

DIPLOMAMUNKA

Pelsőczy János Pál

Debrecen
2010.

Debreceni Egyetem
Informatikai kar
Számítógéptudományi Tanszék

Sakkfeladványok készítése

Témavezető:
Mecsei Zoltán Pál
Egyetemi tanársegéd

Készítette:
Pelsőczy János Pál
Programtervező informatikus

Debrecen
2010.

1 . Bevezetés	5
2. Sakkfeladvány készítő és megoldók	6
3. A sakk elemei	8
3.1 Sakkbábuk és értékük.....	8
3.2 Speciális lépések.....	14
4. Sakkfeladványok.....	17
4.1 A sakkfeladványokról.....	17
4.2 Sakkfeladványok típusai	18
4.2.1 Nyílt vagy közvetlen matt feladványok (directmates).....	18
4.2.2 Önmatt feladványok (selfmates)	19
4.2.3 Reflexmatt feladványok (reflexmates)	21
4.2.4 Segítőmatt feladványok (helpmates)	22
4.2.5 Sorozatlépéses feladványok (seriesmovers)	25
4.2.6 Tündérfeladványok (feary problems)	26
5. Sakk és a mesterséges intelligencia	27
5.1 Mesterséges intelligencia alapfogalmak.....	27
5.1.1 Állapottér-reprezentáció	27
5.1.2 Közvetlen elérhetőség, elérhetőség, megoldás, költség.....	28
5.1.3 Állapottérgráf	29
5.2 Megoldást kereső rendszerek	29
5.2.1 Backtrack keresők	32
5.2.2 Keresőfával kereső rendszerek	34
5.3 Lépésajánló algoritmusok	37
6. Saját fejlesztésű sakkfeladvány készítő és megoldó program JAVA nyelven.....	38
6.1 A programban használt sakkfeladvány elemek.....	39
6.2 Grafikus megjelenítés	41
6.3 Beállítások	42

6.4 Figyelők.....	44
6.4.1 ButtonListener	44
6.4.2 MenuListener	46
6.4.3 PopupListener.....	47
6.5 Állásmentés és állásbetöltés.....	48
6.6 Állapottér reprezentáció megvalósítása	49
6.7 A programban használt keresők	52
7. Összegzés	54
8. Irodalomjegyzék.....	55
9. Függelék.....	56

1 . Bevezetés

„A sakk az értelem próbaköve.” - Goethe

Előszóként szeretném elmondani, hogy a „Sakkfeladványok készítése” című diplomamunkámra hogyan esett a választásom. Egyetemi éveim alatt a sok tárgyat tekintve a logika, formális nyelvek, mesterséges intelligencia vonalat mondhatnám kedvencemnek, persze a programozás mellett. Már régóta szerettem volna ezekkel kapcsolatos témakörben vizsgálódni, ilyen jellegű programot írni. Már tanulmányaim elején sakkkal kapcsolatos programokat írtam C nyelven vagy hálózaton keresztül való kommunikációval a kliens által küldött sakkállapotra a szerver adta meg a választ, mindezt JAVA nyelven. A sakk jelen volt tanulmányaim alatt, így diplomamunka témám is ilyen irányban determinisztikus volt. A sakkfeladványok témaköre számomra érdekes és új is, a sakktól elválaszthatatlan, így ebben a témában szerettem volna vizsgálódni. Ehhez a témához sok segítséget nyújtott a formális nyelvek és a mesterséges intelligencia tárgyak elsajátítása, ami erőteljes lökést adott a választáshoz.

A bevezetés után ismertetem a sakkfeladvány készítőik által kialakult különféle készítési stílusokat, majd a sakk elemeit, milyen bábuk vannak, és hogyan kell velük lépni a sakktáblán. A következő nagy fejezetben magáról a sakkfeladványokról írok, ami dolgozatom alaptémája. Ismertetni fogom a sakkfeladványok lényegét, a sakkfeladványok típusait egy-egy példával bemutatva, ami segít megérteni az adott típusú sakkfeladvány sajátosságait.

Egy külön fejezetben ismertetem a sakk és mesterséges intelligencia kapcsolatát, ismertetem az alapfogalmakat, amik elengedhetetlenek az algoritmusok és a keresők megértéséhez, majd részletezem a programomban használt keresőket, hogy hogyan is működnek.

Zárásként bemutatom saját fejlesztésű programomat, ismertetem programom sakkfeladvány elemeit. Részletezem a használt eszközöket a programban és azt, hogy hogyan valósítottam meg a keresést és a reprezentációt.

2. Sakkeladvány készítőik és megoldók

Sakkeladványok esetén beszélhetünk készítőkről (composer), akik a sakkeladványokat és más sakktanulmányokat készítik. Általában egy bizonyos típusú sakkeladványra specializálódnak, mint például kétlépéses, háromlépéses, segítő matt, stb. feladványokra. Minden készítőnek megvan a maga saját készítési stílusa, amit kompozíciós tanulmányoknak (composition school) hívnak. Ezek a tanulmányok különböző szempontok szerint vizsgálják egy sakkeladvány jelenlegi helyzetét, állapotát.

A legismertebb kompozíciós tanulmányok:

- **régi német** (old German school) - a megoldás nehézségére és összetettségére helyezi a hangsúlyt és a megoldásban úgynevezett modell mattot ¹(model mate) alkalmaznak, a négy - és ötlépéses feladványoknál elterjedt
- **cseh** (Bohemian school) – a művészi szépségre és a variációk számára fekteti a hangsúlyt, ahol modell mattot alkalmaznak, három – és négylépéses feladványoknál elterjedt
- **angol** (English school) – szabad játékot igényel minden variációban, ami azt jelenti, hogy szabadon akárhova lehet lépni legális lépéssel. A sokféle megoldásra fekteti a hangsúlyt.
- **amerikai** (American school) – az eredetiségre és a meglepő elemek jelenlétére helyezi a hangsúlyt
- **új német vagy logikai** (new German school, logical school) – a megoldás logikai struktúrájára és a cél tisztaságára épít
- **új cseh** (new Bohemian school) – a cseh és a logikai ötvözése
- **stratégiai** (strategical school) – a sok variáció összetettségére épít, általában két – és háromlépéses feladványoknál használják
- **szovjet** (Soviet school) – a stratégiai tanulmány magas szintre fejlesztett változata
- **új stratégiai** (new strategical school) – a variációk változtatását igényli
- **szlovák** (Slovakian school) – a fázisok közötti úgynevezett motifok² változtatását igényli

¹ A modell matt egy speciális mattolási állás, amiben a mattot adó fél minden bábujának részt kell venni a matt adásban (kivéve a legálisan elhelyezkedő gyalogok és a király). Segítő – és ön mattoknál elterjedt.

Manapság a készítőket nem egy, hanem több tanulmány alapján készítik feladványokat.

Beszélhetünk még a készítőket mellett megoldókról (solver) is. A megoldók minél több feladvány megoldására törekednek, és nem arra, hogy minél több feladványt készítsenek.

A készítőket és megoldók között nincs határvonal, természetesen vannak olyanok is, akik mindkettőben elismertek.

Nagymester fokozattal rendelkező készítőket (grandmaster composers) a függelék 1. táblázata tartalmazza, a megoldókat (grandmaster solvers) pedig a 2. táblázat.

² A motív egy lépés alapeleme, olyan szempontból, hogy miért kell lépni az adott bábuval vagy éppen a lépés mennyire tesz eleget a kikötéseknek. A motívek lehetnek pozitívak (támadó, gyengülő) és lehetnek negatívak (védekező, ártalmas). Például egy negatív ártalmas motív világos lépés esetén egyben pozitív sötét motív.

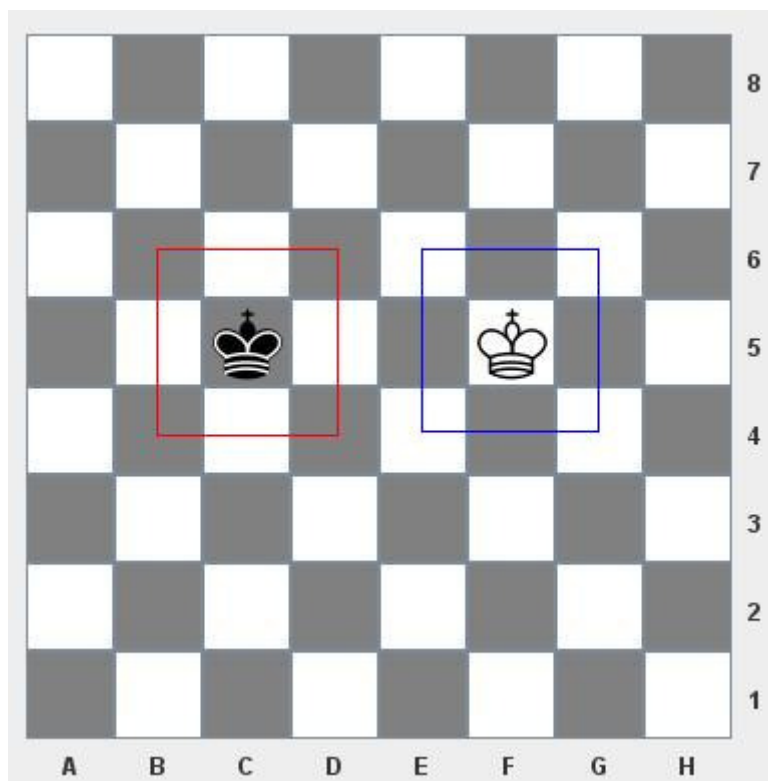
3. A sakk elemei

„Minden lépés egy terv eleme” – Capablanca

3.1 Sakkbábuk és értékük

A király (King)

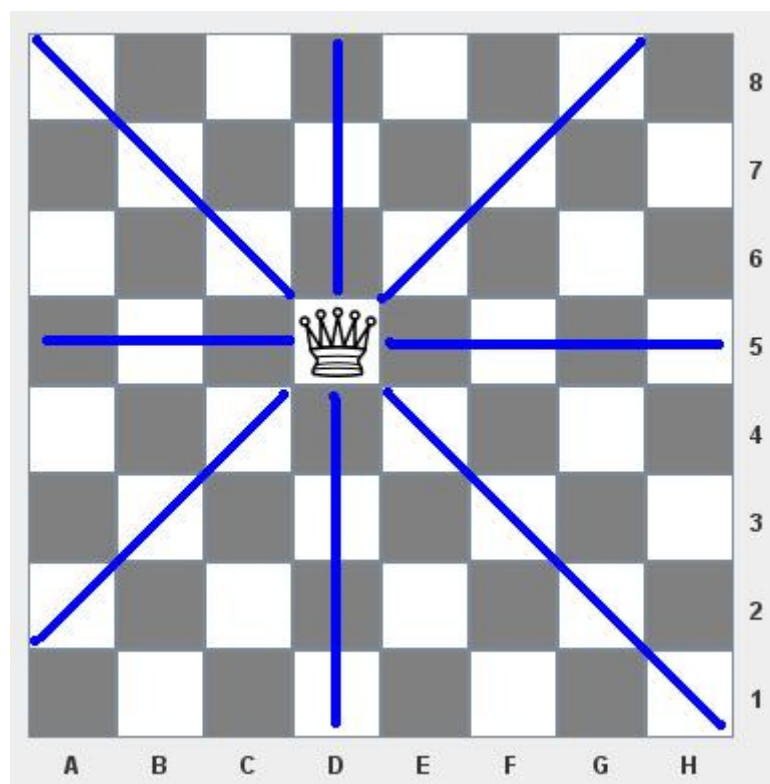
A legfontosabb figura, a sakkban és ugye a sakktudományokban is a király bemutatója a lényeg. Bármely irányban (vízszintesen, függőlegesen, átlósan) léphet egy mezőre, feltéve hogy azon a mezőn nincs saját bábú. Szintén nem léphet olyan mezőre, ami után sakkban lenne, sakkba nem lehet lépni. Nem léphet még olyan mezőre, ami után királyközelítés következik be (a két ellenséges király nem állhat egymás mellett) . Ezeket a korlátozásokat figyelembe véve a király legálisan léphet bárhová. A szabályok szerint a király nem üthető le, csak sakkot lehet adni, és ha már nincs olyan mező ahová legálisan léphet, akkor a játéknak vége sakk matt vagy patt formájában. Mivel a király a legfontosabb figura ezért az értéke is a legmagasabb. A király értéke a legnagyobb, minden más bábú értékénél magasabb kell, hogy legyen. Jele: K.



A király lehetséges lépései

A vezér (Queen)

A vezér a legerősebb figura a sakkban, minden irányban léphet akármennyi mezőt, addig, míg saját bábuba nem ütközik. A legtöbb lépési lehetősége általában a vezérnek van, a bástya és a futó tulajdonságával is rendelkezik egyszerre, így lépéseinek számolásigénye igen nagy. A szabály szerint kezdetben csak 1 db vezérünk van, de maximum lehet 9 is gyalogátváltatással³. A királyon kívül a legértékesebb figura. A vezér értéke: 9. Jele: Q.



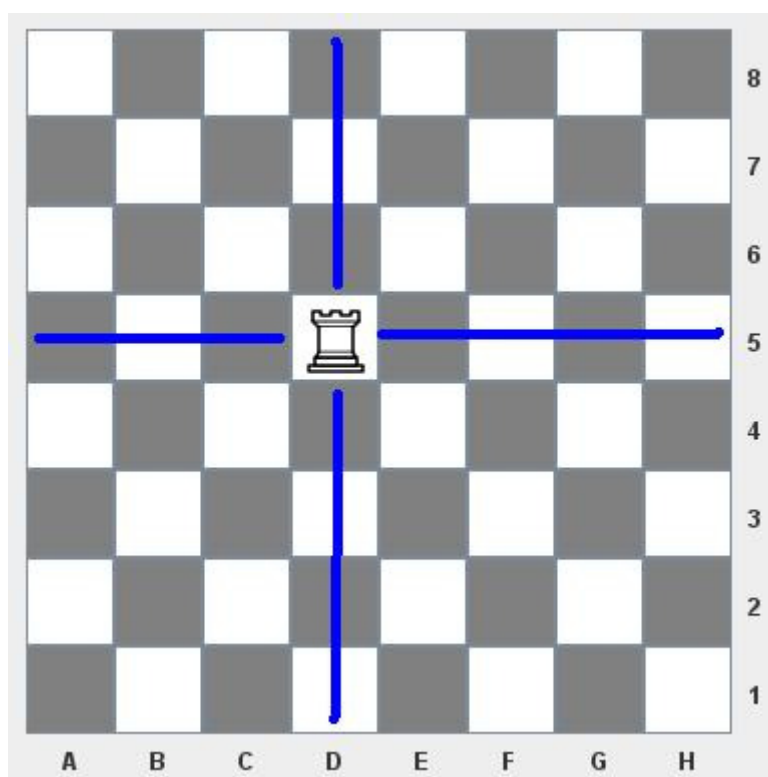
A vezér lehetséges lépései

³ Gyalogátváltatásról akkor beszélünk, amikor eléri az irányának megfelelő utolsó sort. Ekkor kötelezően átváltozik egy általunk választott tisztté (vezér, bástya, huszár, futó).

A bástya (Rook)



A bástya vízszintesen és függőlegesen léphet bármennyit, míg el nem éri a tábla szélét vagy saját bábuba nem ütközik. Lépéseinek száma körülbelül feleannyi, mint a vezérnek, hisz ugyanazokat tudja lépni, kivéve átlósan. A szabály szerint 2 bástyánk van, vagy gyalogcserével beváltható 8 db, így maximum 10 bástyánk lehet. Általában 2 bástya többet ér 1 királynőnél, de van, aki a királynőt részesíti előnyben. A bástya értéke: 5. Jele: R.

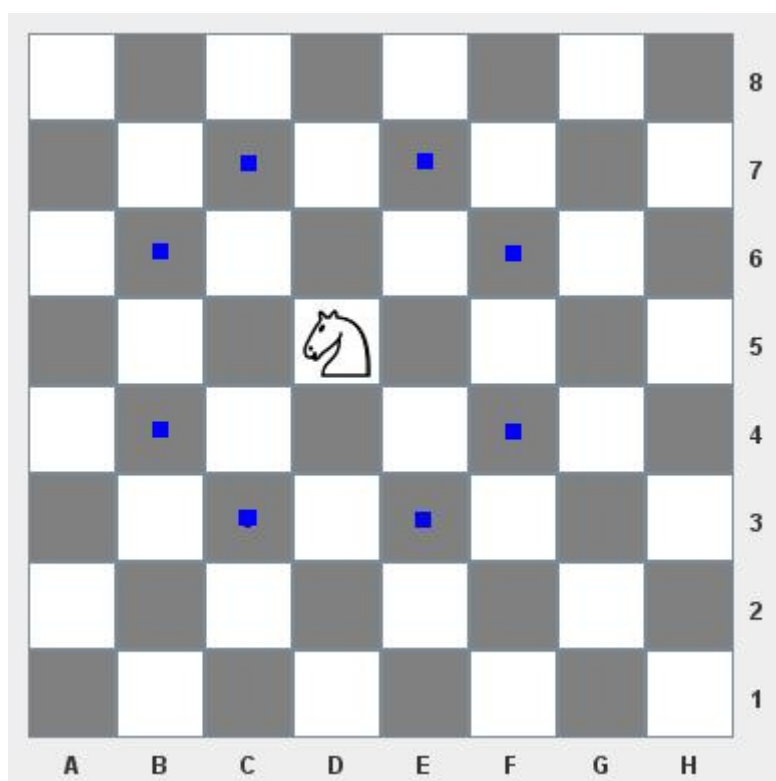


A bástya lehetséges lépései

A huszár (Knight)



A huszár úgynevezett L alakban lép, vagyis vízszintesen 2-t és függőlegesen 1-t vagy vízszintesen 1-t és függőlegesen 2-t. Nem üthet le saját bábút, viszont átugorhat ellenséges és saját bábukat is. Mivel a huszár ugrik és nem halad, mint a többi figura, figyelni kell a lépéseinél, hogy a táblán kívül ne ugorjon (programban külön gondot okoz a huszárlépések számolása). Legális lépéseinek száma maximum 8 lehet. A huszár értéke: 3. Jele: N.

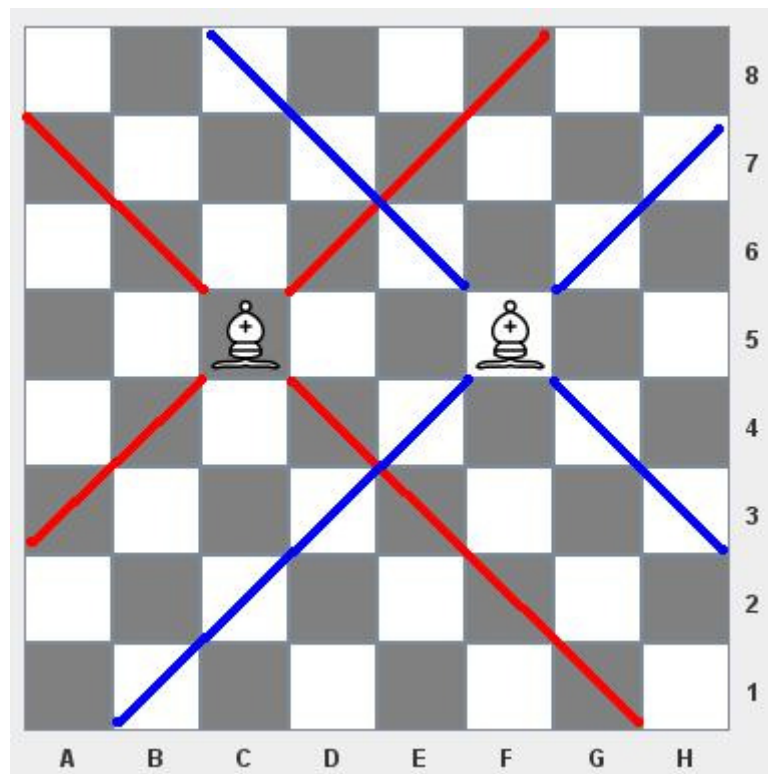


A huszár lehetséges lépései

A futó (Bishop)



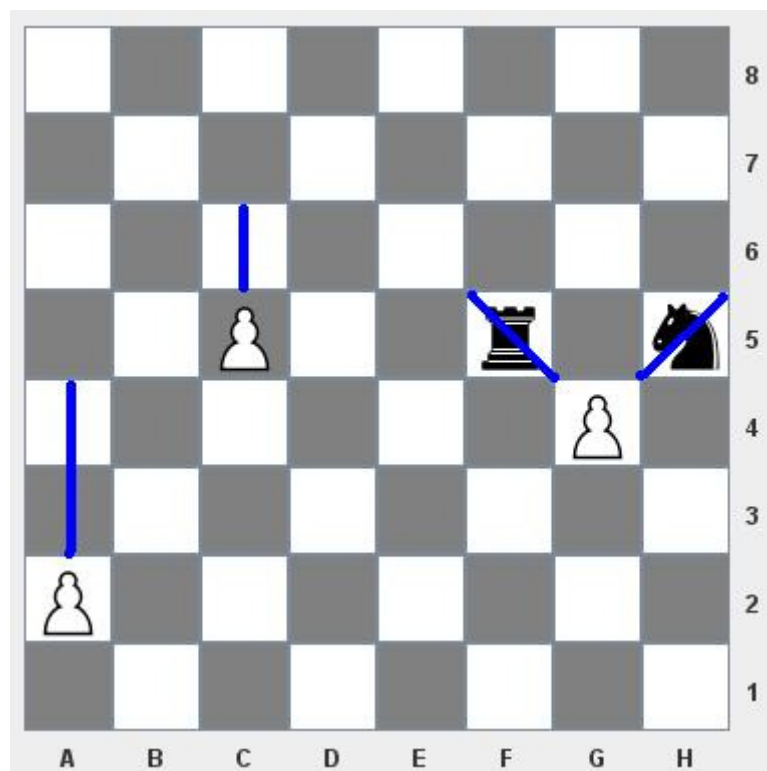
A futó csak átlósan tud lépni bármennyit addig, míg saját bábu nincs a mezőn vagy el nem érte a tábla szélét. 2 db futónk van, együtt futópárnak nevezzük őket, amik ellentétes színen állnak a táblán. Az egyik csak világos mezőn léphet, a másik csak sötét. Gyalogátváltásnál persze lehet több futónk azonos színen, maximum 10 futó lehet. Értéke megegyezik a huszáréval: 3. Jele B.



A futó lehetséges lépései

A gyalog (Pawn)

Gyalog esetén máshogyan kell ütni, mint lépni. Kizárólag csak előre léphet 1-t, vagy ha az alapvonalon áll, akkor 2-t, de nem állhat előtte más figura, átugrani nem tudja. Ütni csak átlósan tud 1 mező távolságból és csak akkor, ha ellenséges bábu van átlósan (kivéve a király). Ha eléri az irányának megfelelő utolsó sort, akkor átváltozik tisztté. 8 gyalogunk lehet maximum, ez a szám csak csökkenhet, hisz gyalogra nem lehet átváltoztatni a gyalogot. A lépésszáma a legkevesebb, általában 1 vagy 2, ennél fogva a lépéseinek számolása gyorsan megy. Speciális lépése a menet közbeni ütés (en passant) lásd speciális lépések. A gyalog értéke: 1. Jele: P vagy nem is jelölik csak a lépést.



A gyalog lehetséges lépései

3.2 Speciális lépések

A sakkban és a sakkfeladványokban is fontosak ezek a speciális lépések, így beszélnem kell még néhány lépésfajtáról, amiben 1 vagy 2 bábu vesz részt egyszerre. Ezek a speciális lépések a sáncolás, a menet közbeni ütés és még beszélhetünk a gyalogátváltoztatásról is. Programban ezeket a lépésfajtákat külön vizsgálni kell, esetleg opcionálisan megadható, hogy vizsgáljuk-e vagy nem.

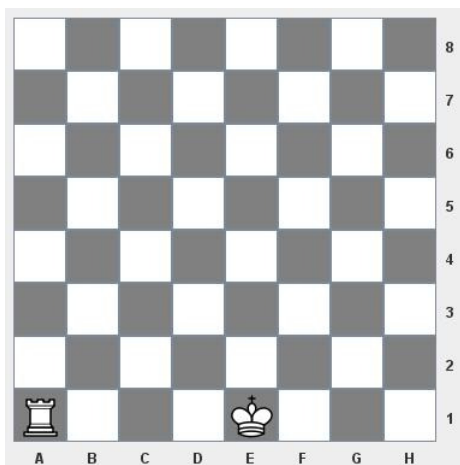
Sáncolás

A király lépései közé soroljuk, az alaplépéseitől eltérően sáncolás esetén 2-t is léphet a király az alapvonalon valamelyik bástya irányába. Beszélhetünk rövid sáncról, ha a királyszárnyra sáncolunk, vagyis a királyhoz közelebbi bástya felé, vagy hosszú sáncról, ha a vezérszárnyra (királytól távolabbi bástya felé) sáncolunk. A játék alatt csak egyszer sáncolhatunk, ki-ki maga választja, hogy melyik irányban szeretne sáncolni. Személy szerint én a rövid sáncot alkalmazom a legtöbbször. Sáncolni akkor tudunk, ha teljesülnek az alábbi feltételek:

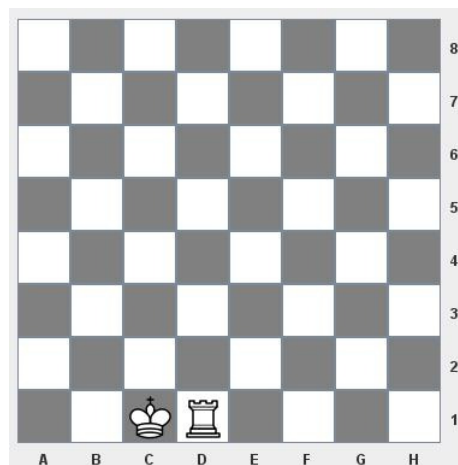
- a király nem lépett még a játék folyamán
- az a bástya, amellyel sáncolni szeretnénk, még nem lépett a játék folyamán
- a király és a bástya között nincs egyetlenegy bábu sem, sem ellenséges, sem saját
- a király nem áll sakkban
- a sáncolás alatt a király nem halad át ellenség által támadott mezőn

Menet közbeni ütés (en passant)

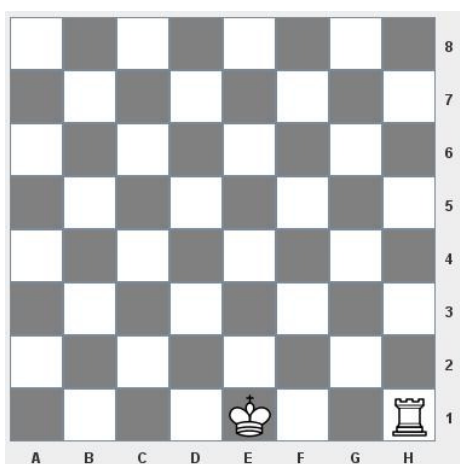
Ha a kiindulási mezőjéről kettőt lépő gyalogunkkal áthaladunk egy ellenséges gyalog ütésmezején, akkor az ellenfél gyalogunkat a következő lépésben – de csakis akkor – leütheti. Az ütés pontosan úgy történik, mintha az ütött gyalog csak egyet lépett volna. Az *en passant* lépés kiegyenlíti a különböző pozíciójú gyalogok közti erőkülönbséget. Erősebbé teszi a felezővonalon túlra tolt gyalogot, amely azonban innentől már csak egy mezőt léphet, szemben az alapvonalon álló gyalogokkal, amelyeket két mezővel is előrébb lehet tolni.



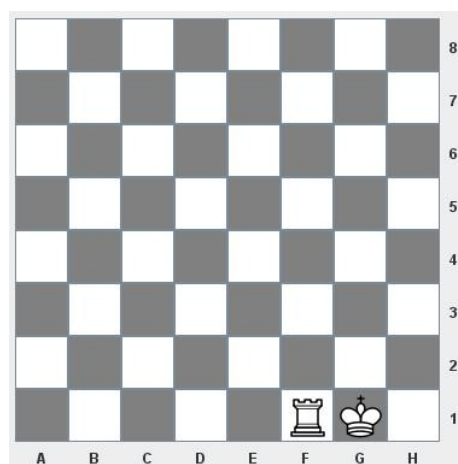
hosszú sánc előtt



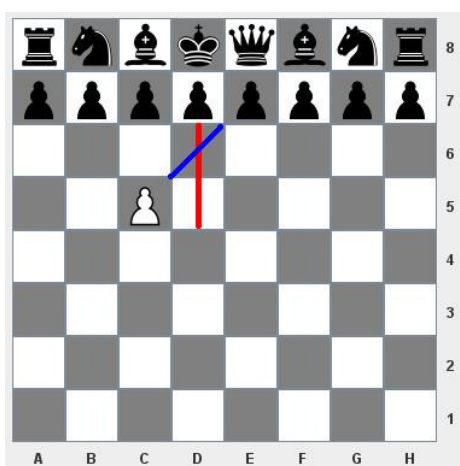
hosszú sánc után



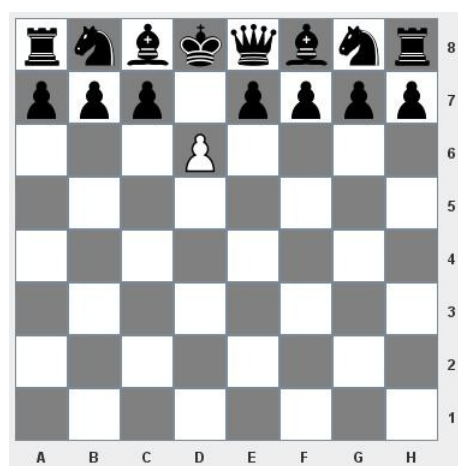
rövid sánc előtt



rövid sánc után



en passant előtt



en passant után

Gyalogátválttatás

Már volt róla szó a tanulmányoknál, hogy a gyalogátválttatás akkor lehetséges, amikor az irányának megfelelő utolsó sorba ér a gyalog. Ekkor kötelezően és azonnal átváltozik egy kiválasztott tisztté. A kiválasztott tiszt lehet vezér, bástya, huszár, futó és amennyi gyalog elérte az utolsó sort annyiszor lehet választani közülük. Szóval a játék végére lehet akár maximum 9 vezér vagy 10 bástya vagy 10 huszár vagy 10 futó. A helyzet dönti el, hogy ki mire változtatja a gyalogot, de a legtöbb esetben vezérré alakítjuk, mert ez a legerősebb figura.

4. Sakkfeladványok

„A sakkfigurák meghálálják, ha jó helyre állítjuk őket.” – Emanuel Lasker

4.1 A sakkfeladványokról

A sakkfeladvány egy rejtvény, amit a készítő a sakktáblán sakkfigurák segítségével raknak össze és valamilyen feladatot kell teljesíteni a megoldónak. Általában megadnak több információt egy sakkfeladvánnyal kapcsolatban, például melyik fél lép először, hány lépésből történik a matt adás, a feladvány típusát vagy éppen egy kulcslépést, hogy hova lépünk először. Egy jó sakkfeladvány a következő tulajdonságokkal és elemekkel rendelkezik:

- **Megszerkesztett pozíció** – egy sakkfeladvány nem egy már lezajlott sakkjátszma egy állása, hanem egy a készítő által kitalált állás. Egy már lezajlott sakkjátszma tanulságos lehet, hogy hogyan, milyen lépéssorozatokkal győzött az egyik fél a másik ellen, de ez nem sakkfeladvány. A konvencióknak megfelelően legális állásnak kell lennie, vagyis egy alapfelállásos sakkjátszmából különböző lépéssorozatokon át el lehessen jutni a sakkfeladvány állásáig. Persze nagyon sok olyan sakkfeladvány létezik, ami nem követi ezt a konvenciót, de ettől még hasonlóan jól megszerkesztett sakkfeladvány lehet.
- **Feltétel megadása** – olyan állítás, amit a megoldónak teljesíteni kell, hogy megoldja a sakkfeladványt. Például megadjuk azt a feltételt, hogy világos kezd és 3 lépésben mattot ad. Ekkor csak az a jó megoldás, amikor a 3. lépésben mattot kap a sötét, és nem a 2.-ban és nem több lépésben. Persze olyan feltétel is lehet, hogy 3 vagy kevesebb lépésben matt. Nagyon sokféle feltétel létezik.
- **Téma megadása** – a téma a legfontosabb része a feladványnak, a készítő általában valamilyen ötletet mutat be a feladványával, ez az ötlet lehet például egy stratégia az ellenfél védelmének feltörésére vagy éppen vezér áldozattal törbe csalni az ellenfelet. A témák listáját a függelék 3. táblázata tartalmazza.
- **Megoldások** – egy feladványnak egy vagy több megoldása is lehet, a megoldás lehet eltervezett vagy a tervezettől eltérő lépéssorozat által előállt megoldások. Ha a nem

tervezett megoldások száma meghaladja a tervezett megoldások számát, akkor azt mondjuk, hogy a sakkeladvány megdőlt vagy kidőlt (cooked). Ugyanis egy nagyon jól megszerkesztett sakkeladványnak 1 vagy 2 (általában kevés) tervezett megoldása van, ha ezen kívül másféle megoldás is lehetséges, akkor ez a sakkeladvány megdőlt.

- **Nehézség** – egy sakkeladvány esetén a nehézségét a legnehezebb megállapítani, ugyanis relatív kérdés. Egy tapasztalt megoldónak lehet egy feladvány könnyű, viszont egy kezdőnek lehet nagyon nehéz is. A legtöbb készítő megpróbálja viszonylag nehézre készíteni a feladványát, így a legtöbb megoldónak élvezetes fejdondolkodás nyújt. Más kérdés az, hogy az úgynevezett Megoldó versenyekre (Solving Tournament) a készítők megpróbálják a lehető legnehezebbre csinálni a sakkeladványokat.

4.2 Sakkeladványok típusai

4.2.1 Nyílt vagy közvetlen matt feladványok (directmates)

Közvetlen matt feladványok esetén világos lép először és meghatározott számú lépésben mattot ad sötétnek bármilyen védelemmel szemben. Gyakran az angolban úgy hívják ezeket, hogy „Mate in n”, ahol az n a lépésszámot jelöli, tehát mattot kell adni n lépésben. A közvetlen matt feladványoknak 3 csoportja van:

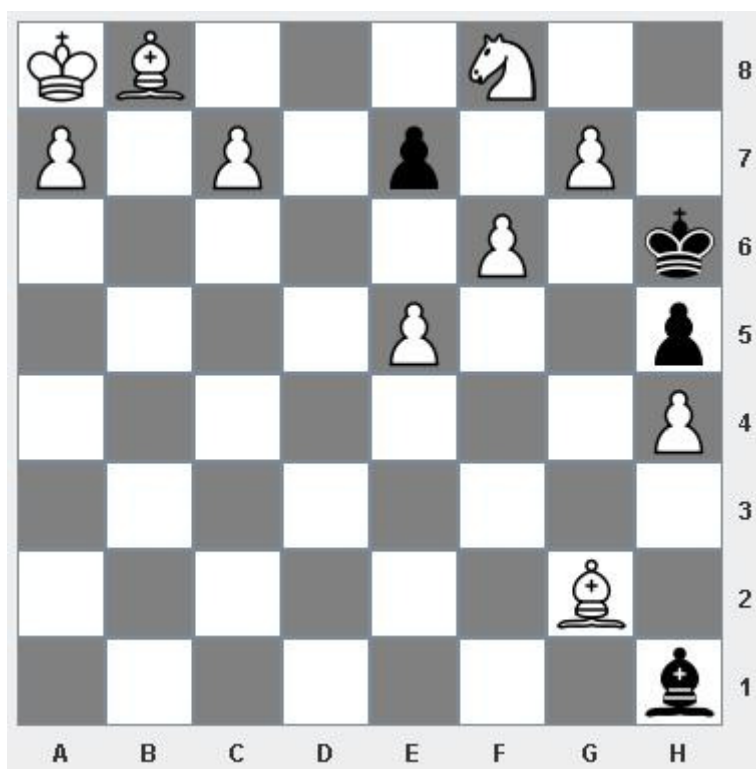
1. Kétlépéses (Two-movers)
2. Háromlépéses (Three-movers)
3. Többlépéses (More-movers)

Nagyon ritkán, de 1 lépéses feladványok is vannak, ezeket a legegyszerűbb megoldani, hisz az első és egyben utolsó lépés mattot eredményez, így ezt a feladványcsoportot nem vettem bele a csoportosításba. Értelemszerűen a kétlépéses feladványoknál világos kezd és 2 lépésen belül mattot ad sötétnek, így összesen 3 lépés történik. Háromlépéses feladványoknál 3 lépésben ad mattot világos a sötétnek, összesen 5 lépés történik. Többlépéses feladványoknál, ha n lépésben ad mattot világos sötétnek, akkor $(2 * n) - 1$ lépés történik, mivel az utolsó

mattot adó lépésnek nincs ellenlépése a sötét oldalon, mert már vége a játéknak. Lehetséges az is, hogy sötét ad mattot n lépésben világosnak, ez fordított közvetlen matt, de a lényege ugyanaz, csak a felek felcserélésével. Minél több lépéses egy feladvány, annál bonyolultabb és nehezebb megoldani és sokat számít, hogy mennyi bábu van a táblán világos és sötét oldalon is.

4.2.2 Önmatt feladványok (selfmates)

Az önmatt olyan sakkeladvány, melyben a világosnak megadott számú lépésben rá kell kényszerítenie sötétet a matt adásra. Tehát megadott számú lépésben önmagunkat mattoljuk be úgy, hogy sötétnek nem kell mattot adnia, ha van más lépési lehetősége. Egy példán keresztül bemutatva tisztán fog látszani egy önmatt feladvány sajátosságai. Vegyük az alábbi sakkeladványt, amelyet Wolfgang Pauly 1912-es „The Theory of Pawn Promotion” könyvében publikált.



Wolfgang Pauly (1912), világos kezd és önmatt 2 lépésben

A feladványt szemügyre véve látszik, hogy 2 lépésben csakis a H1 sötét futó adhat mattot világosnak. Így ha sötétnek semmilyen más lehetősége nincs, akkor BxG2 lépéssel leüti világos futót és önmatt.

Világos a következőket teheti:

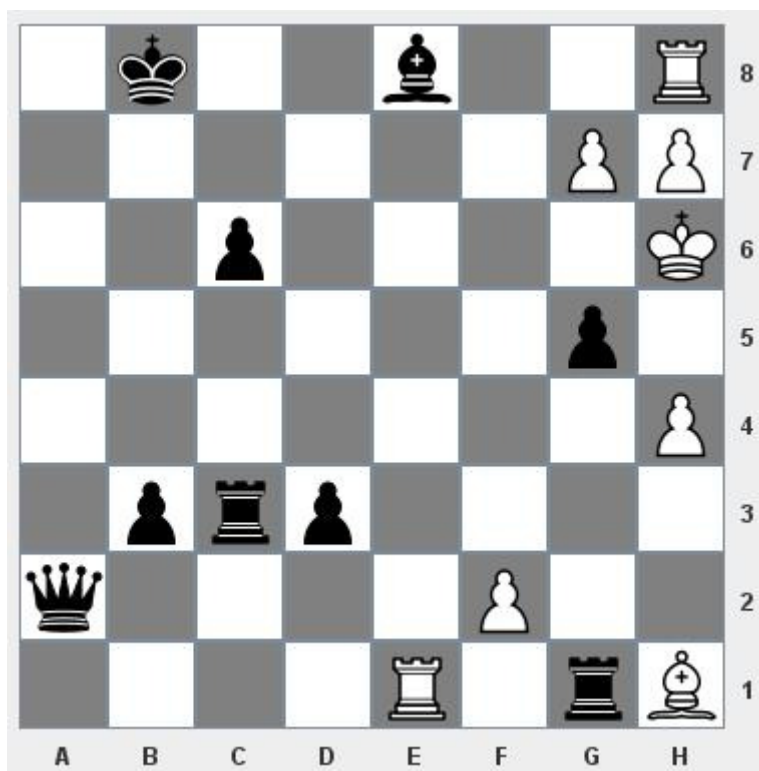
- Ha a G2 futót mozgatja, akkor megnyitja a H1 futó vonalát, ezáltal lépési lehetőséghez juttatva sötétet, így a 2. lépésben nem lesz matt.
- Ha az F8 huszárral lép, akkor a sötét királyt lépésekhez juttatja (G6-H7)
- Ha a G7-ről G8-ra lép és gyalogátváltásnál vezért vagy bástyát választ, akkor G2 futónak védelmet biztosítanak.
- Ha G8 gyalogátváltásnál huszárt választ, akkor a sötétnek matt, így nincs önmatt.
- Ha G8 gyalogátváltásnál futót választ, akkor az E7 sötét gyalog leüti F6 gyalogot, mert más lépése nincs, E5-ről világos gyalog leüti F6 sötét gyalogot, már csak a sötét futó G2-re lépése marad, de ekkor a választott világos futó tud védekezni a D5 mezőn.
- Ha az E5 gyaloggal indulunk meg E6-ra, akkor az E7 sötét gyalog előtt megnyílik az út és lépési lehetőségekhez jut.
- Ha F6 világos gyalog leüti E7 sötét gyalogot, akkor a sötét király lépési lehetősége nő (KxG7), így 2. lépésben nem lehetséges rákényszeríteni az önmattra.

A megoldás:

C8 gyalogátváltás huszárra, semmilyen más bábú nem jó, mert azok védekezni tudnának a sötét futó ellen. Ekkor sötét gyalog üt, világos visszaüt és BxG2 lépéssel sötét futó mattot ad. Másik lehetőség, ha a sötét gyalog nem üt, hanem egyet előre lép, ekkor G8 gyalogátváltás futóra a lépés, ezután BxG2 matt.

4.2.3 Reflexmatt feladványok (reflexmates)

Az önmatt feladványok egy olyan fajtája, melyben a világosnak ugyanúgy matt adásra kell kényszerítenie a sötétet, de kiegészül egy olyan szabállyal, hogy ha bármely fél mattot tud adni az adott lépésben, akkor meg kell lépnie azt. Ha ez a szabály csak a sötét félre vonatkozik, akkor egy fél-reflexmattról beszélünk. Császi Károly kétlépéses reflexmatt feladványának megoldásait mutatom be példaként.



Császi Károly, sötét kezd és reflexmatt 2 lépésben

Az összes lehetőség végignézésére is lehetőség van, de elég terjedelmes lenne leírni, hogy miért melyik bábuval hova nem jó lépni. Ezért a megoldásokkal mutatom meg, hogy milyen helyzetben kötelező mattot adni a másik félnek. Első megoldás, ha G1 sötét bástya leüti H1 futót, akkor H4 gyalog H5-re lépésével sötétnek kötelező mattot adni H1 sötét bástya H5 ütésével. Második megoldás, ha C6 gyalog C5-re lép, akkor megnyílik a 6. vonal, ahonnan mattot kaphat a világos király, ekkor világosnak egy olyan lépést kell tennie, ami nem akadályozza meg sötétet a matt adástól. Például egy F2 gyalog F3-ra lépése nem zavarja meg

világost, így sötétnek A2 vezérrel mattot kell adnia az A6-ra lépéssel. Egy harmadik megoldás, ha G5 sötét gyalog G4-re lép, akkor megnyílik a C1-A6 átló, így ha világos nem lép ellene, akkor sötétnek mattot kell majd adnia. Ha H1 világos futó F3-ra lép, akkor A2 sötét vezér a következő lépésben D2-re lépve mattot adott.

4.2.4 Segítőmatt feladványok (helpmates)

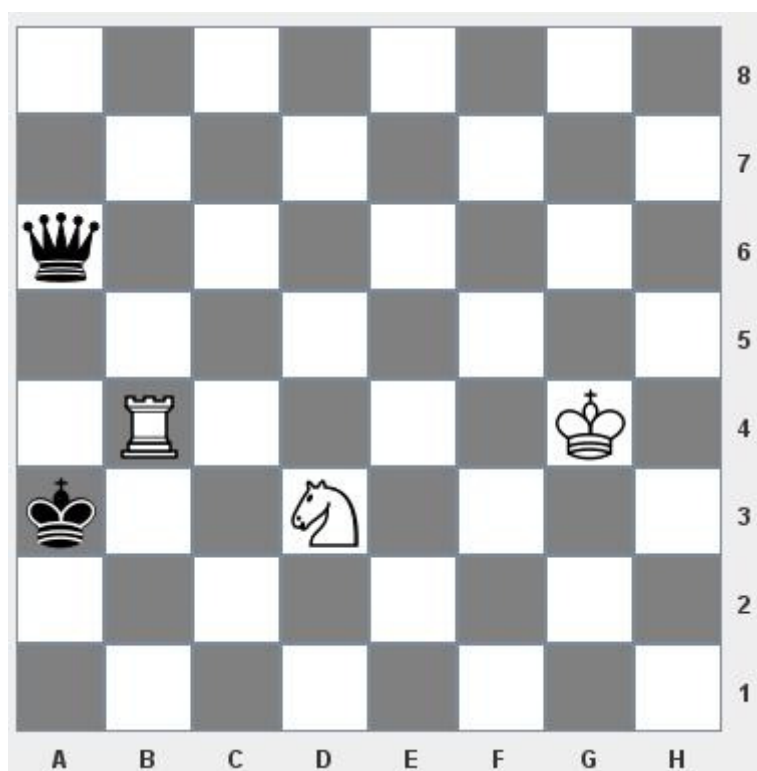
A segítőmatt feladványok, olyan feladványok, amelyekben mindkét fél közreműködik a sötét fél bemattolásában. Egy n lépéses segítőmatt esetén ténylegesen $2n$ lépés van összesen, nem úgy, mint a közvetlen mattok esetén. Sötét kezd és minden sötét lépéshez tartozik egy világos lépés, ezt ismételjük mindaddig, míg a legutolsó világos lépés mattot ad sötétnek. Persze igaz fordítva is, ha világos kezd, akkor minden lépéshez tartozik egy sötét válaszlépés, míg a legutolsó sötét lépés mattot eredményez világosnak. A segítőmatt feladványok könnyebbnek tűnnek a közvetlen feladványoknál, hisz ugyanazon fél bemattolása a cél, a közvetleneknél a másik fél a matt elkerülésére törekszik.

A segítőmattnak három fajtája van:

1. normál segítőmatt, esetleg több megoldással
2. ikermatt (twinning)
3. kettős vagy kétirányú matt (duplex)

Ikermatt

Ugyanazon állásból több segítőmatt konstruálható kisebb változtatásokkal pl. egy bábut egyik sarokból a másikba rakunk, bábut rakunk fel a táblára vagy éppen elveszünk, elforgatjuk a táblát. Bemutatok egy ikermatt példát, ahol a táblán lévő sötét vezérrel érünk el segítőmattot, majd az összes többifajta sötét bábuval (bástya, huszár, futó, gyalog) is ugyanannyi lépésből, amiket ugyanoda állítunk fel, mint a sötét vezért.



ikermatt példa

Mindegyik kicserélt bábu pozíciója A6.

Vezérrel:

1. Sötét vezér F6-ra 2. Világos huszár C5-re 3. Sötét vezér B2-re 4. Világos bástya A4-re

Bástyával:

1. Sötét bástya B6-ra 2. Világos bástya B1-re 3. Sötét bástya B3-ra 4. Világos bástya A1-re

Huszárral:

1. Sötét huszár C5-re 2. Világos huszár C1-re 3. Sötét huszár A4-re 4. Világos Bástya B3-ra

Futóval:

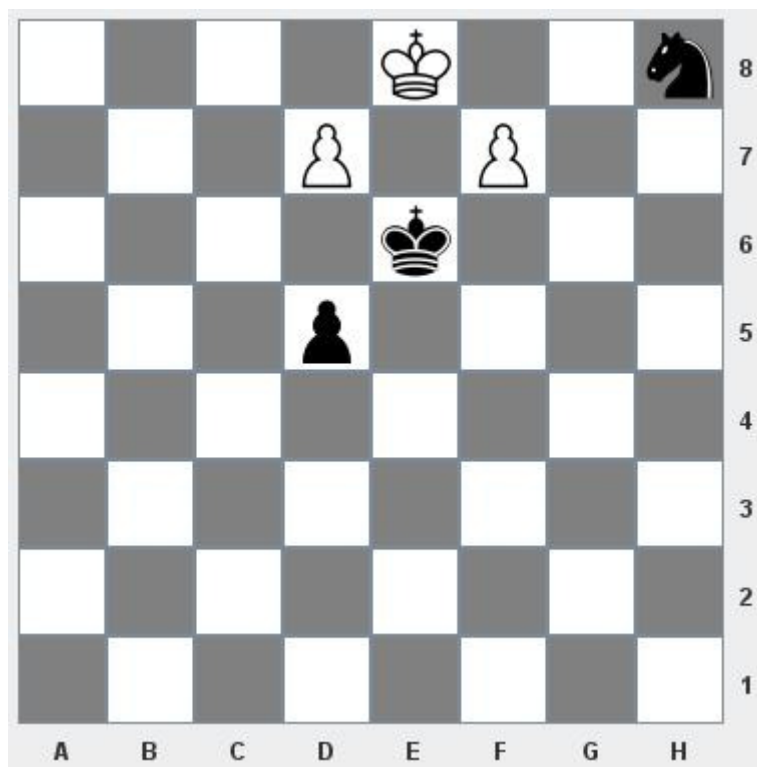
1. Sötét futó C4-re 2. Világos huszár E1-re 3. Sötét futó A2-re 4. Világos huszár C2-re

Gyaloggal:

1. Sötét gyalog A5-re 2. Világos bástya B3-ra 3. Sötét király A4-re 4. Világos huszár C5-re

Duplexek

Duplexek esetén 2 probléma van egy feladványban, az első egy normál segítőmatt, a másodikban pedig világos kezd és sötét segít mattot adni világosnak, vagyis egy fordított segítőmatt. Az alábbi példa egy kétlépéses duplex feladvány.



duplex példa

A feladvány a 2 világos gyalog átváltására épít és mind a négy átváltás különböző bábura történik. Az első probléma megoldása:

Sötét huszár G6-ra, ezután világos F7 gyalogja F8-ra és vezérre csere. A vezér lezárja az F1-F8 vonalat a sötét király számára. Ezután G6 sötét huszár E5-re, ezzel segítve világosnak, hogy a sötét király lépésszámát korlátozza. Második gyalogátváltás D7 gyalog D8-ra és egy világos huszár cserével matt. A második megoldásnál elsőnek F7 világos gyalog F8-ra és bástyára csere. Második lépésként H8 sötét huszár F7-re, ezzel lezárva a jobb oldalt a világos király előtt. Harmadik lépés D7 gyalog D8-ra és futóra csere. Negyedik lépés F7 huszár D6-ra és matt.

4.2.5 Sorozatlépéses feladványok (seriesmovers)

Sorozatlépéses feladványok esetén az egyik fél lépések sorozatát lépi meg (egymásután lép n -szer, anélkül, hogy a másik fél lépne válaszlépést). Az utolsó lépése után a másik fél lép egyet, amivel mattot ad vagy pattot harcol ki, attól függően, hogy mik a feltételek. A sorozatlépés alatt nem adható sakk a másik félnek, csakis a legutolsó lépésben. A sorozatlépéses feladványoknak vannak megfelelőik az eddig tárgyalt feladványok analógiájára:

- Sorozatlépéses közvetlen matt – lépések sorozatával világos mattot ad sötétnek az utolsó lépésben.
- Sorozatlépéses segítőmatt – sötét lépéssorozata után világos mattot ad sötétnek 1 lépéssel.
- Sorozatlépéses önmatt – világos lépéssorozataival olyan pozícióba vezeti az állást, hogy sötét mattot kényszerül adni világosnak.
- Sorozatlépéses reflexmatt – világos lépéssorozataival olyan pozícióba vezeti az állást, hogy sötétnek mattot kell adnia világosnak, ugyanakkor világosnak el kell kerülni azokat a lépéseket, amikkel mattot adhat sötétnek.

4.2.6 Tündérfeladványok (feary problems)

Ezek a feladványok különböznek a klasszikus értelemben vett sakkfeladványoktól. Azok kiegészítésének, továbbfejlesztésének vagy átformálásának tekinthetők. Új feltételek jelentek meg ezekben a feladványokban, például egy kezdőállásból induló állapotig kell eljutni megadott számú lépésben. Ezeket legrövidebb bizonyított játékoknak nevezik (shortest proof games). Ilyen új feltétel még a hátrafelé irányuló analízis (retrograde analysis), ami egy jelen állástól visszafelé nézi, hogy vajon milyen lépéseket tettek, ami elvezetett a jelen állásig. Az alap sakkbábuk sokféle irányban fejlődtek, újabb bábuk jöttek létre. Ilyen új bábu például a huszár és a futó ötvözete (Archbishop), a huszár és a bástya ötvözete (Chancellor), fejjel lefelé fordított vezér (Grasshopper), fejjel lefelé fordított huszár (Nightrider) és még nagyon sok más bábu. Új szabályok is vannak, másak a sakkra vonatkozó szabályok, általános lépések néhol máshogy vannak vagy akár a matt nem ugyanolyan feltételek mellett teljesül. Különböző táblák is használatosak, nem csak a 8x8-as tábla, hanem van 7x8-as is vagy akár teljesen más alakú táblák is vannak.

A tündérfeladványok sokszínűségét szakdolgozatomban nem tárgyalom, megmaradok a hagyományos, klasszikus értelemben vett sakkfeladványoknál.

5. Sakk és a mesterséges intelligencia

„Egyre jobbak a gépek, azonban mi emberek is képesek vagyunk tanulni.” -

Garri Kaszparov

5.1 Mesterséges intelligencia alapfogalmak

A sakkprogramok a mesterséges intelligenciából ismert fogalmakat, algoritmusokat használják, így elengedhetetlen ezek ismerete a hamarosan tárgyalásra kerülő algoritmusokhoz és a saját fejlesztésű programomhoz.

5.1.1 Állapottér-reprezentáció

Egy adott „p” probléma esetén megkeressük e probléma fontosnak vélt meghatározóit, melyeknek száma m. Ezeknek a meghatározó elemeknek vannak értékei, tehát ha rendre

$h_1, h_2, \dots, h_m$ ezek az érték m-esek, akkor a „p” probléma egy állapotát írják le. Ezeknek az állapotoknak a halmaza a „p” probléma állapottere. Az i. jellemző által felvehető értékek halmaza H_i . Ugye m db ilyen jellemzőnk van, tehát „p” állapotai elemei lesznek a $H_1 \times \dots \times H_m$ halmaznak. Ennek a halmaznak az úgynevezett kényszerfeltételekkel szűkített részhalmaza lesznek a probléma valódi állapotai:

$$A = \{ a \mid a \in H_1 \times \dots \times H_m \text{ és kényszerfeltétel}(a) \}$$

Azt az állapotot, amelynek kezdőértékei határozzák meg a probléma világát **kezdőállapotnak** nevezzük. Az állapotokat megváltoztatni *operátorokkal* tudjuk. Az operátorok állapotváltozást leíró leképezések. Nem minden operátor alkalmazható feltétlenül minden állapotra, meg kell adni egy operátor értelmezési tartományát, amit operátoralkalmazási előfeltételekkel tudunk:

$$Dom(o) = \{ a \mid a \in A \text{ és } o\text{-alkalmazásának-előfeltétele}(a) \}$$

Operátorok alkalmazásával különféle állapotok állnak elő, ezek közül a számunkra megfelelőeket **célállapotoknak** nevezzük. Célállapotokat megadhatunk felsorolással vagy célfeltételek segítségével:

$$1. C = \{ c_1, \dots, c_l \mid c_i \in A, i = 1, \dots, l, l \geq 1 \}$$

$$2. C = \{ c \mid c \in A \text{ és } \text{célfeltételek}(c) \}$$

$C \subset A$ és a kezdőállapot $\notin C$, mert akkor nem lenne megoldandó feladat.

Szóval az állapottér-reprezentációt a következő négyessel adhatjuk meg:

$\langle A, \text{kezdő}, C, O \rangle$ - ahol A az állapottér nem üres halmaza, kezdő a kezdőállapot, ami az állapottér eleme, C a célállapotok halmaza és az állapottér részhalmaza, O az operátorok véges, nem üres halmaza.

5.1.2 Közvetlen elérhetőség, elérhetőség, megoldás, költség

Az $a \in A$ állapotból közvetlenül elérhető egy $a' \in A$ állapot, ha van olyan $o \in O$ operátor, melyre o -alkalmazásának előfeltétele(a) és $o(a) = a'$. Jele: $a \Rightarrow a'$.

Az $a \in A$ állapotból az $a' \in A$ állapot elérhető, ha vagy $a = a'$, vagy van olyan $a_1, \dots, a_k \in A$ ($k \geq 2$) véges állapotsorozat, hogy $a = a_1$, $a' = a_k$ és $a_i \Rightarrow a_{i+1}$ minden $1 \leq i \leq k - 1$ esetén.

Jele: $a \Rightarrow^* a'$.

Legyen $p = \langle A, \text{kezdő}, C, O \rangle$. A „ p ” probléma megoldható ebben az állapottér-reprezentációban, ha van olyan $c \in C$ célállapot, hogy $\text{kezdő} \Rightarrow^* c$. Ha a c -t egy $o_1, \dots, o_r$ operátorsorozat alkalmazásával értük el ($r \geq 1$), akkor az $o_1, \dots, o_r$ operátorsorozat a probléma egy megoldása.

Jelölje a *költség* (o, a) az „ o ” operátor „ a ” állapotra alkalmazott költségét. Legyen $p = \langle A, \text{kezdő}, C, O \rangle$ probléma és $o_1, \dots, o_r \in O$ egy megoldása a problémának, azaz $\text{kezdő} \Rightarrow^* c$, ahol $c \in C$ és az $o_1, \dots, o_r$ operátoralkalmazásokkal értük el. Legyen rendre $a_i \Rightarrow a_{i+1}$, amit az o_i operátorral állítottunk elő ($1 \leq i \leq r$), ahol $a_1 = \text{kezdő}$ és $a_{r+1} = c$. Ekkor a megoldás költsége $\sum_{i=1 \dots r} \text{költség}(o_i, a_i)$, vagyis a megoldásig alkalmazott operátorok költségeinek összege.

Ha a *költség* (o, a) = 1, akkora megoldás költsége az alkalmazott operátorok száma.

5.1.3 Állapottérgráf

Ha a „p” problémát megadtuk az $\langle A, kezdő, C, O \rangle$ négyessel, akkor ez a reprezentáció egy irányított gráfot határoz meg.

A gráf csúcsai az A állapottér elemei, jelöljük n_a -val az $a \in A$ állapot által definiált csúcsot. A gráf csúcsainak a halmaza: $N = \{ n_a \mid a \in A \}$.

A kezdőállapotnak a startcsúcs felel meg, jele: s . A célállapotok megfelelője a terminális csúcsok: $T = \{ n_c \mid c \in C \}$.

Minden n_a csúcsból irányított élt húzunk egy $n_{a'}$ csúcsba, ha n_a –ból közvetlenül elérhető $n_{a'}$.

Ezen irányított élek felelnek meg az operátoroknak: $E = \{ (n_a, n_{a'}) \mid a, a' \in A \text{ és } a \Rightarrow a' \}$.

Az így kapott $\langle N, s, T, E \rangle$ négyest a „p” probléma állapottérgrádjának nevezzük.

5.2 Megoldást kereső rendszerek

Sakkprogramozásban nincs igazán jelentőségük a megoldást kereső rendszereknek, mivel maga a sakk egy kétszemélyes stratégiai játék, ahol lépéseket kell ajánlani a megadott félnek. Viszont egy sakktárgy esetén determinisztikus, hogy megadott lépésben matt lesz, így ezek a keresők találnak megoldást, ha létezik megoldás. Ha nem létezik megoldás, azt véges sok lépésben felismerik.

Ezek a rendszerek az állapottérgráfban keresnek a start csúcsból induló terminális csúcsba vezető utat, vagyis megoldást. Az állapottérgráfot implicit módon megadjuk a rendszernek és a keresés során addig és úgy építik fel a gráfot, míg megoldást nem találnak vagy valamilyen oknál fogva sikertelen a keresés. A megoldást kereső rendszerek felépítése 3 részből áll:

1. **adatbázis** – az állapottérgráf a keresés során előállított része, amit kiegészíthetünk a hatékony kereséshez szükséges információkkal (költség, heurisztika)
2. **műveletek** – az adatbázist módosítják, vagyis az állapottérgráf adatbázisbeli részéből egy újabb részét állítják elő

3. **vezérlő** – a keresést irányítja, a vezérlő mondja meg, hogy a megoldáskeresés folyamán az adatbázisra, annak mely részére, mely műveletek és mikor hajtódjanak végre. Figyeli, hogy befejeződhet-e a keresés, talált-e megoldást vagy nem.

```
1: procedure KERESŐ( $\langle \mathcal{A}, kezdő, \mathcal{C}, \mathcal{O} \rangle$ )
2:    $adatbázis \leftarrow$  INICIALIZÁL( $kezdő$ )
3:   while IGAZ do
4:     if MEGOLDÁS-TALÁL( $adatbázis$ ) then
5:       break
6:     end if
7:     if NEM-FOLYTAT( $adatbázis$ ) then
8:       break
9:     end if
10:     $művelet \leftarrow$  VÁLASZT( $adatbázis, műveletek$ )
11:     $adatbázis \leftarrow$  ALKALMAZ( $adatbázis, művelet$ )
12:  end while
13:  if MEGOLDÁS-TALÁL( $adatbázis$ ) then
14:    MEGOLDÁS-KÍR( $adatbázis$ )
15:  else
16:    print „Sikertelen keresés”
17:  end if
18: end procedure
```

kereső rendszer pszeudo algoritmus⁴

Többféle módon lehet osztályozni a megoldást kereső rendszereket:

1. Az alkalmazott művelet hatása módosítható-e
 - a. Nem módosítható keresők: próba-hiba módszer, hegymászó módszer

⁴ Várterész Magdolna – Mesterséges intelligencia 1 előadás jegyzet

b. Módosítható keresők:

- i. Visszalépéses (backtrack) keresők: alap backtrack, körfigyeléses backtrack, úthosszkorlátos backtrack, ág és korlát algoritmus
- ii. Keresőgráffal keresők: szélességi kereső, mélységi kereső, optimális kereső, best-first algoritmus, A algoritmus, A* algoritmus, monoton A algoritmus

2. A keresés során valamilyen információ alapján keresünk-e

- a. Nem: irányítatlan keresők: backtrack keresők, szélességi - és mélységi kereső
- b. Igen: heurisztikus keresők: optimális kereső, best-first algoritmus, A algoritmusok

Lehet beszélni a keresés irányáról:

- Előrehaladó, adatvezérelt – a kezdőállapotból keresünk célállapotokat és az oda vezető utat.
- Visszafelé haladó, célvezérelt – a célállapotból kiindulva próbáljuk meg megtalálni a kezdőállapotot és a kezdőállapotból a célállapotba vezető utat.
- Kétirányú – mindkét irányban keresünk és valahol a keresés során találkoznak.

A megoldáskereső rendszerek értékelési szempontjai:

1. teljesség – ha létezik megoldás, megtalálja-e a kereső azt?
2. optimalitás – a megoldások közül vajon az optimális megoldást találja meg a kereső?
3. időigény – mennyi ideig tart egy megoldás esetleg megadott számú vagy az összes megoldás megtalálása
4. tárigény – megoldás megtalálásához mennyit tárterületre van szükség?

Az algoritmusok mindegyikét nem részletezem csak a backtrack keresőt és a mélységi keresőt, amit a saját fejlesztésű programomban is használok. Sok más féle algoritmust is lehetne használni, de ezt a két algoritmust ragadtam ki a sok közül egyéni döntés alapján.

5.2.1 Backtrack keresők

Az alap backtrack kereső adatbázisa az úgynevezett aktuális utat tartalmazza, ami a start csúcsból indul és az aktuális csúcsban ér véget. Az út csúcsait és a csúccsal kapcsolatban lévő éleket tartalmazó csomópontokból épül fel. A csomópontok a következő információkat tartalmazzák:

$$\begin{aligned}\text{ÁLLAPOT}[\text{aktuális-csomópont}] &\leftarrow \text{kezdő} \\ \text{SZÜLŐ}[\text{aktuális-csomópont}] &\leftarrow \text{NIL} \\ \text{OPERÁTOR}[\text{aktuális-csomópont}] &\leftarrow * \\ \text{KIPRÓBÁLT}[\text{aktuális-csomópont}] &\leftarrow \emptyset\end{aligned}$$

alap backtrack kezdő csomópont

Az **ÁLLAPOT** $\in A$ állapot a csomópont kezdőállapota lesz, a **SZÜLŐ** egy mutató a szülő állapotra, amelyre alkalmazva valamely operátort előállt az **ÁLLAPOT**. Kezdetben nincs szülő állapot, ezért kezdeti értéke nincs. Az **OPERÁTOR** az az operátor, melyet alkalmazva a **SZÜLŐ** állapotra előállt az **ÁLLAPOT**. A **KIPRÓBÁLT** egy halmaz, mely azokat az operátorokat tartalmazza, amelyeket már alkalmaztunk az **ÁLLAPOT**-ra.

Műveletei az operátorokból származtatott műveletek. Alkalmazási előfeltétele: az aktuális csomópont állapotára alkalmazható az operátor, de az adott úton erre az állapotra még nem alkalmaztuk az operátort. Másik művelet a visszalépés, melynek előfeltétele, hogy legyen aktuális csomópont az aktuális úton.

Vezérlője eldönti, hogy mikor melyik műveletet kell végrehajtani az adatbázisra, ha még nem teljesülnek a megállási feltételek.

Az alap backtrack keresőt kiegészíthejük plusz információval, ami segíthet elkerülni bizonyos nem kívánatos eseteket pl. végtelen ciklus vagy éppen a backtrack kereső is használható legoptimálisabb út megtalálására. Ilyen kiegészítő információ lehet például **körök figyelése**, ami azt jelenti, hogy ha az aktuális csúcs már szerepelt az aktuális úton, akkor a vezérlő a visszalépést választja, ezzel elkerülve egy végtelen ciklust. További hasznos információ lehet **úthosszkorlát** megadása, amely szintén elkerüli a végtelen ciklusba futást. Ha az aktuális út hossza eléri vagy meghaladja az úthosszkorlátot, akkor a visszalépést választja a vezérlő.

Lehetőség van optimális, vagyis legrövidebb út keresésére is az ág és korlát algoritmussal.

Az ág és korlát algoritmus megtalálja az optimális megoldást, ha:

- egy induló úthosszkorlátnál rövidebb megoldást keresünk.
- ha van ilyen megoldás, akkor ennek az útnak a hosszát választjuk új úthosszkorlátnak és folytatjuk a keresést, míg véget nem ér.

Sakkfeladványoknál ott lehet jelentősége, amikor ugye megadtunk egy feltételt, hogy 3 lépésben matt, viszont a kereső talál olyan megoldást is ugyanennél a feladványnál, hogy 2 lépésben is matt adható. Ekkor az ág és korlát algoritmus a rövidebb megoldást választja. Viszont ilyen esetben célszerűbb az összes lehetséges megoldást keresni, hogy tudjunk válogatni a megoldások közül, feltéve, ha létezik több megoldás. Úthosszkorlátos backtrack kereső esetén pontosan olyan hosszú utakat találunk, mint ahány lépésből mattot kell adni, vagyis az út hossza n lépéses feladvány esetén, ha az aktuális csomópont állapota célállapot: $2n - 1$. Az alap backtrack kereső egy sakkfeladvány esetén minden valószínűséggel végtelen ciklusba futna, mivel a keresés irányítatlan, nincs körfigyelés és úthosszkorlát sincs megadva. Mivel minden lépés után újra meg kell adni a használható operátorokat, ezért egyes operátorok alkalmazhatóak egy másik állapotnál is.

A programomban használt úthosszkorlátos, körfigyeléses backtrack keresőt a következő fejezetben ismertetem JAVA nyelven megvalósítva.

5.2.2 Keresőfával kereső rendszerek

Keresőfával kereső rendszerek esetén az adatbázis az állapottérgráf már bejárt részét feszítő fa, a keresőfa. A csomópontok, melyek a keresőfa csúcsait és a velük kapcsolatban lévő éleket tartják nyilván, a következő információkat tartalmazzák:

- $a \in A$ állapot
- mutató, amely a szülő csomópont állapotára mutat
- operátor, melyet alkalmaztunk a szülő csomópont állapotára és előállt „a”.
- státusz, amely lehet zárt vagy nyílt. Zárt státusz esetén az „a” állapot összes utódját a keresés során már előállítottuk, egyéb esetben nyílt státuszú egy állapot

$$\begin{aligned}\text{ÁLLAPOT}[\text{csomópont}] &\leftarrow \text{kezdő} \\ \text{SZÜLŐ}[\text{csomópont}] &\leftarrow \text{NIL} \\ \text{OPERÁTOR}[\text{csomópont}] &\leftarrow * \\ \text{nyíltak} &\leftarrow \{\text{csomópont}\}; \text{zártak} \leftarrow \emptyset\end{aligned}$$

keresőfával kereső rendszer kezdő csomópontja

Műveletük a **kiterjesztés**. Kibővíti a keresőfát egy nyílt csomópontján keresztül. Előfeltétele, hogy legyen nyílt csomópont, a start csúcs kezdetben nyílt státuszú. A kiterjesztendő nyílt csomópont állapotára alkalmazzuk az összes alkalmazható operátort, az előálló állapotok közül, amelyek még nem szerepeltek egyetlen csomópontban sem, nyílt csúcsokat hozunk létre és felfűzzük a keresőfára, a kiterjesztett csomópont utódaiként. A már szerepelt állapotok sorsa keresőfüggő. Ha kiterjesztettünk egy nyílt csomópontot, akkor zárttá válik.

A vezérlő azt mondja meg, hogy melyik nyílt csúcs legyen a következő lépésben kiterjesztve. Ha a kiválasztott nyílt csomópont teljesíti a célfeltételeket, a keresőfa start csúcsától vezető kiválasztott csomópontig tartó út egy megoldás, amit a csomópontokban lévő szülő csomópontokra mutató mutatókból elő tud állítani a kereső.

A különböző keresőfával kereső rendszerek más-más módon kereshetnek megoldást. Eltérés lehet a kiterjesztésre kiválasztott csomópont megválasztásakor. A vezérlő választhat:

- irányítatlanul
 - a csomópont keresőfabeli mélysége alapján: szélességi és mélységi kereső
 - a csomópontok állapotait előállító költség alapján: optimális kereső
- heurisztikusan
 - best-first algoritmus
 - A algoritmusok

Eltérés lehet olyan esetben is, amikor egy csomóponthoz érve újabb odavezető utat talál a kereső más hosszúsággal. A keresőtől függ, hogy melyik utat választja. Általában a rövidebb út az optimális, ha ezen kívül plusz információ nincs megadva. Ha meg van adva még egy olyan információ, ami az operátorok költségére vonatkozik, akkor az út hossza és az élek költsége dönti el a választást.

Mélységi és szélességi kereső

Egy csomópont előállításakor, nyilvántartjuk, hogy a keresőfában milyen mélyen van. Egy előállt csomópont mélysége a szülő mélységszámától 1-el nagyobb lesz, tehát:

Ha $m = s$, akkor $mélység(m) = 0$ egyébként $(n, m) \in E$ esetén $mélység(n) + 1$. Az n, m csomópontok.

A szélességi kereső kiterjesztésre a legkisebb mélységi számú csomópontot választja, a mélységi kereső a legnagyobb mélységi számú csomópontot.

Ezen algoritmusok közül, a mélységi keresőt mutatom be a programomban. Mivel megadott mélységig kell vizsgálnom a sakkeladványokat, célszerűbb volt ezt választani a szélességivel szemben. Az optimális kereső csak költség alapján terjeszti ki az összes csomópontot, a best-first csak heurisztika alapján, az A algoritmus pedig költség és heurisztika alapján. Én a programomban az összes megoldás keresésére törekedtem, legyen az bármilyen lépés függetlenül attól, hogy milyen sorrendben vannak az egyes megoldások költség és heurisztika alapján, amit a mélységi kereső figyelmen kívül hagy. A keresés ideje valamivel kevesebb mélységi kereső használatakor összes megoldás esetén, hiszen a csomópontok kiterjesztésre való kiválasztása külön vizsgálat. A különbség egy megoldás esetén viszont élesen látszik, mert nincs garancia rá, hogy optimális a megoldás vagy éppen a legkisebb költségű és heurisztikájú megoldást adja a mélységi kereső.

5.3 Lépésajánló algoritmusok

Kétszemélyes stratégiai játékoknál, így a sakknál is szinte csak lépésajánló algoritmusokat alkalmaznak, ennél fogva számtalan sakkprogram van, ami használja. Egy lépésajánló algoritmus a támogatott játékosnak ajánl egy lépést, hogy mit lépjen meg. Az, hogy egy állapot mennyire jó a támogatott játékosnak, a heurisztika segítségével tudjuk eldönteni. Sakk esetében tökéletes heurisztika nem létezik, csak elég jó.

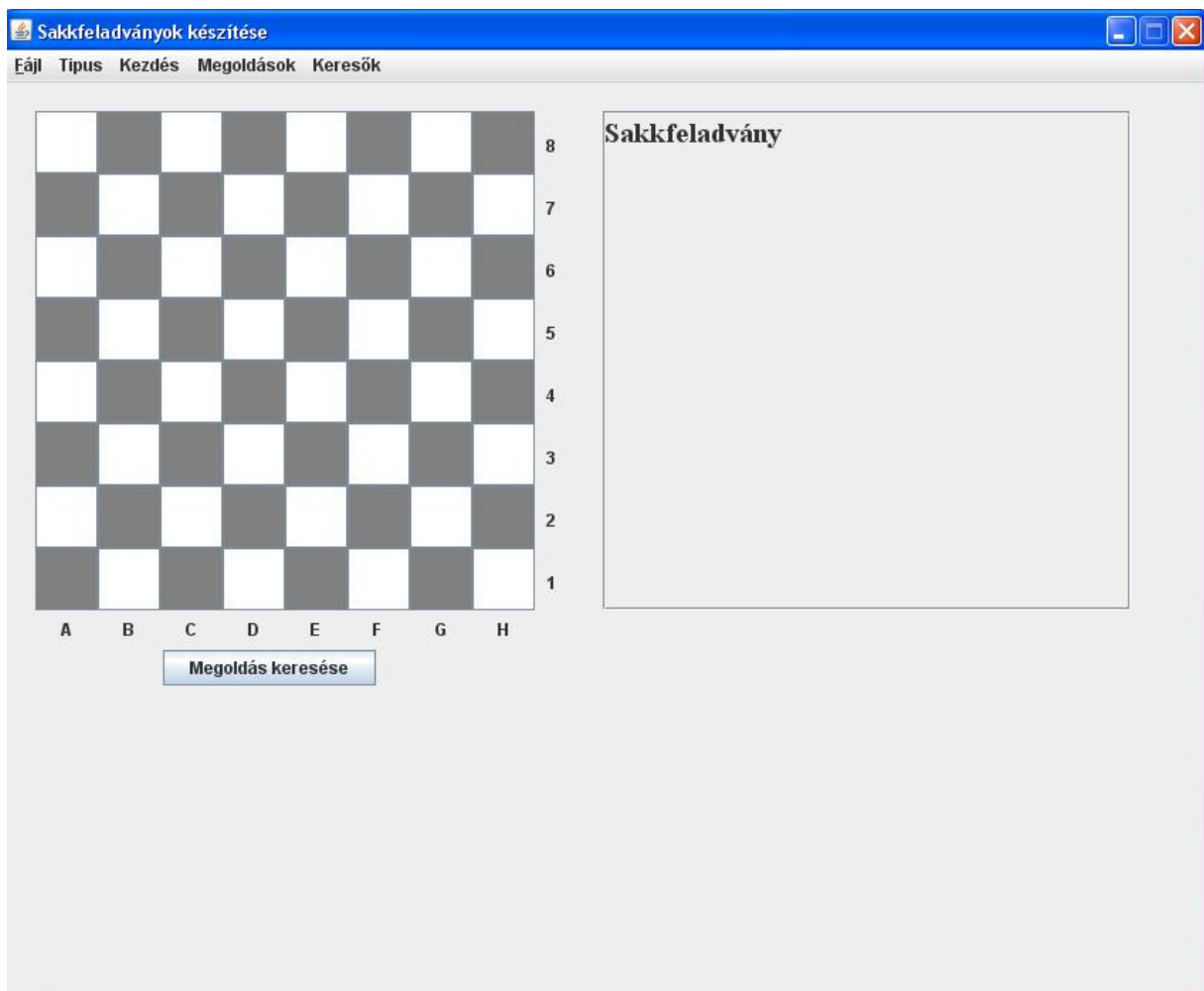
Esetemben sakkfeladványokhoz kellene lépést ajánlani, hogy merre lépjenek a felek. Megadott mélység mellett ugyan megoldáshoz vezető lépésajánlatok lennének, de minél jobb heurisztikát kellene választani, ami sakk esetében nagyon nehéz. Nehéz eldönteni egy adott állásból, hogy melyik lépés milyen jó a támogatott játékosnak. Mindezek ellenére a legelterjedtebbek a sakkprogramokban a különféle lépésajánlók:

- MINIMAX algoritmus
- Alfabéta MINIMAX algoritmus
- NEGAMAX algoritmus
- Alfabéta NEGAMAX algoritmus

Egy megoldást kereső rendszer egyszemélyes problémáknál használatos, viszont ebben az esetben sakkfeladványokhoz használok megoldást kereső rendszereket, ugyanis a sakkfeladvány egy rejtvény, visszavezethető egyszemélyes problémára. Ezen okból kifolyólag nem részletezem a lépésajánló algoritmusokat, mivel a programomban nem használom őket, de mindenképpen meg kellett említenem a sakkalgoritmusok szempontjából.

6. Saját fejlesztésű sakkeladvány készítő és megoldó program JAVA nyelven

A program nyílt forráskódú, bárki szabadon használhatja, módosíthatja saját célra, esetleg részeket használhat fel saját munkájához. Célok a programmal az, hogy a szakdolgozatomban tárgyalt tanulmányokat, sakkeladványokat, keresőket egy egészben bemutassam grafikus megjelenítéssel. Noha a program nem tökéletes, de melyik program az? Programom sajátossága a keresőkben és a grafikus megjelenítésben rejlik, mivel nem a megszokott lépésajánló algoritmust alkalmaztam, hanem megoldást kereső rendszert, ami minden lehetséges kimenetelű lépéssorozatot vizsgál a megadott feltételig. A táblára kényelmesen egy előugró menüből rakhatjuk fel a bábukat tetszőleges helyre, vagy akár törölhetjük őket. Ha elindítjuk a programot egy ilyen képernyőt kapunk:



Egy üres sakktábla felcímkézve a mezőkkel, egy jobboldali információs ablak, amiben majd a kiválasztott megoldás lépéseit láthatjuk oda-vissza. Egy „Megoldás keresése” gomb, amivel keresünk majd, ha felállítottuk a bábukat oda ahova szeretnénk. Egy menüsor, amely a beállításokat tartalmazza majd. A fájl menüpont alatt van lehetőségünk egy állás elmentésére vagy éppen betöltésére.

A szakdolgozatomban megtalálható sakktáblák ábrákat és a bábuk bemutatási ábráit saját programommal készítettem.

6.1 A programban használt sakktábla elemek

Vegyük sorra milyen sakktábla elemeket használok:

- Kompozíciós tanulmányok közül az angol tanulmányt valósítom meg, ami szerint mindenféle lépéslehetőséget vizsgál és a sok megoldást helyezi előtérbe. Szabad játék jellemzi, így nem feltétlenül kell sötétnek ellenálló lépéseket megtennie, hanem minden olyan lépése is számít, ami legális, noha botornak tűnik.
- Sakkbábuk közül bármennyi felhelyezhető a táblára, nem követem a sakk konvencióját, miszerint megadott számú bábu lehet a táblán egyfajta figurából. A programom szerint lehetséges ez is, maximum nem talál megoldást, ha rosszul vagy hiányosan helyezzük fel a figurákat. Annyi megkötést alkalmazok, hogy amelyik fél kezd, ez a fél rendelkezzen a matt adáshoz elegendő bábuval. Ez a következőket jelenti:
 - A mattot kapó félnek rendelkeznie kell királlyal
 - A mattot adó félnek
 - minimum király + vezér
 - minimum király + bástya
 - minimum király + 2 futó
 - minimum király + futó + huszár

Ha ezek teljesülnek, akkor található megoldás, attól függően, hogy vannak felállítva a figurák.

- Speciális lépések közül a programban nem valósítom meg egyiket sem a visszafelé irányuló analízis (retrograde analysis) miatt. Ha készítünk egy feladványt, a programban oda-vissza lehet lépkedni az adott állásokon, viszont ha a sakk matt állapotából, akarunk visszafele következtetni, akkor nem lenne egyértelmű a következtetés. Ez az analízis viszonylag új feltételfajta a sakkeladványok körében.
- A táblára saját ízlésünk szerint pakolhatunk figurákat, szóval a pozíciót saját magunk szerkesztjük. PGN⁵ fájlok megnyitására nincs lehetőség, hisz nem sakkjátszmákat akarunk végignézni. Csakis a saját magunk feladványait menthetjük el vagy nyithatjuk meg egyedi formátumban. Persze a figurákat felállíthatjuk pontosan ugyanúgy, mint egy kész feladványt, és megadhatjuk a feltételeket, viszont megoldáskeresésnél a program egyedi megoldáskeresője más sorrendben vagy más megoldást is találhat, mint ami a feladvány szerint van.
- A program a közvetlen feladványokra épít, szóval kétlépéses, háromlépéses és többlépéses (a programban a többlépéses 4 lépéset jelent, ennél nagyobb lépésszámnál a keresés nagyon sok időt vesz igénybe) feladványokat lehet vele megoldani és egyes normál segítőmatt feladványokat is, mivel az összes lehetséges lépéslehetőséget vizsgáljuk, beletartoznak azok a lépések is, amikor a két fél közreműködik a matt adásban. Önmatt, reflexmatt, sorozatlépéses feladványok a reprezentáció módosításával megoldhatók lennének, de alapjában véve közvetlen mattokra összpontosítottam, ezek a feladványok mindenki számára a legkönnyebben megérthetők és ezek a legelterjedtebbek. Tündérfeladványokat nem kezel a program.
- Megoldások tekintetében kereshetünk egy megoldást vagy az összes megoldást, értelemszerűen az összes megoldás jóval időigényesebb, mint egy megoldás megtalálása
- Tetszőleges nehézségű feladványt összeállíthatunk, a keresés ideje attól fog függeni, hogy mennyi figurát helyeztünk el a táblán, ha sokat akkor több ideig tart a keresés.

⁵ Portable Game Notation – számítógép által feldolgozható formátum már lejátszott sakkjátszmák tárolására, sok sakkprogram ismeri ezt a formátumot.

6.2 Grafikus megjelenítés

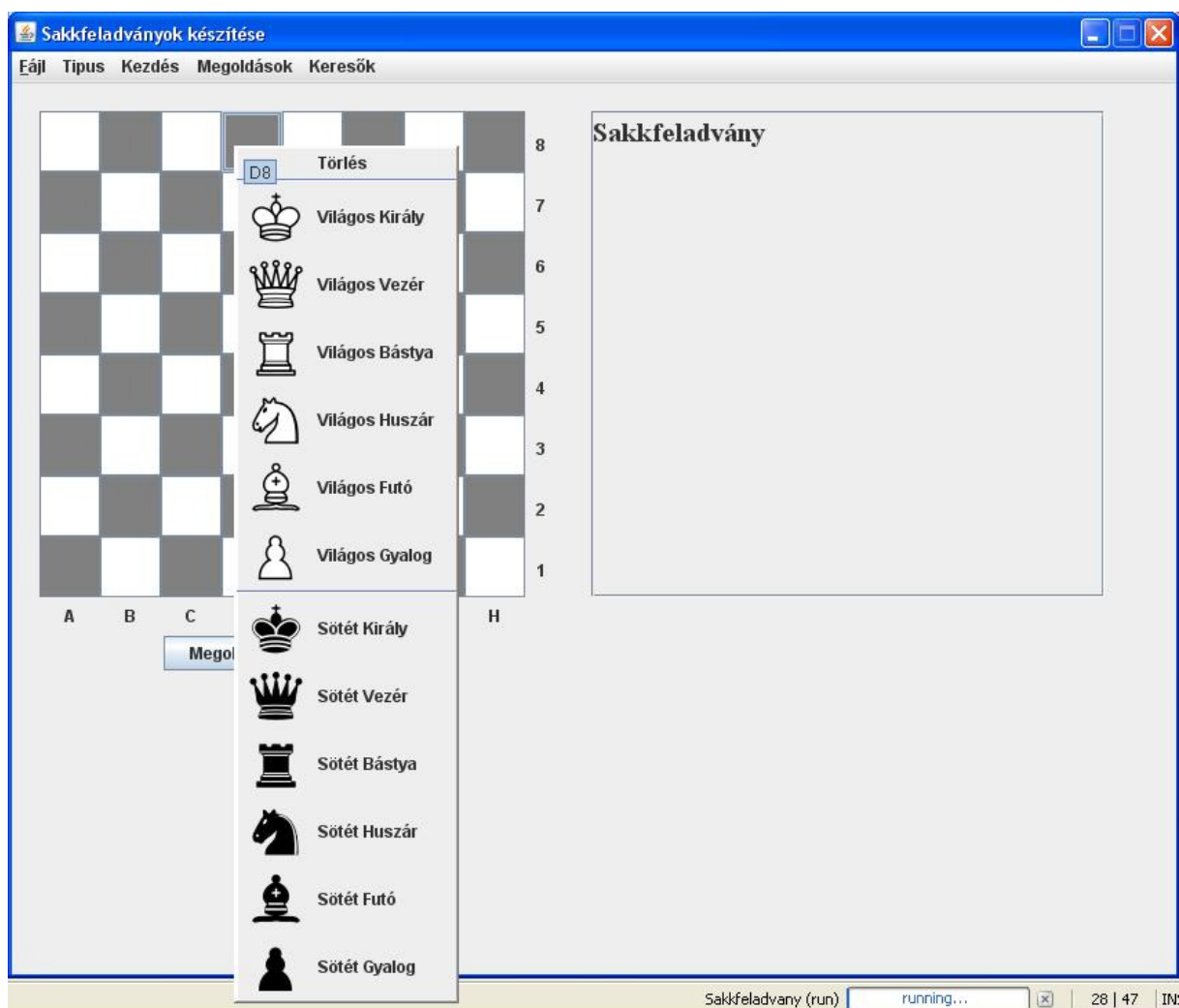
A program képernyője egy ChessBoard nevű osztály, amely a JFrame osztályból van származtatva: `public class ChessBoard extends JFrame{ ... }`

A képernyőn található komponensek egy listában vannak tárolva, mivel sok ilyen komponens van: `private LinkedList<Component> ablakelemek;`

Ha egy ablakelemet akarok elérni, akkor kikeresem a listából és a nevével hivatkozok rá.

Az egyes komponenseket inicializáló metódusokkal hozom létre ezek a következők:

- **initSakkTabla()** – a sakktáblát hozza létre, minden mezőhöz beállítja a ToolTipText-eket és minden mezőhöz beállít egy MouseListener osztályt, ami figyeli, hogy melyik mezőre akarunk előugró menüt nyitni, ahonnan ki tudjuk választani, hogy milyen figurát akarunk elhelyezni.



A sakktábla mezői *Pictures* nevű osztályok, amely a *JButton*⁶ osztályból vannak származtatva.

A *Pictures* osztály tartalmazza az összes figura képét, tehát ha egy mezőre valamilyen figurát akarunk elhelyezni, akkor az adott *Pictures* mezőre beállítjuk az elhelyezni kívánt figura képét. A mezőket a **createButton()** metódussal hozom létre.

- **initPopupMenu()** – az előugró menüt tölti fel *JMenuItem*⁷-ekkel, aminek megadjuk az elem nevét és egy fájlnevet, ami a kép neve.
- **initMenu()** – létrehozza a menüsort, feltölti a megfelelő menüelemekkel a megfelelő menüt.
- **initTextMezo()** – létrehozza a jobb oldali információs ablakot.
- **initGombok()** – létrehozza az ablakon található gombokat.
- **initLabels()** – a sakktábla melletti címkéket hozza létre. (A, B, C, D, E, F, G, H, 1, 2, 3, 4, 5, 6, 7, 8)

6.3 Beállítások

A beállítható lehetőségeket a menüsor tartalmazza.

Fájl menü

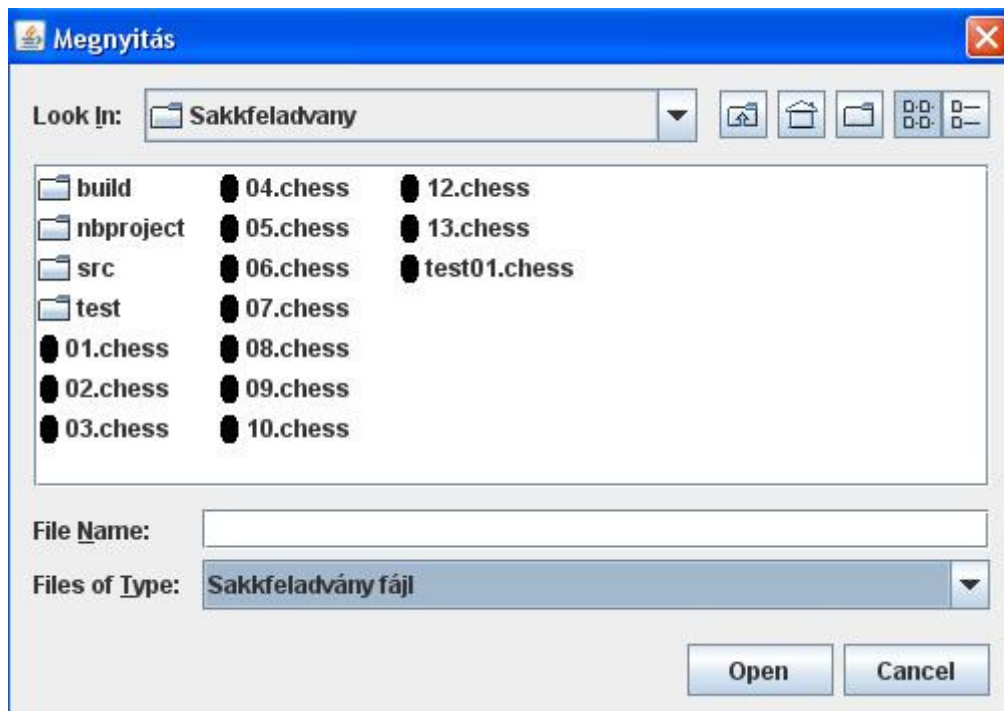
A fájl menü alatt található a megszokott Új, Megnyitás, Mentés, Kilépés menüpontok. Az „Új” menüpont hatására új sakkfeladványt készíthetünk, az előzőleg felrakott tábla elvész. Az alapbeállítások állítódnak be. Az alapbeállítás: Kétlépéses feladvány, világos kezdés, egy megoldás keresése és Backtrack kereső használata.

A „Megnyitás” menüponttal egy megnyitóablak jelenik meg, ahol betallózzhatjuk a korábban már elmentett sakkfeladványainkat. A program a „chess” kiterjesztésű szerializált állományokat tudja megnyitni, amiről majd az Állásmentés és állásbetöltés fejezetben lesz

⁶ <http://java.sun.com/j2se/1.4.2/docs/api/javaw/swing/JButton.html>

⁷ <http://java.sun.com/j2se/1.4.2/docs/api/javaw/swing/JMenuItem.html>

szó. A „Mentés” menüponttal „chess” kiterjesztésű szerializált fájlt menthetünk ki név megadása után. A „Kilépés” menüpont hatására befejeződik a program futása.



Megnyitás ablak

Típus menü

A sakkfeladvány típusát választhatjuk ki, ami egylépéses, kétlépéses, háromlépéses és többlépéses (négy) feladványokat jelent.

Kezdés menü

Itt választhatjuk ki, hogy melyik fél kezdi a lépést.

Megoldások menü

Kiválaszthatjuk, hogy egy vagy az összes megoldást akarjuk keresni.

Keresők menü

Itt választhatjuk ki, hogy backtrack vagy mélységi keresővel akarunk keresni

6.4 Figyelők

A „*listeners*” csomagban található meg a különféle akciófigyelő (ActionListener)⁸ osztályok.

6.4.1 ButtonListener

A JButton komponenseknek ez a figyelő van megadva, kivéve a mezőknek. A „Megoldás keresése”, „Előző”, „Következő” gombok figyelője. Attól függően, hogy mire kattintottunk, lefut a **megoldasKereses()**, **elozoAllapot()** vagy a **kovetkezoAllapot()** metódus.

- **megoldasKereses()** – Ha megnyomjuk a „Megoldás keresése” gombot, akkor változóba összegyűjtődnek a keresőnek szükséges információk és itt hozzuk létre a *SakkFeladvany* osztályt is, ami maga az állapottér-reprezentációs osztály. A keresőnek szükséges változók:
 - **chessboard.megoldasok** – egy vagy több megoldást keressünk (0 és 1 lehet)
 - **sf** – *SakkFeladvany* példány
 - **melyseg** – amit a programban így számolok ki: $melyseg = chessboard.sakkFeladvanyTipusa * 2 - 1$;

A *SakkFeladvany* osztály számára szükséges változók:

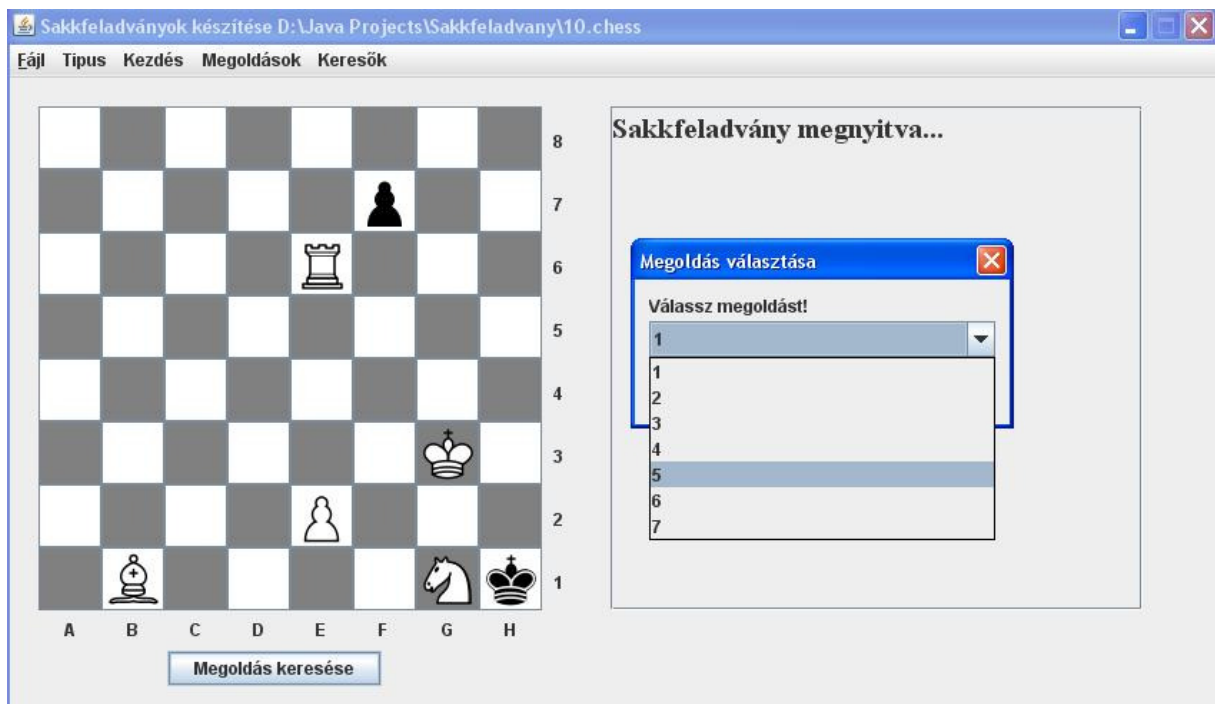
- **chessboard.sakkfigurak** – a felállított sakktábla alapján a kezdőállapot
- **chessboard.kezdoJatekos** – melyik fél kezdi a feladványt

Ha a keresőnek és a sakktáblának megvan minden szükséges információja, akkor indul a keresés. Ha van megoldás akkor egy dialógus ablakban felsorolja a megoldásokat és kiválaszthatjuk, hogy melyiket akarjuk megtekinteni. Ha nincs

⁸ <http://java.sun.com/j2se/1.4.2/docs/api/java/awt/event/ActionListener.html>

megoldás, akkor a jobb oldali információs ablakban kiírja a program, hogy nincs megoldás. Ha találtunk megoldást vagy megoldásokat, akkor egy megoldás kiválasztása után aktív lesz az „Előző” és „Következő” gomb és megnézhetjük, hogy a kezdőállapotból hogy jutunk el a célállapotig. Ha másik megoldást akarunk megtekinteni, akkor kattintsunk a „Megoldások” gombra és előjön a dialógusablak, amiből választhatunk más megoldást.

- **elozoAllapot()** – az „Előző” gomb megnyomására a megoldás előző állapota kerül a sakktáblára, kivéve kezdőállapot esetén.
- **kovetkezoAllapot()** – a „Következő” gomb megnyomására a megoldás következő állapota kerül a sakktáblára, kivéve célállapot esetén.



megoldás választása



6. megoldás kiválasztása után

6.4.2 MenuListener

Ez a figyelő felelős a menüsorban mindegyik menüben a beállításokért. Ha egy menü alatt kiválasztunk egy beállítást, akkor ez a figyelő beállítja a megfelelő változó értékét, amik később szükségesek lesznek a sakkfeladvány szempontjából vagy a sakkfeladvány elmentésénél. A beállítandó változók:

- **sakkFeladvanyTipusa** – 1, 2, 3, 4
- **kezdoJatekos** – 'W', 'B'
- **megoldasok** – 0, 1 (0 jelenti az 1 megoldást, 1 jelenti az összes megoldást)
- **kereso** – 0 esetén backtrack, 1 esetén mélységi

6.4.3 PopupListener

Ez a figyelő felelős azokért az eseményekért, amikor egy figurát helyezünk el a táblára. A már feltöltött előugró ablakból, ha kiválasztunk egy figurát, akkor, amelyik mezőn állunk, arra a mezőre beállítja az adott figurát. Ugye az összes mezőnek meg van adva ez a figyelő. Így kényelmesen elhelyezgethetjük figuráinkat, nem kell mezőneveket megadni, hogy mit hova helyezzen.

Ez a 3 figyelő a legfontosabb, a sakkeladványok készítésében nagyban szerepet játszó osztályok. A „listeners” csomagban a többi figyelő és egyéb osztályok apróbb finomításokat látnak el, például megnyitáskor a megnyitható sakkeladványnak különböző az ikonja a többi fájl ikonjától, és más apró dolgokat látnak el.

6.5 Állásmentés és állásbetöltés

A program sajátos betöltést és mentést alkalmaz, nem követem azt a trendet, miszerint PGN fájlokat is lehessen megnyitni. Állások kimentésére és betöltésére egy *AllasMentes* nevű osztályt használok, ami implementálja a *Serializable*⁹ interfészt. Ez az osztály a „sakk” csomagban található. Ebben az osztályban összegyűjtöm az összes olyan információt a sakktábláról, ami szükséges ahhoz, hogy betöltéskor ugyanazokat az állapotokat elérjük. Ezek az információk a már tárgyalt beállítások a menüben, úgy, mint a sakktábla típusa, kezdő játékos, megoldások száma és a használt kereső. Kiegészül még egy egész számokat tartalmazó mátrix-al, ami a sakktáblának felel meg, és a *SakkFeladvany* számára feldolgozható adatokat tartalmaz. Ez egy 12x12-es mátrix, ami manapság használatos a 8x8-as helyett a huszárlépések miatt. Állásmentéskor egy *AllasMentes* példány jön létre, beállítjuk az adattagjait a jelenlegi megfelelő változók értékeivel és szerializáljuk az *ObjectOutputStream*¹⁰ osztály segítségével. Állásbetöltéskor a „chess” kiterjesztésű szerializált fájlokat tudjuk megnyitni az *ObjectInputStream*¹¹ osztály segítségével, amiből megnyitás után visszaolvassuk az elmentett beállításokat és beállítjuk a megfelelő változókba. A mátrix alapján, amit kimentettünk, feltöltjük a sakktábla mezőit a megfelelő figurával az **addPictures()** metódus (*ChessBoard* osztályban található) segítségével. Csak a beállításokat mentjük el és a sakktábla alapállását, így a sakktábla megoldásait nem tároljuk, de minden szükséges információt elmentettünk, hogy újra ugyanolyan körülmények között keressünk megoldásokat. Hogy tudjuk éppen melyik fájllal dolgozunk, a fejlécen megjelenik az elmentett fájl neve vagy betöltéskor a beolvasott fájl neve.

⁹ <http://java.sun.com/javase/6/docs/api/java/io/Serializable.html>

¹⁰ <http://java.sun.com/j2se/1.4.2/docs/api/java/io/ObjectOutputStream.html>

¹¹ <http://java.sun.com/j2se/1.4.2/docs/api/java/io/ObjectInputStream.html>

6.6 Állapottér reprezentáció megvalósítása

Az $\langle A, kezdő, C, O \rangle$ négyesben mindegyik halmaznak egy osztály felel meg. Ezek az osztályok az „állapotter” csomagban és a „sakk” csomagban találhatóak meg. Egy adott problémától függetlenül az *Allapot* absztrakt osztály az állapot, az *Operator* absztrakt osztály az operátor. Ezekből az osztályokból származtatunk, ami majd a saját problémánk állapota és operátorai lesznek. Az így származtatott *SakkFeladvany* osztály és a *Move* osztály már az én konkrét problémám osztályai, amik a „sakk” csomagban találhatóak. Egy *SakkFeladvany* osztály példányosítása lesz a kezdőállapot. Ezen osztály főbb metódusai:

- boolean VegAllapot()
- boolean eloFeltetel (Operator o)
- Allapot Alkalmaz (Operator o)
- Azt, hogy egy állapot célállapot-e a *VegAllapot()* metódussal vizsgálom, ami igaz értékkel tér vissza, ha teljesíti a célfeltételeket és hamissal, ha nem. Az operátor alkalmazási előfeltételek vizsgálatát egy állapotra az *eloFeltetel (Operator o)* metódus végzi. Ha egy állapotra alkalmazható az „o” operátor akkor igazzal tér vissza, egyébként nemmel. Egy állapotra alkalmazni egy alkalmazható operátort az *Allapot Alkalmaz (Operator o)* metódussal tudok, ami egy állapotot ad visszatérési értékül. A hívó állapotra alkalmazzuk az operátort és előáll egy új állapot.

A reprezentációnak át kell adni egy már említett egészeket tartalmazó 12x12-es mátrixot és a lépő játékost. A mátrix tartalmazza a kezdőállapotot, amit felállítottunk a sakktáblán, melynek elemei a következők lehetnek:

```

private static final int uresMezo = 0;
private static final int whiteKing = 1;
private static final int whiteQueen = 2;
private static final int whiteRook = 3;
private static final int whiteKnight = 4;
private static final int whiteBishop = 5;
private static final int whitePawn = 6;
private static final int blackKing = 7;
private static final int blackQueen = 8;
private static final int blackRook = 9;
private static final int blackKnight = 10;
private static final int blackBishop = 11;
private static final int blackPawn = 12;
private static final int ervenytelenMezo = 13;

```

12x12-es mátrix lehetséges értékei

Az üres sakktábla mátrix formában a következőképpen néz ki:

```

public int[][] sakktabla = new int[][] {
    {13,13,13,13,13,13,13,13,13,13,13,13},
    {13,13,13,13,13,13,13,13,13,13,13,13},
    {13,13, 0, 0, 0, 0, 0, 0, 0, 0,13,13},
    {13,13, 0, 0, 0, 0, 0, 0, 0, 0,13,13},
    {13,13, 0, 0, 0, 0, 0, 0, 0, 0,13,13},
    {13,13, 0, 0, 0, 0, 0, 0, 0, 0,13,13},
    {13,13, 0, 0, 0, 0, 0, 0, 0, 0,13,13},
    {13,13, 0, 0, 0, 0, 0, 0, 0, 0,13,13},
    {13,13, 0, 0, 0, 0, 0, 0, 0, 0,13,13},
    {13,13, 0, 0, 0, 0, 0, 0, 0, 0,13,13},
    {13,13,13,13,13,13,13,13,13,13,13,13},
    {13,13,13,13,13,13,13,13,13,13,13,13}
};

```

A mátrix lehetséges értékei közül az 1 és 12 közötti értékek jelentik a bábukat, ezeket helyezhetjük el a 0 helyekre, amik az üres mezőt jelentik. A 13-as mezők a huszárugrás miatti ellenőrzéseket kerülnek ki, így ha a huszár érvénytelen mezőre lép, akkor nem keletkezik kivétel a keresés során csak illegális lépés.

A *Move* osztály felel meg az operátoroknak, paraméterként megadjuk neki a következőket:

- milyen bábut mozgatunk
- jelenlegi pozíciójának x koordinátája
- jelenlegi pozíciójának y koordinátája
- a célmező x koordinátája
- a célmező y koordinátája
- ha van leütött bábu, akkor a bábura vonatkozó információk, ha nincs akkor null

A bábut azonosíthatjuk a mátrixbeli számértékével, az x és y koordináták pedig a mátrix sor- és oszlopindexei. Egy segédosztályt is használok egy adott bábu információinak összegyűjtésére. Ezeket akkor lehet használni például, ha leütöttek egy bábut vagy éppen sakkot ad egy bábu. A bábu számértékét és a pozícióját tárolja. Az osztály neve *SakkBabu*.

Rengeteg segédmetódust használok a legális lépések vizsgálatára, az adott állásra alkalmazható operátorok újraszámolására, sakk vizsgálatára. A 3. fejezetben részletezett információk alapján írtam meg őket, a figurák hogyan tudnak lépni, melyik csak halad, melyik ugrik, saját bábu vizsgálata, királyközelítés, saját sakkadás, sakk elhárítása. Ezeknek a metódusoknak a forráskódjai a mellékletben (és az egész program forráskódja) megtekinthetők.

6.7 A programban használt keresők

A megoldást kereső rendszerek közül az úthosszkorlátos backtrack keresőt és egy olyan mélységi keresőt, ami megadott mélységig terjeszti ki a keresőfát. Az adatbázis egy *Csucs* nevű osztály, mely a következőket tartalmazza:

```
protected Allapot allapot ;  
protected Csucs szulo ;  
protected Operator op ;  
protected int melyseg ;
```

Az „allapot” az aktuális csomópont állapota, a „szulo” a szülő csomópontra mutat, az „op” a szülő csomópont állapotára alkalmazott operátor, a „melyseg” pedig a csomópont keresőfabeli mélysége. Ezek az információk az adott keresőtől függően kiegészülnek további adatokkal. Backtrack esetén az aktuális csomópont állapotára alkalmazható operátorok listájával, ezt úgy érem el, hogy a *Csucs* osztályból származtatok egy osztályt, aminek a neve *BTCsucs*:

```
public class BTCsucs extends Csucs{protected LinkedList<Operator> operatorok ; ... }
```

Mélységi kereső esetén ez a plusz információ a nyílt és zárt csomópontokat nyilvántartó listák lesznek:

```
protected LinkedList<Csucs> nyiltCsucsok;
```

```
protected LinkedList<Csucs> zartCsucsok;
```

A backtrack az aktuális utat veremben tárolja: `protected Stack<BTCsucs> verem ;`

A keresők egy absztrakt *Kereso* osztályból vannak származtatva, ami a közös információkat tartalmazza. Ilyen közös információ például a célállapotokat tartalmazó lista, ami minden keresőnél ugyanaz. A backtrack kereső esetén további információkat lehet megadni. Ezek az információk az úthosszkorlát és a körök figyelése, megoldás száma, amit a *BackTrack* származtatott osztály különböző paraméterszámú konstruktoraival állíthatunk be. A *BackTrack* vezérlője a **Keres()** metódus, amiben egy végtelen ciklusban alkalmazom az operátorokat az állapotokra, felépítem az aktuális utat a beállított úthosszkorlát mélységig. Ha talált megoldást az aktuális úton, akkor eltároljuk és attól függően, hogy egy vagy több

megoldást keresünk, kilépünk a végtelen ciklusból. Ha az adott úton nincs megoldás, akkor a kereső visszalép egy szintet. Végül, ha a verem üres, véget ért a keresés.

A *MelysegiKereso* származtatott osztály esetén is konstruktoraival állíthatjuk be a szükséges információkat. Megadjuk a kezdőállapotot, mélységet és egyéb jellemzőket, amik lehetnek például, hogy csakis a célállapotot tároljuk-e el vagy éppen egy vagy több megoldást keresünk. A mélységi kereső vezérlője szintén a **Keres()** metódus, viszont máshogy működik, mint a backtrack vezérlője. Alkalmaz minden alkalmazható operátort a kiterjesztendő csomópont állapotára, az előálló csomópontok a kiterjesztett csomópont gyerekcsomópontjai lesznek. A kiterjesztett csomópont a *nyiltCsucsok* listájából törlődik és a *zartCsucsok* listájába kerül. Kiterjesztésre mindig a legnagyobb mélységű csomópontot választja a vezérlő, így úgy működik, mint egy verem. Azonos mélységszám esetén a listába legutoljára bekerült csomópontot terjeszti ki. Ha a megadott mélységkorlátot elértük, akkor nem építi tovább a keresőfát az adott ágon, hanem veszi a következő legnagyobb mélységű csomópontot. Így ezt is tekinthetjük visszalépésnek. A klasszikus mélységi keresőben nincs megadva mélységkorlátozás, ezt én építettem be a keresőbe. Mélységkorlátozás nélkül irányítatlanul és végtelenül építené fel a keresőfát, olyan lenne, mint egy végtelen lépéses sakkfeladvány. Backtrack és mélységi keresőnél nincs lépésajánlás, minden lehetséges csomópontot kiterjesztünk (backtrack esetén alkalmazzuk az alkalmazható operátorokat). Ez időigényes, viszont mindenféle megoldási lehetőséget megtalál. Ha a *nyiltCsucsok* lista üres, nincs több kiterjesztendő csomópont, akkor a keresés véget ér.

Az absztrakt *Kereso* osztálynak van még egy *void Kiir(Csucs cs){...}* metódusa, ami egy megoldást ad, vagyis a kezdőállapottól kezdve a célállapotig hogyan jutottunk el, mely operátorokat alkalmazva és milyen közbülső állapotok voltak. Ez hasznos lehet a jobb oldali információs ablakban való kiíráshoz. Mivel mindkét kereső ebből az osztályból van származtatva, így használható mindkettőnél. A backtrack kereső vezérlőjét a függelék 3. ábrája mutatja, a mélységi kereső vezérlőjét pedig a függelék 4. ábrája.

7. Összegzés

Diplomamunkámban ismertettem a sakkeladványokkal kapcsolatos legismertebb tanulmányokat, amik alapján készülnek a sakkeladványok. Bemutattam a sakk elemeit, amik ismerete alapvető a sakkhöz és a sakkeladványokhoz is. Ezek ismerete nélkül aligha lenne érdeemes elolvasni a diplomamunka további fejezeteit, így mindenképpen fontosnak tartottam a bábuk bemutatását, lépéseiket ábrákkal egybekötve. Egy nagy fejezetben tárgyaltam a sakkeladványok tulajdonságait, típusait, mindegyik típust egy-egy példán keresztül végigvezetve, így megmutatva az adott feladványtípus sajátosságait. Részleteztem, hogy milyen tulajdonságokkal kell, vagy milyen tulajdonságai lehetnek egy jól megszerkesztett feladványnak. Manapság nagyon sokféle feladványtípus létezik, diplomamunkám keretein túllyúlna az összes bemutatása. Például a tündérfeladványok egy külön diplomamunkaként is megállná a helyét, de említhettem volna akármelyik más típust is. Mivel sakkprogrammal szemléltettem a feladványok készítését és megoldását, így fontosnak tartottam egy külön fejezetben tárgyalni a sakk és a mesterséges intelligencia kapcsolatát. A használt fogalmak és algoritmusok ismertetése nélkül a program megértése jóval nehezebb lett volna az olvasó számára. E fejezet alapján mutattam be programomban a megvalósított reprezentációt és a keresőket. A programom sakkeladvány elemeit a sakkeladványok fejezet alapján mutattam be így párhuzamosítva a diplomamunkám fejezeteit a program részeivel. Végül az utolsó fejezetben, az előző fejezetekben végigtárgyalt minden információ ismeretében bemutattam a JAVA nyelven készített programomat. Kizárólag JAVA SE technológiát alkalmaztam, mindenki számára érthető kódot próbáltam írni. Részleteztem minden olyan fontos mozzanatot a programban, amik a sakkeladványok, mesterséges intelligencia szempontjából fontosak voltak. Külön kis részben tárgyaltam a grafikus megjelenítést, a kényelmesen elhelyezhető bábukat előugró ablakból, a menüsor elemeit, amik minden fontos beállítást tartalmaztak. Mivel egy program sem tökéletes, így az én programom sem az. Számos hiányossága van, de ezek a hiányosságok fejleszthetőek. Akár az összes tárgyalt megoldást kereső rendszerrel lehetne megoldást keresni, vagy akár az összes lépésajánló algoritmust is lehetne használni. A programban az osztályokat úgy írtam meg, hogy akár lépésajánlóval is lehessen megoldásokat keresni, ha egy lelkes olvasómnak ahhoz lenne kedve. Az ehhez szükséges osztályok a mellékletben megtalálhatóak.

8. Irodalomjegyzék

1. Várterész Magdolna – Mesterséges intelligencia 1 előadás jegyzet
2. http://en.wikipedia.org/wiki/Composition_school
3. http://en.wikipedia.org/wiki/Chess_problem
4. <http://en.wikipedia.org/wiki/Selfmate>
5. <http://en.wikipedia.org/wiki/Helpmate>
6. <http://en.wikipedia.org/wiki/Reflexmate>
7. <http://www.chessville.com/Wong/glossary.htm>
8. http://www.magysakkszerzok.com/mssz_lista.htm
9. <http://www.chessproblems.com/>
10. <http://java.sun.com/j2se/1.4.2/docs/api/>
11. <http://java.sun.com/javase/6/docs/api/>
12. http://en.wikipedia.org/wiki/List_of_grandmasters_for_chess_composition

9. Függelék

1. táblázat – nagymester fokozattal rendelkező sakkeladvány készítőik

Év	Név	Ország
1972	Genrich Kasparjan (†)	Szovjetunió
1972	Lev Loshinskij (†)	Szovjetunió
1972	Comins Mansfield (†)	Egyesült Királyság
1972	Eeltje Visserman (†)	Hollandia
1976	Vladimir Bron (†)	Szovjetunió
1976	Jindrich Fritz (†)	Csehszlovákia
1976	Vladimir Korolkov (†)	Szovjetunió
1976	Vladimir Pachman (†)	Csehszlovákia
1976	György Paros (†)	Magyarország
1976	Nenad Petrović (†)	Horvátország
1980	György Bakcsi	Magyarország
1980	Hrvoje Bartolović (†)	Horvátország
1980	Bo Lindgren	Svédország
1980	Gia Nadareishvili (†)	Szovjetunió
1980	Valentin Rudenko	Ukrajna
1984	Claude Goumondy	Franciaország
1984	Iosif Krikheli (†)	Szovjetunió
1984	Petko Petkov	Bulgária
1984	Hans Peter Rehm	Németország
1984	Touw Hian Bwee	Indonézia
1988	Cor Goldschmeding (†)	Hollandia
1988	Alexandr Grin-Guljaev (†)	Szovjetunió

Év	Név	Ország
1988	Ernest Pogosjancz (†)	Szovjetunió
1988	Jakov Vladimirov	Szovjetunió
1988	Milan Vukčević (†)	Egyesült Államok
1989	Herbert Ahues	Németország
1989	Viktor Chepizhnij	Szovjetunió
1989	Emilian Dobrescu	Románia
1990	David Gurgenidze	Grúzia
1990	Jacobus Haring (†)	Hollandia
1992	Fadil Abdurahmanović	Bosznia Hercegovina
1992	Jan Rusinek	Lengyelország
1993	Venelin Alaikov (†)	Bulgária
1993	Michel Caillaud	Franciaország
1993	Andrej Lobusov	Oroszország
1993	Norman Macleod (†)	Egyesült Királyság
1993	Byron Zappas (†)	Görögország
1995	Michael Keller	Németország
1995	Alexandr Kuzovkov	Moldova
1996	Toma Garai	Egyesült Államok
1996	Živko Janevski	Macedónia
2001	Virgil Nestorescu	Románia
2004	Unto Heinonen	Finnország
2004	Jean-Marc Loustau	Franciaország
2004	Mikhail Marandyuk	Ukrajna
2004	Waldemar Tura	Lengyelország
2005	Udo Degener	Németország

Év	Név	Ország
2005	Nikolai Kralin	Oroszország
2005	Franz Pachl	Németország
2005	Oleg Pervakov	Oroszország
2007	Alexandr Feoktistov	Oroszország
2007	Yves Cheylan	Franciaország
2007	Marjan Kovačević	Szerbia
2007	Miodrag Mladenović	Szerbia
2007	Valery Shanshin	Oroszország
2007	Valery Shavyrin	Oroszország
2007	Anatoly Slesarenko	Oroszország
2009	Uri Avner	Izrael
2009	Andrej Selivanov	Oroszország
2009	Camillo Gamnitzer	Ausztria

2. táblázat – nagymester fokozattal rendelkező sakkfeladvány megoldók

Év	Név	Ország
1982	Pauli Perkonoja	Finnország
1984	Kari Valtonen	Finnország
1984	Milan Velimirović	Jugoszlávia
1985	Ofer Comay	Izrael
1988	Roland Baier	Svájc
1988	Marjan Kovačević	Jugoszlávia
1988	Arno Zude	Németország

Év	Név	Ország
1991	Georgi Evsejev	Oroszország
1993	Michael Pfannkuche	Németország
1997	Jonathan Mestel	Egyesült Királyság
1997	Sergej Rumyantsev	Oroszország
1998	Ram Soffer	Izrael
1999	Jorma Paavilainen	Finnország
2000	Boris Tummes	Németország
2001	Noam Elkies	Izrael
2002	Michel Caillaud	Franciaország
2002	Graham Lee	Egyesült Királyság
2002	Piotr Murdzia	Lengyelország
2004	John Nunn	Egyesült Királyság
2004	Dolf Wissmann	Hollandia
2007	Alexandr Azhusin	Oroszország
2008	Miodrag Mladenović	Szerbia
2008	Andrey Selivanov	Oroszország
2008	Bojan Vučković	Szerbia
2008	Eddy van Beers	Belgium
2008	Vladimir Podinić	Szerbia

3.ábra – backtrack kereső vezérlője JAVA nyelven

```
while(true){
    if(verem.isEmpty())
        break;
    BTCsucs bt = verem.peek() ;
    if( ((SakkFeladvany)bt.getAllapot()).VegAllapot() && bt.getMelyseg() == korlat ){
        terminalisok.add(bt);
        if(osszesmegoldas){
            System.out.println( (terminalisok.size()) + " megoldast talaltam.");
            verem.pop();
            continue;
        }
        else
            break;
    }
    if( korlat > 0 && bt.getMelyseg() == korlat ){
        System.out.println("Elertuk a korlatot.");
        verem.pop();
        continue;
    }
    LinkedList<Operator> ops = bt.getOperatorok();
    if(ops.isEmpty()){
        verem.pop();
        continue;
    }
    Operator op = ops.iterator().next();
    BTCsucs uj = new BTCsucs(bt,op);
    letrehozottCsucsokSzama++;
    System.out.print(uj.getAllapot().toString());

    if(korfigyeles && verem.contains(uj))
        System.out.println("Kort talaltam.");
    else
        verem.push(uj);

    ops.remove(op);
}
```

4. ábra – mélységi kereső vezérlője JAVA nyelven

```
while(true){
    if(nyiltCsucsok.isEmpty())
        break;
    Csucs csomopont = nyiltCsucsok.get( nyiltCsucsok.size()-1 );
    if ( ((SakkFeladvany)csomopont.getAllapot()).VegAllapot() && csomopont.getMelyseg() == melyseg ){
        terminalisok.add(csomopont);
        if(osszesmegoldas){
            System.out.println( (terminalisok.size()) + " megoldast talaltam.");
            nyiltCsucsok.remove( nyiltCsucsok.size()-1 );
            zartCsucsok.add(csomopont);
            continue;
        }
        else
            break;
    }
    if( csomopont.getMelyseg() < melyseg )
        kiterjeszt(csomopont);
    else{
        zartCsucsok.add(csomopont);
        nyiltCsucsok.remove(csomopont);
    }
}
```