hogy sakkban áll a király, és csak egyetlen helyre léphet, még nem következik, hogy a sakkadó félnek térelőnye van. A 17. ábrán világos 45 lépést tehet, míg sötét mindössze 22-t.

Az idő a még rendelkezésre álló időt jelenti. Sakkjátékokban minden fél azonos gondolkodási időt kap, ezt okosan kell beosztani a játszma végéig. Egy kiértékelő függvény gyakran paraméterezhető, hiszen ha már csak néhány perc maradt hátra, és még közel sincs a parti vége, akkor bizony gyorsítani kell a számításokat, magyarul egyszerűsíteni a kiértékelést. Ez elérhető a keresési mélység csökkentésével, vagy a kiértékelés egyes részeinek elhagyásával. Egy sakkprogramnak nem illik időhiány miatt veszítenie, ha már csak egy perce van hátra, akkor szinte azonnal lép, nyilvánvalóan ekkor nem lesz annyira átgondolt a lépése, mint egy öt perces számítás után.

Programom nem veszi figyelembe az időt, a kiértékelő függvény úgy van implementálva, hogy bármilyen bonyolultságú állásnál egy-két percen belül döntsön a gép (természetesen ez hardverfüggő).

18. ábra Kasparov - Karpov 2009

Az egész sakkvilág nagy érdeklődéssel várta Kasparov és Karpov két egykori sakkvilágbajnok újabb összecsapását. Erre 2009-ben került sor Valenciában. A kép a negyedik játszma[10] utolsó jelenete, a 33. lépésben vagyunk, és sötét lép. A nyerési esélyek közel egyformák, bár elemzők szerint világos az egy oszlopon álló két bástyának köszönhetően talán valamivel jobban áll, de Karpov minden bizonnyal kiharcolhatta volna a döntetlent. Ekkor azonban váratlan esemény történt, sötét zászlója leesett[11] és így a játszmát elvesztette. Kasparovnak még rengeteg ideje maradt. Összefoglalva tehát világos átlagosan gyorsabban tudott dönteni és ebben a játszmában ez elég volt a győzelemhez.

[10] http://live.chessdom.com/kasparov-karpov-game-4.html
[11] elfogyott az ideje

Másodlagos szempontok, heurisztika

Már csak az anyag-tér-idő hármas figyelembe vételével meglepően jól lehet értékelni egy hadállást. Arról nem is beszélve, hogy ezek nem igényelnek bonyolult számolásokat, hiszen az anyagi érték számításakor mindössze a figura típusát kell tudnunk, a tér számításakor pedig óriási segítség az objektumorientált szemlélet (minden figura önálló objektum, és tudja, hogy hova léphet). A figurákat tartalmazó kollekció egyszeri bejárásával így megkaphatjuk mind az anyagi értéket, mind a lehetséges lépések számát.

Ha mindössze erre a két vizsgálatra támaszkodna minden sakkprogram, akkor a kiértékelések eredményei közel egyformák lennének, és csak a keresés mélysége, vagy a jobb futtató számítógép döntené el a játék kimenetét (azonos adatbázisok mellett). A sakkprogramok tudásában azonban óriási különbség van, ez pedig arra utal, hogy vannak még más értékelési szempontok, amit egy-egy programban implementálnak, másokban pedig nem.

Az anyag-tér-idő hármasban nincs benne semmilyen tapasztalat, semmi ökölszabály. Nem tartalmazza azokat a stratégiákat és trükköket, amiket a sakkozók évszázadok során alakítottak ki, pusztán egyszerűen, matematikai úton értékelik az adott állást. A másodlagos szempontok azonban figyelembe vesznek már ilyen tapasztalatokon nyugvó, matematikai eszközökkel nehezen, vagy egyáltalán le nem írható tulajdonságokat is. Összefoglalva azokat az egyszerűsítéseket, trükköket, stratégiákat, vagy bármilyen olyan eszközt, amit előnyünkre fordíthatunk egy állapot kiértékelésekor heurisztikának nevezzük. Ezek a tulajdonságok is nagyban befolyásolják az állás értékét, de hogy milyen súllyal az vita tárgyát képezi. Az elsődleges szempontokkal nem lehet vitatkozni, minden sakkot már játszó ember elismeri azok fontosságát. A heurisztikák értéke és súlya, egyáltalán a heurisztikák figyelembe vétele azonban sakkozóról sakkozóra változhat. Heurisztika például, hogy egy olyan hadállás, amin mindkét futója megvan az egyik félnek erősebb annál a hadállásnál, ahol a futók helyett két huszárral rendelkezik. A futók jobban összedolgozhatnak, ez egy tapasztalatokon nyugvó megállapítás, aminek van némi alapja, de nem bizonyítható egyértelműen. Így ennek igazságtartalmával nem mindenki ért egyet, például Réti sakknagymester, a múltszázad egyik nagy sakkozója huszárjátékáról volt híres, ő nem szívesen cserélte huszárjait futókra.

Vizsgáljunk most meg olyan másodlagos heurisztikákat, amelyeket bonyolultságuk vagy csekély súlyuk miatt nem építenek be minden sakkprogramba:

a) a centrum elfoglalása. A centrum a sakktábla közepén lévő négy mező (e4, e5, d4 és d5). Könnyen belátható, hogy a centrum uralása térelőnyhöz vezet.
b) többet ér két egyforma könnyűtiszt, mint két különböző. Ha két futónk, vagy két huszárunk van, akkor általában ezekkel jobban össze tudunk dolgozni, mint ha egy-egy lenne mindkettőből.
c) egy oszlopon lévő két gyalog (duplagyalog) nehezen védhető. Gyalogokat sokszor bástyával védünk, és könnyen belátható, hogy ha két gyalog van egy oszlopon, akkor csak a hátsót tudjuk megfelelően védeni.
d) a figurák elhelyezkedése. Egy *a* vagy *h* oszlopon lévő gyalog általában kevesebbet ér, mint a közbenső oszlopokon lévők, hiszen kevesebb mezőt tudnak támadni. Egy sarokban lévő huszár mindössze két helyre léphet, míg a tábla közepén álló akár nyolc helyre is.
e) a gyalogok struktúrája. A gyalogok kiválóan tudják egymást védeni, sőt egy gyalog számára az a legjobb, ha egy társa védi, mert ilyenkor az ellenfél csak gyaloggal ütheti (ha tiszttel ütné, akkor anyagi hátrányt szenvedne). A gyalogstruktúrát tehát úgy kell felépíteni, hogy gyalogjaink egymást védjék, és ne szétszórtan helyezkedjenek el.
f) két bástya egy vonalon. Ha két bástya van egy soron, az általában nem jelent előnyt, viszont két bástya egy oszlopon rendkívül erős tud lenni.
g) az utolsó gyalog védelme. Ne feledkezzünk meg a gyalogok másodlagos lépéséről, az átváltozásról. Az utolsó gyalog elvesztése megfoszthat minket a nyeréstől, hiszen nem egy gyalogot, hanem az utolsó potenciális vezért vesztjük el. Ha a táblán a király mellett már csak egy futó, vagy két huszár van, akkor nem adhatunk mattot (feltéve hogy az ellenfél nem követ el valami baklövést). Tehát az utolsó gyalog elméletileg több pontot ér, mint egy futó (viszonylag üres táblán).
h) a bástya mozgását a gyalogstruktúra általában erősen korlátozza. Minél kevesebb gyalog (úgy saját, mint ellenfél) marad a pályán, a bástya értéke annál több lesz. A huszár a speciális lépése miatt viszont pont a bonyolult gyalogstruktúrák gyengítésére ideális.

A megvalósított kiértékelő függvény

A program által ténylegesen alkalmazott kiértékelő függvény öt tag összege alapján állítja elő a végső értéket. Mivel tapasztalatokon nyugvó tagok is szerepelnek ebben az ötben, így elmondható, hogy a kiértékelő függvény heurisztikus. Négy tag kiszámolásához a sakktábla teljes bejárása szükséges, ezért ezek vizsgálatát célszerű párhuzamosan végezni, elkerülve a felesleges ciklusokat. A kiértékelés során egy-egy numerikus változót felhasználva külön számoljuk ki a világos és sötét fél értékét. Világos értéke a `wHeur` nevű változóban, míg sötété a `bHeur`-ban kap helyet. Az állapot összértéke ennek a két változónak a különbsége lesz, ami minden esetben egy egész szám.

A legfontosabb szempont és így a legnagyobb súllyal rendelkező tag a figurák anyagi értékének összege. Ez független a figura helyzetétől, ha egy bizonyos típusú figura a táblán van, akkor a figura színének megfelelő változó értéke, a figurának az 1. táblázatban szereplő értékének százszorosával nő. A király nem szerepel a táblázatban, anyagi értéke 30000, ami azért ilyen magas, hogy minden áldozatot elkövessen a kereső a király megvédésének érdekében (ha például öt lépéspár mélységig keres a számítógép, és minden lépésünkkor lenyerünk egy vezért, akkor is csak $2275 \cdot 5 = 11375$ lenne a vesztesége, tehát inkább a királyt fogja védeni, aminek elvesztése jóval nagyobb veszteség lenne). Ez egy pusztán matematikai szempont, nem tapasztalatokon alapszik.

A második szempont a figurák elhelyezkedését veszi alapul. Ennek a tagnak a számolására előre definiált mezőértékelési mátrixokat használ a program, ez a mátrix minden egyes figuratípus esetén különböző értékekkel van feltöltve. A kódrészletben a világos gyalog mátrixa látható:

```
private static byte pawn [][] = {{ 0,  0,  0,  0,  0,  0,  0,  0 },
                                 { 70, 80, 90,100,100, 90, 80, 70 },
                                 { 60, 70, 80, 90, 90, 80, 70, 60 },
                                 { 50, 60, 70, 80, 80, 70, 60, 50 },
                                 { 20, 40, 50, 70, 70, 50, 40, 20 },
                                 { 15, 30,-10,  0,  0,-10, 30, 15 },
                                 { 15, 10, 10,-70,-70, 10, 10, 15 },
                                 {  0,  0,  0,  0,  0,  0,  0,  0}};
```

A mátrixban megadott értékek azt fejezik ki, hogy ha az adott típusú bábu az adott mezőn áll, akkor mennyivel csökkentsük, vagy növeljük az állapot értékét. Ha egy világos gyalog áll az *e2* mezőn, akkor -70 pont adódik a `wHeur`-hoz, ami nem jó világos számára. A számítógép így ösztönözve van az *e2*-tőn álló gyalog elmozdítására. Láthatjuk, hogy *e4* helyen már +70 adódna a világos értékéhez, így ha megnyitáskor *e2*-*e4*-et lép világos, akkor a `wHeur` máris 140 ponttal nő, ami nagyjából egy gyalog anyagi értékének a fele. A mátrix feltöltése nem egyszerű feladat. Gyalogok esetén olyan szempontokat kell figyelembe venni, hogy egy adott mezőn állva mennyire akadályozza a saját figurák mozgását, mennyire rontja sötét struktúráját és milyen közel van az átváltozáshoz, azaz a nyolcadik sorhoz. Mivel az első és nyolcadik soron nem állhat gyalog, így az ott lévő érték sose kerül felhasználásra, értéke bármi lehet. Az *e4* és *d4* nyitások kapták a legtöbb értéket (a mátrix ötödik és hatodik sorában ez a legmagasabb érték). Így ezek a kezdések támogatva vannak. A kezdeti felállásból egy gyalog akkor lépi a legrosszabbat, ha *c3*-ra vagy *f3*-ra lép, ugyanis ilyenkor meggátolja a huszár fejlődését, így ezen mezők értéke a legalacsonyabb, -10. Világos huszár esetén a mátrix értékei teljesen mások:

```
private static byte knight [][] = {{-50,-40,-30,-30,-30,-30,-40,-50 },
                                   { -40,-20,  0,  0,  0,  0,-20,-40 },
                                   { -30,  0, 10, 15, 15, 10,  0,-30 },
                                   { -30,  5, 10, 20, 20, 10,  5,-30 },
                                   { -30,  0, 15, 20, 20, 15,  0,-30 },
                                   { -30,  5, 15, 15, 15, 15,  5,-30 },
                                   { -40,-20,  0,  5,  5,  0,-20,-40 },
                                   { -50,-20,-30,-30,-30,-30,-20,-50}};
```

Egy huszár számára az a jó mező, ahonnan nyolc helyre léphet, tehát a közbenső mezőknek nagyobb értéket kell adni, mint egy a tábla szélén lévő mezőnek, míg a négy sarokmező a lehető legrosszabb.

Ez a szempont már ténylegesen tapasztalatokon nyugvó értékekkel dolgozik, hiszen a mátrixok feltöltése teljesen önkényes, a saját tapasztalataimra és ösztöneimre támaszkodva lettek megadva az értékek. Vannak, akik egy-egy ritka megnyitásra esküsznek, és szerintük a világos gyalog nagyon jó helyen van a *c3* mezőn, mások szeretik, ha a futók a *b2* és *g2* mezőkről támadják a főátlókat, tehát a futó mátrixában ennek a mezőnek adnak magasabb értéket. Mivel a mátrixokban szereplő legmagasabb érték 150, így egy figurának a legjobb

helyzete megegyezik a leggyengébb figura anyagi értékének kb. felével. Magyarul a második szempont ugyan beleszól a teljes heurisztika értékébe, de az első szempont súlya sokkal nagyobb, így a program mindig az anyagi értéket fogja leginkább szem előtt tartani.

A sötét figuráknak nem kellett saját mátrixot készíteni, esetükben az értékek megegyeznek a világos társaik mátrixának tükrözéséből vett értékekkel.

A királyok két mátrixot is kaptak, ugyanis egy király helyzetére a következő ökölszabály érvényes: a játszma elején, amíg sok bábu van a pályán a király számára legfontosabb a biztonság, így a legjobb hely a gyalogok által védett alapsor, azon belül is a sáncolással elérhető *c1* vagy *g1* mező. Végjátékokban azonban a királynak elsősorban az a feladata, hogy elfoglalja a tábla közepét, és biztosítsa az átváltozni igyekvő gyalogok útját. Tehát a király mátrixának értéke attól függ, hogy végjátékról, vagy nem végjátékról van szó. A végjáték egyszerűen és gyorsan vizsgálható, ha a pályán lévő figurák anyagi értékeinek összege már kisebb egy előre megadott értéknél, akkor végjátékról beszélünk, ellenkező esetben nem.

A harmadik és negyedik tag nagyon hasonló. A teljes táblát bejárva, ha egy oszlopon két azonos színű gyalogot találunk, akkor csökkentjük az adott színű játékos változóját, mert ez előnytelen, kerülendő álláshoz vezet. Ilyenkor 150 pont kerül levonásra.

Ha az oszlopokat vizsgálva azt tapasztaljuk, hogy az egyiken két azonos színű bástya áll, akkor némiképp jutalmazni kell az adott állapotot a bástyákat birtokló fél javára. Egy duplabástyás sor 200 többletpontot jelent.

Az ötödik tag a `whiteAttack` és `blackAttack` mátrixok mezőinek összértéke alapján mond valamit az állapot értékéről. Mindkét mátrixot bejárva meghatározzuk a mezőkön szereplő számok összegét. Ez a szám `whiteAttack` esetén megadja, hogy a világos bábuk összesen mennyi mezőt támadnak vagy védenek, és ezt ötszörös szorzóval hozzáadjuk a világos értékét tároló `wHeur`-hoz. Sötét esetén hasonlóan járunk el.

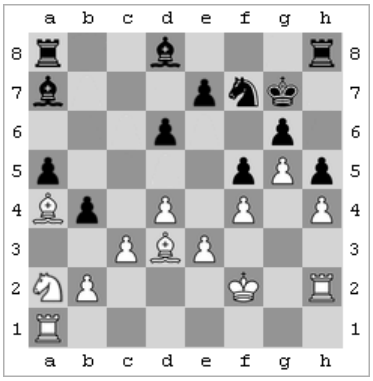
```
        for ( int i = 1; i < 9; i++ )
            for ( int j = 1; j < 9; j++ ) {
                  wHeur += chessBoard.getWhiteAttack()[i][j] * 5;
                  bHeur += chessBoard.getBlackAttack()[i][j] * 5;
            }
```

A megvalósított kiértékelő függvény segítségével olyan értéket sikerült minden állapothoz rendelni, ami magában foglalja a fentebb tárgyalt fő szempontok közül az anyag és a tér vizsgálatát, valamint a másodlagos szempontok közül az a, c, d és f pontokat.

Értékelési különbségek az egyes variánsokban

A fent leírt kiértékelő függvény elsősorban hagyományos sakkhoz készült, de jól alkalmazható a másik két variáns esetén is, némi módosítással. A függvény négy tagja minden variáns esetén megállja a helyét, hiszen a figurák anyagi értéke, vagy sötét és világos megtehető lépéseinek különbsége nem függ a kezdőfelállástól, ha az szimmetrikus. A két bástya egy oszlopon minden variáns esetén előny, míg a dupla gyalog hátrány, tehát ezen a szemponton sem kell változtatni a variánsok esetén. Amit viszont változtatni kellene, az a figurák helyzetének értékelési mátrixa. Hagyományos sakknál nagyon erősen támogatja a program a *d2* és *e2* (sötét esetén *d7* és *e7*) mezőkön lévő gyalogok *d4*-re és *e4*-re (*d5-re, e5-re*) mozgatását, hiszen ilyenkor teret kapnak a futók és a vezér, valamint a gyalogok a centrumot is elfoglalják. Egy véletlen felállásnál azonban már nem biztos, hogy ezek a lépések ennyi előnnyel járnak. Az igaz, hogy a centrumot megtámadják, de lehet, hogy a mögöttük lévő mezőkön nem futók és vezér áll, hanem huszár, így a lépés megtétele nem jár térelőnyhöz jutáshoz. Az is lehet, hogy egy *e4* nyitás esetén a király erősen támadhatóvá válik. A véletlen felállításoknál tehát nem a középső gyalogokat kellene támogatni, hanem mindig azokat, amelyek lehetővé teszik a futók aktívvá válását, esetleg segítik a bástyák csatatérre jutását. Ezt vizsgálni azonban nehéz, hiszen Fischer-sakknál 960, random sakknál pedig több ezer kezdőállás létezik, mindegyiknél más és más lenne a legjobb kezdeti gyaloglépés. Így nem is változtattam a figurák helyzetének értékelési mátrixán, mert a gyalogok kezdőlépéseit leszámítva a játék további részére már variánstól függetlenül érvényesek a mátrixban szereplő értékek. Egy huszárnak bármely variáns esetén hátrány a tábla szélén állni, a királynak bármely variáns esetén előny a védet tisztsoron maradni, stb.

Random sakk esetén előfordulhatnak olyan kezdőállások, amik megkövetelik a figurák anyagi értékének módosítását. Ha a két futó azonos színű mezőn mozog, akkor a két futó védeni tudja egymást, közös mezőt tudnak támadni, ami egy a hagyományos sakkban nem létező, de komoly erőt jelenthet. Természetesen az ellenfélnek is azonos színű mezőn fognak állni a futói, csak pont az ellentétes színeken. Így egy eltolódás jön létre, az egyik fél a világos, míg a másik fél a sötét mezők felett tesz szert komoly uralomra. Random sakknál tehát az egyszínű futók anyagi értékét növelni kellene, legalább egy fél gyalog értékével. Ha azonban elveszítjük egyik futónkat, akkor nem csak a futó az áldozat, hanem a másik futó partnere is. Így futóvesztéskor a másik futó erejét vissza kell állítani a hagyományos sakkban szereplő megfelelő értékre.



19. ábra Azonos színen álló futók

Az egy színen álló két futó nagy ellenfele lehet a megfelelően felépített ellenséges, és a rosszul felépített saját gyalogstruktúra. A 19. ábrán világosnak két fehér, míg sötétnek két fekete futója van. Világos úgy szervezte a gyalogjait, hogy azok lerontsák ellenfele futóinak hatását, gyakorlatilag elzárják előlük a tábla egy jelentős részét, míg a saját futók mozgását egyetlen egy mezőn sem blokkolják. Ugyanakkor sötét futóit saját gyalogjai is korlátozzák. Ezek a gyalogok viszont nem képeznek akadályt világos számára, világos mindkét futója akadálytalanul közlekedhet a tábla szinte egészén. Természetesen a gyalogstruktúra számmá alakítása random sakk esetén is roppant bonyolult feladat, így ezzel nem is kísérleteztem.

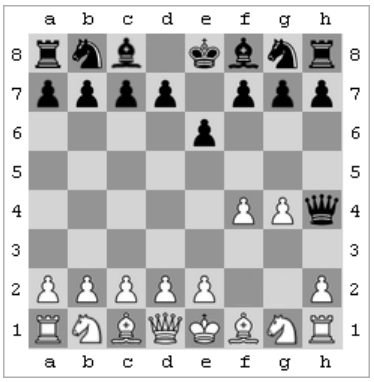
Lépésajánló algoritmusok

Most hogy minden állapothoz értéket tudunk rendelni, vizsgáljuk meg, hogy hogyan tudjuk ezeket felhasználni. A lépésajánló algoritmusok felépítenek egy játékfát a memóriában. A fa gyökere a kezdőállapot, majd az első szint csúcsaiban azok az állapotok szerepelnek, amelyek létrejöhetnek egy a gyökérelemre alkalmazható operátor felhasználásával. Aztán az első szint állapotaival is hasonlóan járunk el, mindegyikükre végrehajtjuk az összes alkalmazható operátorát. A játékfa levélelemei lesznek a célállapotok. A játékfa páros szintjein a kezdőjátékos (jelöljük A-val) lép, míg a páratlanokon az ellenfele (B). Egy ilyen teljes játékfa a játék összes játszmájának ábrázolására alkalmas. A kereső feladata a teljes játékfa ismeretében egy nyerő stratégia keresése. Ehhez a fa csúcsait a levélelemektől kezdve, felfelé haladva megcímkézi. A levélelemek A vagy B címkét kapnak, attól függően, hogy A vagy B játékos nyer az adott célállapotban. Egy nem levél csúcs esetén a kereső annak figyelembevételével dönt a címkézés során, hogy melyik játékos lép a csúcsot tartalmazó szinten. Ha A lép és van olyan kivezető ág a csúcsból, mely A-val van címkézve, akkor A lesz a csúcs címkéje, hiszen A-ról feltételezzük, hogy a legjobbat lépi, magyarul olyan csúcsot választ, amelyben A nyer. Ha nincs A-val címkézett kivezető csúcs, akkor az aktuális csúcs címkéje B lesz. Azokon a szinteken, ahol B lép hasonlóan járunk el. Az eljárást a fa minden csúcsára alkalmazzuk, utoljára a gyökér fog címkét kapni. A gyökér címkéje megadja azt a játékost, amely számára létezik nyerő stratégia, mert mindig tud olyan utat választani az ellenfele lépéseitől függetlenül, amely a saját címkéjével jelzett célállapothoz vezet. Teljes játékfa esetén nincs szükség az állapotok kiértékelésére, csak címkékkel kell őket ellátni, és a vázolt algoritmussal nyerő stratégia található.

A teljes játékfa azonban nem minden játék esetén építhető ki. Kisebb, egyszerűbb játékoknál gyorsan kiépíthető, így könnyedén megtalálhatjuk a legjobb stratégiákat. Azonban minél bonyolultabb egy játék, annál bonyolultabb lesz az állapotokból felépülő játékfa. Egy hagyományos sakkjáték esetén 169 518 829 100 544 000 000 000 000 000 módja van mindkét játékos első tíz lépésének[12]. Ebből a számból világosan látszik, hogy sakkjáték esetén a teljes játékfa kiépítése lehetetlen, de még egy húsz szint mélységű játékfa kiépítésére is csak kevés számítógép képes. Tehát a játékfa teljes mélységű kiértékeléséről és ezzel egy biztosan nyerő stratégia meghatározásáról le kell mondanunk. Csak egy bizonyos, a hardvertől és a

[12] http://deepblue.lib.umich.edu/bitstream/2027.42/50736/2/1984-winter-dividend-text.pdf

még élvezhető döntési sebességtől függő mélységig tudjuk kiépíteni a játékfát, melynek levélelemei így nem, vagy csak ritkán lesznek végállapotok.



20. ábra A bolondmatt

Ha a hagyományos sakk kezdőállását tekintjük gyökérnek, akkor egy négyszintes játékfában több ezer levélelem lesz, de ezek közül csak egyetlen egy lesz célállapot, hiszen négy lépésben csak az úgynevezett „bolondmatt[13]" esetén érhet véget a játék.

Ha tehát nem építhető ki a teljes játékfa, akkor nem a nyerő stratégia meghatározása a cél, hanem általában a lehető legjobb első lépés megtalálása, minden lépés után. Olyan részfa épül fel minden lépés után, amelynek gyökere az aktuális állapot, első szintjén pedig az aktuális állapotra alkalmazható operátorok alkalmazásával elérhető állapotok lesznek. A második szinten olyan állapotok szerepelnek, melyek az első szint valamely állapotára végrehajtott alkalmazható operátor útján jönnek létre, stb. Így itt nem igaz az, hogy a páros szinten a kezdőjátékos lép, részleges játékfánál mindig az aktuálisan lépni következő játékos kerül a gyökérbe, és így a páros szintekbe. Jelen esetben tehát nem építjük fel a teljes játékfát, hanem egy bizonyos szinten megállunk. Ezt a szintet általában előre megadjuk a keresőalgoritmusnak, ami elérve a megadott szintet befejezi a játékfa építést. De szint helyett megadható idő- vagy memóriakorlát is. Az első esetben a keresés rögzített mélységben zajlik bármely lépés után, míg a másik két esetben előfordulhat, hogy egyes lépések után nagyobb mélységű fa építhető ki. Ha kész az adott mélységű részfa, akkor hasonlóan a teljes játékfánál tárgyaltakhoz, itt is a levélelemek címkézésével kell kezdeni a minősítést, majd felfelé haladva, a gyökérnek utoljára címkét adva kapjuk meg a teljesen felcímkézett fát. Csakhogy mivel nem a teljes játékfa van felépítve, így a levélelemeknek jelen esetben nem adható A vagy B címke, ugyanis nem célállapotok. A levélelemekben található állapotok a heurisztikus kiértékelő függvény segítségével kapnak értéket, ami így egy szám lesz. Minél nagyobb ez az érték, annál jobb az állapot A játékos számára, és minél kisebb annál jobb B számára. Ha

[13] http://hu.wikipedia.org/wiki/Bolondmatt, a leggyorsabban beadható matt

minden levélelemet kiértékeltünk, akkor továbbléphetünk egy fentebbi szintre, és ott a keresőalgoritmusunktól függően több módon járhatunk el. A minimax eljárást alkalmazva például a következőképpen járunk el: mindkét játékosról feltételezzük, hogy nyerni akar, tehát a számára lehető legjobbat lépi. Így ha a szinten A játékos lép, akkor a csomópont értéke a kivezető értékek közül a legnagyobb lesz. Ha B játékos lép, akkor pedig a legkisebb. Legvégül a gyökér is kap egy értéket, és hasonlóan a teljes játékfához, ilyenkor is leolvasható, hogy melyik játékosnak van nyerő stratégiája. Ha a gyökérben pozitív szám van, akkor A játékosnak, negatív szám esetén B-nek. Ez a nyerő stratégia azonban nem az egész játékra vonatkozik, hanem az adott mélységben kiépített részfára. Magyarul, ha egy x szintű játékfában, a gyökérelem egy pozitív értéket kap, akkor elmondható, hogy a legenerált x mélységben A játékosnak van nyerő stratégiája, de lehet, hogy x+1 szintnél már B játékos állna nyerésre.

A program a minimax eljárást valósítja meg, aminek egyik hátránya, hogy a játékfa egyes részfái feleslegesen kerülnek legenerálásra. Ezt azonban alfa-béta eljárással egyszerűen sikerült orvosolni. A program keresési mélysége beállítható, a tesztelések során az alapértelmezett 3 mélységtől kezdve lépett olyat, ami élvezhetővé teszi a játékot, tehát védi a figuráit, sakkot ad, látványosan előregondolkozik. Az 1 és 2 mélységet csak inkább a lépések megtanulására tudja használni a felhasználó, tényleges kihívást nem jelent. Lehetőség van 4 és 5 keresési mélységben való játékra, ahol a válaszlépés még másfél percen belül megtörténik, ennél nagyobb keresési mélység viszont már túl nagy gondolkodási időt igényel.

A programról

A program neve „Pawn”, ami angolul a gyalogot jelenti. A kódolás elkezdése előtt úgy gondoltam célszerű lenne egy objektumorientált nyelvet választani, mert annak előnyei itt kiaknázhatóak, így esett a választásom a Java-ra. A számos hasonló figura, és hasonló lépés miatt rengeteg öröklődést tudtam használni, nagyban csökkentve a kód méretét. A figurák objektumként való kezelése is jó ötletnek bizonyult, így minden egyes figura ismeri a színét, helyzetét, valamint a lehetséges lépéseit és ezek könnyedén elérhetőek. Persze megjelentek a szemlélet hátrányai is, a rengeteg példányosítás roppant időigényes futtatáskor.

Minden egyes bábu a `Figure` ősosztályból származtatható.

```
public abstract class Figure {

      protected PointPair pos;
      protected char type;
      protected Vector<PointPair> possibleMoves = new Vector<PointPair>();

      public Figure( PointPair pos, Character type ){
         this.pos = pos;
         this.type = type;
      }

      public Figure( Figure f ) {   }

      public int getAnyag( char type ) { … }

      public Character getColor() {
                  return type > 96 ? 1 : 0 ;}

      public int canMove( char[][] table, int x, int y ){

          if ( this.getPos().getX()+x < 1 || this.getPos().getX()+x > 8 )
              return 0;
          if ( this.getPos().getY()+y < 1 || this.getPos().getY()+y > 8 )
              return 0;

          char next = table[this.getPos().getX()+x][this.getPos().getY()+y];

          if ( next == '0' )  return 1;
          if ( next > 96 && this.getType() > 96 )     return 0;
          if ( next < 96 && this.getType() < 96 )     return 0;

          return 2;
      }
      public abstract void calculateMoves(char[][] table);

      …
}
```

A program során szükségünk lehet a figurák pozíciójára, típusára, színére, anyagi értékére és a megléphető lépéseik kollekciójára, valamint mindegyikhez hozzá van rendelve egy kép is, amire a játék során majd egérrel tudunk kattintani. Ezek a tulajdonságok némiképp függnek egymástól, így a konstruktornak mindössze két paraméter kell ahhoz, hogy bármilyen típusú bábut létre tudjon hozni.

```
public Figure( PointPair pos, Character type ){…}
```

A figura színét a típusa egyértelműen meghatározza, a világos bábuk nagybetűvel, a sötétek kisbetűvel vannak jelölve, ha tehát a `type` értéke 96-nál nagyobb, akkor biztosan sötét a figura.

```
this.color = type > 96? 1 : 0;
```

A kép könnyen megadható a szín és a típus ismeretében a `color+type+.gif` módon, ugyanis például egy sötét huszár képe a bn.gif, míg a világos vezér képe a wq.gif fájlban található. Az anyagi érték szintén a típustól függ. Végeredményben tehát szinte az összes lekérdező metódus a `type` értékét használja fel.

```
public ImageIcon getPic() {
       return new ImageIcon("data/figures/iron/"+(
            this.getColor()==0?'w':'b')+type+".gif");
}
```

A `Figure` egy absztrakt osztály, olyan metódusai vannak, melyeket az egyes gyermekosztályoknak kell megvalósítani. Ilyen a `calculateMoves()` metódus, ami a lehetséges lépések kiszámításához alkalmazható, ezt természetesen különböző típusú figurák esetén másképp kell felüldefiniálni. A `canMove()` segít egy-egy figura lehetséges lépéseinek számolása közben eldönteni, hogy a figura a lépést megtéve egyáltalán a táblán maradna-e, ha igen akkor saját vagy ellenfél által birtokolt mezőre lépne-e. Az ősosztály után nézzük most meg az egyik tényleges figuratípus osztályát is:

```
public class Knight extends Figure{

	public Knight( PointPair p, Character type ) {
		super( p, type );
	}

	@Override
	public void calculateMoves( Board chessBoard ) {

		possibleMoves.clear();
		int ret;

		ret = this.canMove(chessBoard.getTable(), 2, 1);
		if( ret != 0 ) {
			if ( this.getColor()==0 )
				chessBoard.setWhiteAttack(this.getPos().getX()+2 ,
							this.getPos().getY()+1);
			else chessBoard.setBlackAttack(this.getPos().getX()+2 ,
							this.getPos().getY()+1);
			if ( ret != -1 )  possibleMoves.add( new PointPair(
							this.getPos().getX()+2 ,
							this.getPos().getY()+1 ));
		}

		ret = this.canMove(chessBoard.getTable(), 2,-1);
		if( ret != 0 ) {
			if ( this.getColor()==0 )
				chessBoard.setWhiteAttack(this.getPos().getX()+2 ,
							this.getPos().getY()-1);
			else chessBoard.setBlackAttack(this.getPos().getX()+2 ,
							this.getPos().getY()-1);
			if ( ret != -1 )  possibleMoves.add( new PointPair(
							this.getPos().getX()+2 ,
							this.getPos().getY()-1 ));
		…
		}
```

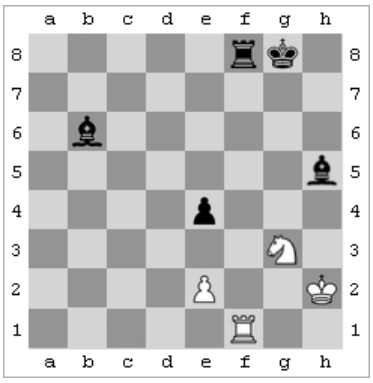
Mint látható egy huszár a pozíciójával és a típusával hívja meg szuperosztálya konstruktorát. A `calculateMoves()` metódus felüldefiniálása kötelező, hiszen ez absztrakt metódusa az ősosztálynak. A lehetséges lépések kollekcióját kiürítjük, majd a `canMove()` metódus segítségével feltöltjük a lehetséges lépésekkel. A `canMove()` egy felállított táblát vár, és azokat a koordinátákat, melyekről el kell dönteni, hogy oda léphetünk-e. Huszár esetén a saját pozícióhoz képest a (2,1), (2,-1), (-2,1), (-2,-1), (1,2), (1,-2), (-1,2) és (-1,-2) mezők jöhetnek szóba, ahol a (2,1) számpár azt jelenti, hogy a soron lépünk kettőt lefelé, az oszlopon pedig egyet jobbra. A `canMove()` hívása a legtöbb figuránál ciklussal megoldható, hiszen a figurák lépéseiben szabályszerűség van. Huszár esetében azonban

célszerű a `canMove()`-ot mind a nyolc lehetséges lépéssel cikluson kívül meghívni. A `canMove()` az átadott elempárt hozzáadja a hívó bábu aktuális pozíciójához, és így megkapja a vizsgálandó mező koordinátáit. Három értékkel térhet vissza:

- 0-val, ha a lépés során lemennénk a tábláról, vagy ha ugyan a táblán maradnánk, de mégse léphetünk a célmezőre, mert már saját figuránk áll rajta.
- 1-gyel, ha üres mezőre lépnénk
- 2-vel, ha a célmezőn ellenséges figura áll

A `canMove()` nem tudja eldönteni, hogy a lépés megtétele lehetséges-e, azaz megtétele után nem lenne-e sakkban a lépő játékos királya.

A `possibleMoves` vektort a `canMove()` által szolgáltatott értékek segítségével már fel tudjuk tölteni. Ha az érték nulla, akkor nem lehetséges a lépés, ha egy vagy kettő, akkor huszár esetén igen. Az üres és ellenség által megszállt mezőket azért kell külön vizsgálni, mert vannak olyan bábuk, például a gyalogok, melyek csak akkor léphetnek, ha a célmezőn nincs semmilyen figura, tehát ha a `canMove()` egyet ad vissza.



21. ábra Huszár lépéseinek vizsgálata

Az ábrán egy végjáték látható, világosnak van egy huszárja a (6,7) mezőn. A valóságban ez a (6,7) mező a sakktábla *g3* mezőjének felel meg. A különbség onnan ered, hogy a sakktábla számozása a bal alsó saroktól, az *a1* mezőtől kezdődik és oszlopfolytonos, míg a megvalósítás során könnyebb volt a hagyományos tömbkezelést alkalmazni, azaz az *a8* mező lett az (1,1) indexű, és sorfolytonosan következnek a többiek. A tömbindexek valóságtól eltérő kezeléséből a felhasználók természetesen semmit nem vesznek észre, kiírásoknál szinkronizálás történik egy egyszerű mechanizmussal:

```
public static String synchronize( PointPair p ) {

      return (char)(p.getY()+96) + "" + (9-p.getX());

}
```

Ha az ábrán lévő (6,7) huszárra meghívjuk mind a nyolc lehetséges `canMove()` metódust, akkor a következőt tapasztaljuk:

- két esetben 1-gyel tér vissza a metódus (két üres mezőre léphetünk).
- szintén két esetben, (-2,1) és (-1,-2) paraméterek esetén 2-vel tér vissza, ekkor hajtanánk végre ütést.
- a többi esetben nullát ad a metódus, két esetben ((2,-1),(1,-2)) azért, mert saját figura van a célmezőn, (1,2) és (-1,2) esetén pedig elhagynánk a pályát.

A játék során, a sakktáblán mezőin négy szín jelenhet meg:

```
    public void setColor( char color ) {
            switch( color ){
                  case 'w': setBackground(java.awt.Color.white);break;
                  case 'b': setBackground(java.awt.Color.black);break;
                  case 'y': setBackground(java.awt.Color.yellow);break;
                  case 'p': setBackground(java.awt.Color.pink);break;
            }

    }
```

A mezők fehér vagy fekete alapszíne mellett felvehetik a sárga vagy rózsaszín színt is. Egy mező akkor lesz sárga színű, ha a rajta lévő figura van kijelölve, és azok a mezők színe vált rózsaszínre, amelyekre a kijelölt figura szabályosan léphet.

A program több üzenetet is felugró ablakban jelenít meg:

```
      if ( ! src.hasFigureOn() ) {
            showUpErrMsg("Ezen a mezőn nincs figura!"); return;
      }else if ( src.getFigureOn().getColor()!=chessBoard.getNext() ) {
            showUpErrMsg("Az ellenfél figurájával nem lehet lépni!");
            return;
      }
```

Ilyen például, amikor az egérrel kijelölt `src` mezőn nincs figura, vagy figura ugyan van rajta, de az nem a sajátunk, így nem léphetünk vele. De nem csak hibaüzenetekkor jelenik meg a felugró ablak, a program a játék végeredményét is így közli a felhasználóval (lásd: Függelék).

A főprogramnak a következő adattagjai vannak:

```
public static Board chessBoard;
public PointPair honnan;
public PointPair hova;
public static Player white = new Player( "Játékos", true );
public static Player black = new Player( "Számítógép", false );
public Move mo;
public static int depth = 3;
public static int depthw= 3;
public static Step op = null;
public static Vector<TablePiece> srck = new Vector<TablePiece>();
public static Vector<TablePiece> pink = new Vector<TablePiece>();
public static Vector<Step> operator = new Vector<Step>();
public int gameType = 0;
```

Természetesen tartozik hozzá egy *chessBoard* nevű sakktábla, aminek konstruktorát a gameType értéke alapján hívjuk majd meg. A világos bábukat alapértelmezés szerint a felhasználó irányítja, míg sötéttel a számítógép lép. Ez azonban tetszőlegesen kombinálható, lehet mindkét színnel számítógép, és mindkét színnel élő játékos is. Utóbbi esetben a program mindössze a lépések szabályosságát vizsgálja. Számítógép számítógép ellen típusú játékban a két fél keresési mélysége különböző lehet, ezt a *depth* és *depthw* változók tárolják. A pink vektorban az aktuálisan éppen rózsaszínre és sárgára színezett mezők vannak, így az eredeti szín visszaállításához elég ezt a vektort bejárni, nem kell mind a 64 mezőt külön megvizsgálni.

Összefoglalás

Sokan intettek attól, hogy a sakkot válasszam dolgozatom témájának, hiszen ez az egyik legbonyolultabb táblás játék. Válasszak inkább valami könnyebbet, dámát vagy amőbát, esetleg ha mindenképp sakktáblán akarok maradni, akkor rókavadászatot, vagy más, kevés bábus, rövid játékot. Ezzel szemben számomra is meglepő eredményeket értem el, a kezdeti célkitűzések többségét sikerült megoldanom, és most már a fentebb említett játékok nem okoznának komoly nehézséget.

Persze a program tekintetében még bőven van javítási, és bővítési lehetőség. Nem gondoltam volna, hogy a sakkheurisztikáknak ekkora, úgy magyar, mint külföldi irodalma van, és bizony legtöbbjüket nem egyszerű megvalósítani. Úgy gondolom, hogy munkám a szakdolgozat beadásával nem ér véget, mindenképp meg szeretném valósítani a kedvenc sakkvariánsaimat (lavinasakk, defenzív sakk), amelyekről bonyolultságuk miatt itt nem is esett szó. Ahogy egyre mélyebbre ástam magam a heurisztikák között, a programírást többször is újra kellett kezdenem, hiszen amikor már leprogramoztam egy elég jó, és viszonylag könnyen megvalósítható heurisztikát, akkor mindig akadt egy gyorsabb, egy szimpatikusabb változat, vagy egy jobb ötlet. Kellemes meglepetés, hogy a program többször is tudott nyerni a tesztelések folyamán, ez megerősített abban, hogy jó úton haladok.

Nem túl távoli célok között szerepel még a hálózati kommunikáció, és egy kisebb adatbázis beépítése a programba, de régóta foglalkoztat az a gondolat, hogy szálkezeléssel rengeteg idő nyerhető, hiszen amíg az ember lép, addig a számítógép csak tétlenül várakozik, holott az ember lépését megsejtve előre is gondolkozhatna.

A legtöbb időt a grafikus felület megtervezésére és megvalósításra fordítottam, ezzel eddig csak keveset foglalkoztam, és bizony a Pawn írásakor sokszor álltam fel a géptől úgy, hogy az Eclipse-ben több osztály volt pirossal aláhúzva. Sok időt elvett a mesterséges intelligencia megvalósításától, de ez egy másik olyan témakör, amibe úgy érzem sikerült beletanulnom a dolgozatom írása alatt. Talán nem kellett volna a grafikus felület, és tovább jutok a heurisztikák vizsgálatában, de az volt már a kezdeti véleményem is, hogy egy sakkprogram egérkezelés és színes tábla nélkül nem igazán élvezhető.

Irodalomjegyzék

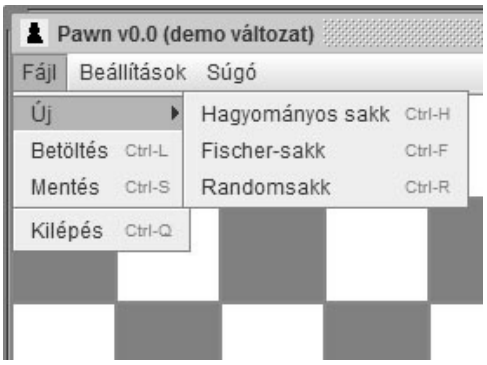
Könyvek:

1. Asztalos Lajos - Bán Jenő:
   A sakkjáték elemi (Kossuth Kiadó, 2005)
2. Kállai Gábor - Szabolcsi János:
   Sakk ünnepnapokra és hétköznapokra (Alfadat-Press Kiadó)
3. Stuart Russell - Peter Norvig:
   Mesterséges intelligencia modern megközelítésben (Panem Kiadó, 2005)
4. Futó Iván (szerk.):
   Mesterséges intelligencia (Aula Kiadó, 1999)
5. Várterész Magda:
   Mesterséges intelligencia (Egyetemi jegyzet, 2009)

Web címek:

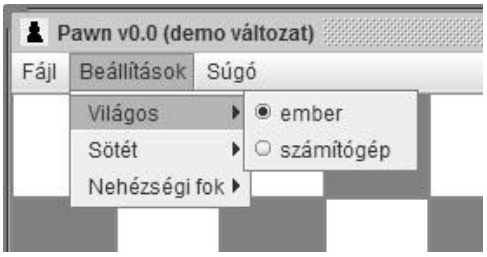
1. Fischer Random Chess:
   http://www.dwheeler.com/essays/Fischer_Random_Chess.html
2. Chess Variant Applets:
   http://www.pathguy.com/chess/ChessVar.htm
3. Legéndi Richárd Olivér:
   http://leriaat.web.elte.hu/forraskodok.html
4. Dividend – The Magazine of the Graduate School of Business:
   http://deepblue.lib.umich.edu/bitstream/2027.42/50736/2/
   1984-winter-dividend-text.pdf
5. Wikipedia:
   http://en.wikipedia.org/wiki/Chess960
   http://hu.wikipedia.org/wiki/Fischer_random_sakk
   http://en.wikipedia.org/wiki/Chess
   http://hu.wikipedia.org/wiki/Bolondmatt

Függelék



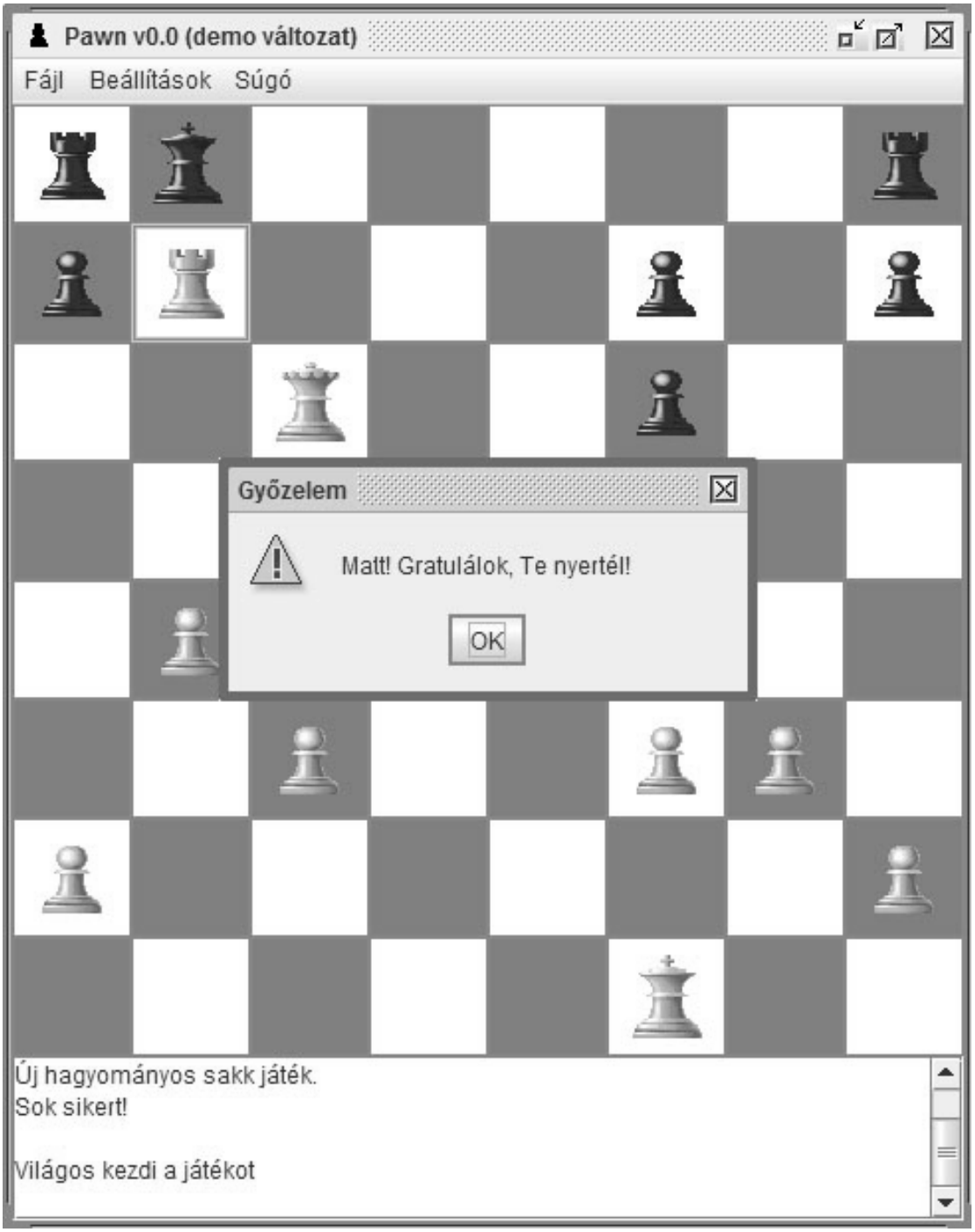
1. kép Fájl menü

2. kép Nehézség kiválasztása

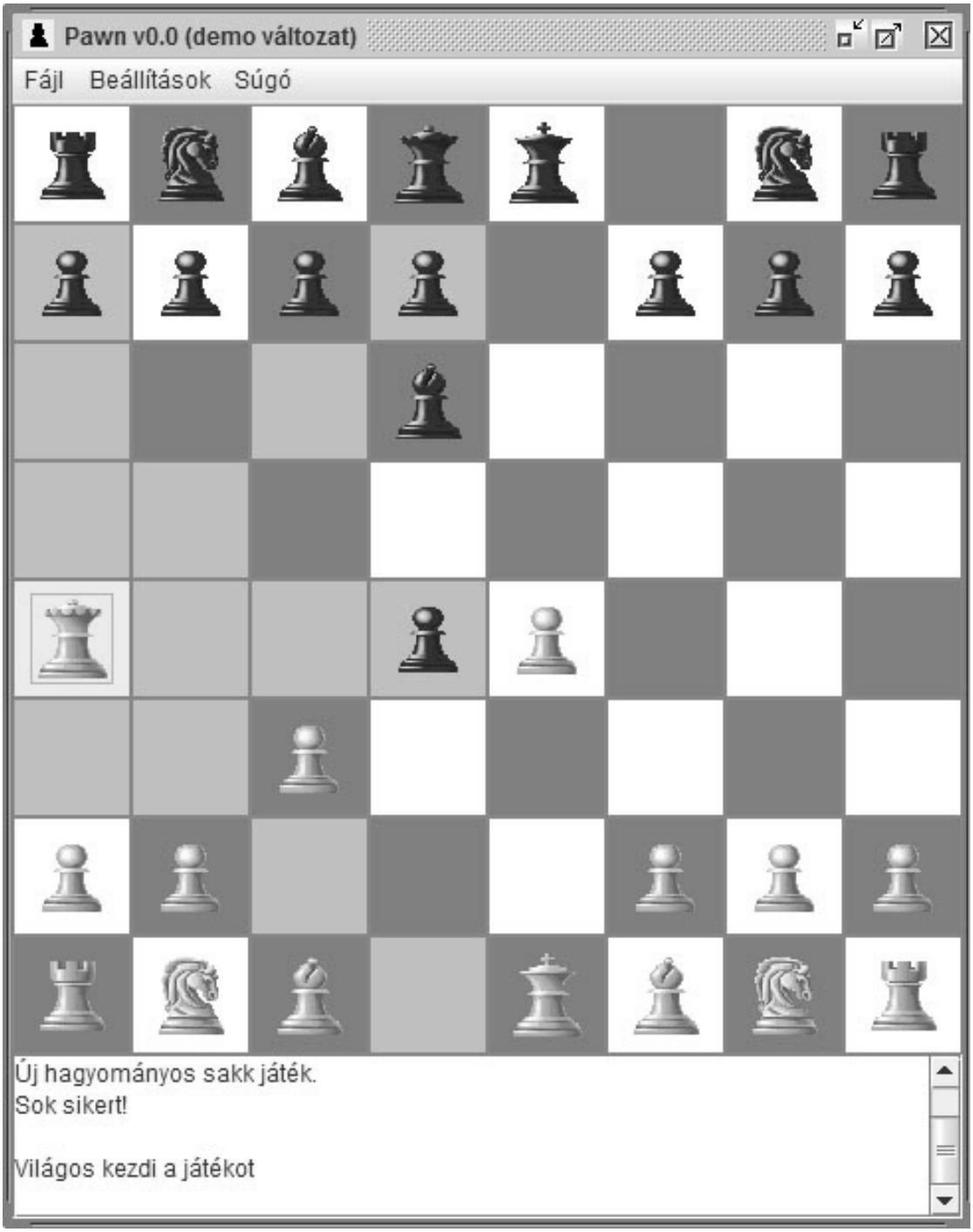


3. kép Ellenfél kiválasztása

A menük, és az egyes menüpontokhoz rendelt billentyűkombinációk segítségével könnyen elérhetőek az alapvető funkciók és beállítások.

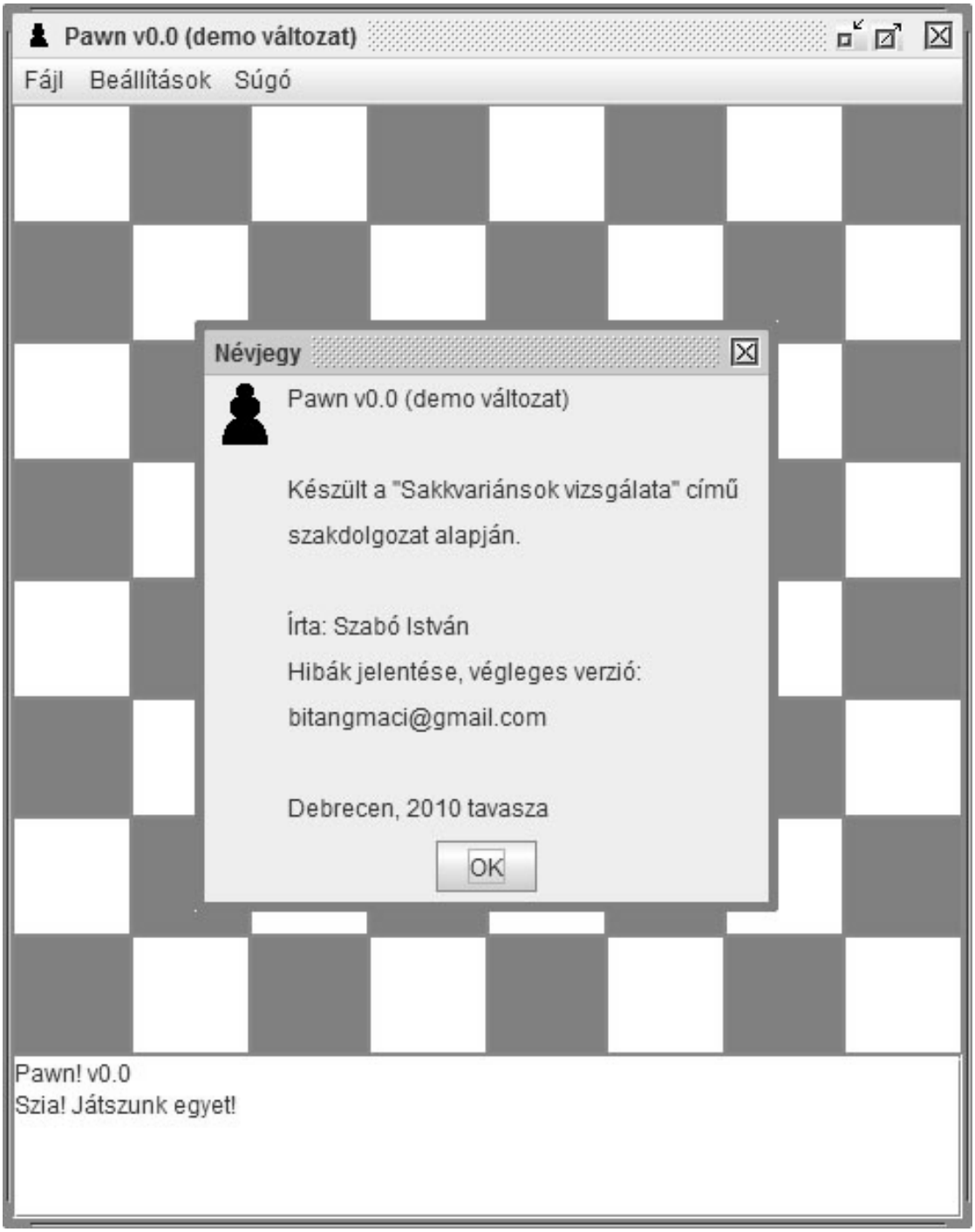


4. kép Egy mattkép



5. kép A kijelölt vezér lehetséges lépései

6. kép A huszár sakk miatt csak két helyre léphet



7. kép A program névjegye