Szakdolgozat

Pongor Gábor

Debrecen
2009

Debreceni Egyetem
Informatikai Kar

Egy kétszemélyes játék számítógépes megvalósítása

Témavezető:
Mecsei Zoltán
Egyetemi tanársegéd

Készítette:
Pongor Gábor
Programozó matematikus

Debrecen
2009

Tartalomjegyzék

Bevezetés

A játékok a történelem kezdete óta jelen vannak, és nem csak, mint kellemes időtöltés, hanem mint olyan tevékenység, amely az emberi tulajdonságokat fejleszti, legyen az logikai készség, kézügyesség, vagy fizikai erőnlét. Számítógépes környezetben sem csak szórakozás céljából készültek játékprogramok.

1950-ben Alan Turing tette fel a kérdést, hogy vajon egy számítógép rendelkezhet-e a gondolkodás képességével. Ezzel a gondolattal tette meg a mesterséges intelligencia az első lépést, hogy a tudományos-fantasztikus regények témájából, számítógép-tudománnyá nője ki magát. Alan Turing felvetette, hogyha egy gép viselkedése megkülönböztethetetlen az emberi viselkedéstől, akkor az egyet jelent-e azzal is, hogy intelligens. Ennek bizonyítását szolgáló, kísérletet nevezzük ma Turing tesztnek. A leggyakoribb gyakorlati elképzelése a kísérletnek, amikor a számítógép és az ember természetes nyelven kommunikálnak egymással. A gép akkor megy át az intelligencia teszten, ha a kommunikáció emberi résztvevőjével el tudja hitetni, hogy egy másik emberrel beszélget. A feladat nehézsége abból áll, hogy gépnek fel kell ismernie, és értelmeznie kell a természetes nyelvet, valamint rendelkeznie kell a természetes nyelven megfogalmazott válasz létrehozásához szükséges tudással. A Turing tesztre alapozva a korai mesterséges intelligenciakutatók úgy vélték, hogyha a számítógép egy általuk komplexnek tartott problémát meg tud oldani, akkor lehetséges intelligens gépet alkotni. A logikai készséget igénylő kétszemélyes játékok, mint komplex problémák tökéletesnek bizonyultak, hogy az intelligens döntéshozó algoritmusok és technikák hatékonyságát teszteljék. A mai mindennapi életben is alkalmazott mesterséges intelligencia technikák nagy része összekapcsolható, a korai kétszemélyes játékok implementációinál is felhasznált megoldásokra, valamint ezek a megoldások elősegítették játékok matematikai megközelítését is magába foglaló tudomány fejlődését, melyet ma játékelméletnek nevezünk. A matematika ezen ágát sikeresen alkalmazzák az informatika mellett olyan területeken is, mint a közgazdaságtan és a pszichológia. Az igazi kihívás a kétszemélyes játékokkal kapcsolatban egy olyan program megalkotása, amely képes túlszárnyalni az emberi teljesítményt. Ehhez egyértelműen elsődleges cél az ellenfél legyőzése, vagyis az algoritmusok fő feladata, hogy a győzelemhez vezető utat, stratégiát megtalálják.

Ennek egyik módja a kereső algoritmusok alkalmazása, melyek azon az elven alapulnak, hogy a játék során lejátszható játszmák olyan irányított gráffal, az úgynevezett játékgráffal modellezhetőek, melyben a csomópontok a játszmák során előálló állásokat, valamint az egy csomóponthoz tartozó élek a csomópont által adott állásból megléphető lépéseket reprezentálják. Így tehát ebben a játékráfban minden, a kezdő állást tartalmazó csomópontból kiinduló út, amely egy terminális csúcsban végződik, egy játszmának felel meg. A kereső algoritmusok feladata szűkebb értelemben, hogy a kezdő pozíciót tartalmazó csomópontból olyan utat találjanak a gráfban, amely olyan terminális csomópontban ér véget, mely az ő szemszögéből győztes állást tartalmaz. Egy számítógép közelibb megoldás, hogy a gráfot fára, úgynevezett játékfára egyenesítjük ki a könnyebb kezelhetőség érdekében, oly módon, hogy amennyiben egy csomóponthoz több út is vezet, akkor a csomópont a fában többször szerepel, illetve a fa mélységi szintjeit tekintve a 0. szint a kezdőpozíció, valamint minden páros szint az egyik játékoshoz, illetve minden páratlan szint a másik játékoshoz tartozik. Tehát ha egy konkrét páros szinten *A* játékos következik lépni, akkor az eggyel mélyebben található páratlan szinten *B* játékos lépése következik, valamint még egy szinttel mélyebben ismét *A* következik. Mivel még az egyszerűbb játékok fái is, méretüket tekintve meglehetősen nagyok, ezért nincs arra mód, sem a kereséshez szükséges időt, sem pedig a fa tárolásához igénybevett erőforrások méreteit tekintve, hogy az egész fát egyben kezelje a program a memóriában. Ha a Sakkot tekintjük példának, melyben az átlagos lépésváltások száma 45, és minden állásból átlagosan 35 a szabályos lépések száma, akkor a játékfa mélysége 90, és a terminális csomópontok száma 35^{90} . A fát ezért valamilyen módon méreteiben korlátozni kell. A kereső algoritmusok ezért a teljes játékfa csak egy részfáját járják be. Ennek egyik következménye, hogy a bejárt részfa általában nem tartalmaz olyan csomópontokat, melyek végállásokat reprezentálnak, ezért a programnak magáról a játékról is tartalmaznia kell információkat, hogy el tudja dönteni egy nem terminális állásról, hogy az számára kedvező, vagy sem. A másik következmény hogy a program előre a teljes játszmát soha nem látja, sőt az estek túlnyomó többségében még a részfában előálló részjátszmák közül sem tudja pontosan meghatározni előre, hogy a továbbiakban melyik fog előállni, ugyanis ezt nagyban befolyásolják az ellenfél lépései is. Tehát a keresést az ellenfél minden lépése után újra el kell végezni. Ebből kifolyólag az algoritmusnak egy olyan lépést kell tudnia ajánlani a bejárt részfa gyökerének lépései közül, amely a számára legkedvezőbb állást

tartalmazó, a részfában terminális csomópontként szereplő, csomópont felé vezet, úgy hogy közben figyelembe veszi az ellenfél lehetséges lépéseit is. Ezért célszerűen a kereső algoritmusok helyett a továbbiakban a lépésajánló algoritmusok elnevezést fogom alkalmazni.

Az általam választott kétszemélyes játék egy teljes információjú, véges, zérusösszegű, diszkrét és determinisztikus játék, a Dáma. A szakdolgozat írásának céljai közzé tartozott egy olyan program elkészítése, amely megvalósít egy lépésajánló algoritmust, illetve annak hatékonyságnövelő technikáit, valamint hogy a program képes legyen Dámában, emberi ellenfelek ellen nyerni. Ezen célok maradéktalanul megvalósultak. A program megírásához a Microsoft Visual Studio 2008 keretrendszert és a C# programnyelvet használtam, ebből kifolyólag a program futtatásához a .NET keretrendszer 3.5 verziója szükséges. A program megírásakor elsődleges szempont a program hatékonyságának, a gyorsaságának és a könnyen kezelhetőségének megvalósítása volt.

1. Alapfogalmak

1.1. Játékok osztályozása

A játékok két nagy csoportba sorolhatóak, mely csoportok a szerencsejátékok csoportja, és a stratégiai játékok csoportja. A *stratégiai játékok* olyan játékok, melyben a játékosok ellenőrizhető módon befolyásolják a játék kimenetelét. A stratégiai játékok az őket leíró szabályok által adott jellemzők alapján osztályozhatóak. A játékban résztvevő játékosok számát tekintve beszélhetünk $2,3,\cdots,n$ *személyes* játékokról. *Diszkrétnek* nevezünk egy játékot akkor, ha a játék során előforduló összes állást tekintve valamennyi játékos által alkalmazható legális lépések száma véges. *Véges* egy játék akkor, ha diszkrét, és a játék során előforduló összes játszma véges sok lépés után véget ér. Bármennyire is meglepő de léteznek végtelen játékok is, azonban ezeknek a játékoknak elsősorban a gazdasági életben van szerepük. *Zérusösszegű* egy játék, ha a játékosok veszteségeinek és nyereségeinek összege nulla. Ilyen játék például a Póker, a Sakk és a Dáma. Egy játékot *teljes információjúnak* nevezünk, ha játék során a játékban résztvevő játékosok a játékkal kapcsolatos valamennyi információval rendelkeznek. Teljes információjú játékok többek között a Sakk, a Go és a Dáma, illetve a nem teljes információjú játékok közzé tartozik a Póker és a Bridzs. Ha a játékban a véletlen szerepét tekintjük, akkor beszélhetünk *determinisztikus* és *sztochasztikus* játékokról. Determinisztikus játékok esetében a véletlennek nincs hatása a játék kimenetelére, mint például a Sakkban és a Dámában, viszont a sztochasztikus játékoknál fontos tényezőnek számít, úgymint a Backgammonnál a kockadobás értéke, és a Pókernél a leosztás. Fontos tényező még, hogy a játékosok milyen módon léphetnek a játék során. Ennek kapcsán beszélhetünk *szekvenciális* játékokról, ahol a játékosok felváltva lépnek egymás után, illetve *szimultán* játékokról, ahol a játékosok akár egy időben is léphetnek. A szakdolgozat folyamán a kétszemélyes, diszkrét, véges, zérusösszegű, determinisztikus, szekvenciális és teljesinformációjú játékokról lesz szó.

1.2. Mesterséges intelligencia alapfogalmak

A játékoknak létezik egy formális leírása, melyet *játék reprezentációnak* nevezünk. Jelölje S a játék során előforduló összes állás halmazát, ahol $s_0 \in S$ a kezdőállást

jelöli, $P = \{A, B\}$ a játékosok halmazát, ahol A és B a két játékost és $p_0 \in P$ a kezdőjátékos jelöli, valamint $U = \{l \mid l : S \to S\}$ a játék során alkalmazható lépések halmazát. Egy lépés egy adott s állásban akkor alkalmazható, ha a lépés *előfeltételeinek* az állás megfelel. Például, a Sakkban egy bástyával nem lehet átlósan lépni. Jelölje egy lépés előfeltételének értékét *előfeltétel(l,s)*, ahol l a vizsgált lépés és s azon állás, melyre a lépést vizsgáljuk. Minden véges játék, a szabályai alapján meghatározva, valamely állásokban véget ér, így jelölje egy állás végállás tesztjét *végállás(s).* A végállásokban, a játékszabályokban leírtak szerint valamely játékos veszít, és valamely játékos nyer, ezért jelölje $nyer : \{s \mid végállás(s)\} \to P$ azt a függvényt, amely meghatározza a nyertes játékost. Ezek alapján egy játék reprezentációja felírható egy $< A, a_k, V, O >$ rendezett elem négyes segítségével ahol:

- $A = \{(s, p) \mid s \in S, p \in P\}$ az *állapotok halmaza*, ahol p az s álláson lépni következő játékos.
- $a_k \in A$ és $a_k = (s_0, p_0)$ a *kezdőállapot*
- $V = \{(s, p) \mid végállás(s), p \in P\}$ a *végállapotok halmaza*, ahol a soron következő p a nyertes játékos, ha $p = nyer(s)$
- $O = \{o \mid o(s, p) = (l(s), p'), p \in P, p' \in P, p \neq p'\}$ az *operátorok halmaza*. [10]

Tehát ha a játékfát tekintjük, akkor a fa gyökércsomópontja a_k kezdőállapotot, a fa terminális csúcsai a V halmaz elemit, valamint a köztes csomópontok, ha a kezdőállapotot és a végállapotokat nem számítjuk, A halmaz elmeit tartalmazzák. A csomópontokat összekötő élek pedig O halmaz egy-egy elemét reprezentálják. Egy játszma az a_k kezdőállapotból kiinduló, olyan $o_1, ..., o_n$ operátorsorozat, ahol $n \geq 1$,és amely a fa valamelyik terminális csomópontjába vezet. Egy játékos *stratégiája* egy olyan $D_p = \{(s, p) \mid (s, p) \in A\} \to O$ döntési terv, amely a játékos számára megadja, hogy a játék során előálló azon állásokban, melyekben a játékos következik lépni, melyik lépés a célravezető. Amennyiben az egyik játékos D_A, és a másik játékos pedig D_B stratégia szerint játszik, akkor a két stratégia egyértelműen meghatároz egy játszmát. Egy játékos akkor mondhatja el, hogy létezik *nyerő stratégiája*, ha az ellenfél összes stratégiája esetén nyerni tud. Minden játék esetén elmondható, hogy a két játékos közül

valamelyik számára létezik nyerő stratégia, ha a döntetlen a játékban nem megengedet, illetve ha döntetlen is előfordulhat, akkor legalább nem vesztő stratégiája.

1.3.ÉS/VAGY gráfok játékokkal kapcsolatos alapfogalmai

Esetünkben egy gráf egy adott problémát és annak részproblémákra való bontását hivatott szemléltetni. *Gráfnak* nevezünk egy olyan $< N, E >$ rendezett elem párt, ahol:

- N a gráf csúcsait tartalmazó nem üres halmaz,
- $E \subseteq \{N \times N\}$ a csúcsokat összekötő élek halmaza.

Egy gráfot *végesnek* mondunk, ha a csúcsok halmaza véges. *Irányított* egy gráf, ha minden éléhez tartozik egy irány, amely pontosítja az éleken keresztül történő, csúcsok közötti mozgás módját. Vagyis egy irányítatlan gráf $n_1, n_2 \in N$ csúcsait összekötő (n_1, n_2) és (n_2, n_1) élek, ugyanazt az élt jelölik, viszont egy irányított gráf esetében már két különböző élt jelentenek. Egy irányított vagy irányítatlan gráfon belül *útnak* nevezzük azon élek $(n_1, n_2), (n_2, n_3), ..., (n_{i-1}, n_i)$ sorozatát, ahol:

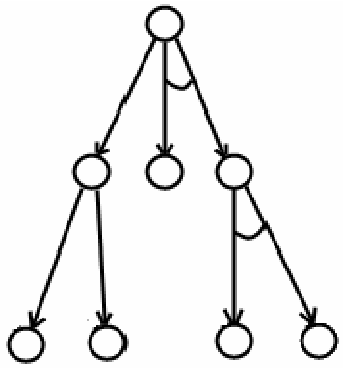
$$n_1, n_2, n_3, ..., n_{i-1}, n_i \in N \text{ és } n_1 \neq n_2 \neq n_3 \neq ... \neq n_{i-1} \neq n_i, \text{ ahol } i \leq |N|.$$

Egy él kiinduló csúcsát *szülőnek*, célcsúcsát *utódnak*, valamint az egy szülőtől származó utódokat *testvéreknek* nevezzük. Minden játék szemeltethető egy olyan irányított gráffal, melynek csúcshalmazának elemei a játék során előforduló összes lehetséges állás, és élhalmazának elemei a játék során megléphető összes lehetséges lépés. Értelemszerűen mivel véges játékokról beszélünk, az őket modellező gráfok is végesek, és ez a jellemző fenn áll a továbbiakban tárgyalt gráfelméleti fogalmakra is. Informatikai tekintetben a gráfok kezelése időigényes, és nehézkes, ezrét a gráfot fává alakítjuk úgy, hogy azokat a csúcsokat melyekhez a gráfban több út is vezet, többször is felvesszük a fába. *Fának* nevezzük gráfelméletben azt a gráfot, amelynek bármely két csúcsát pontosan egy út köti össze. A fa azon csúcsát, melybe nem vezet él *gyökérnek*, illetve azon csúcsokat, melyekből nem vezet ki él, *leveleknek* vagy *terminális csúcsoknak* hívjuk. A fa szintjeit tekintve a gyökércsúcs a 0. szinten helyezkedik el, illetve a fában minden páros szinten azon állások találhatóak, melyek esetén az egyik játékos, és minden páratlan szint esetén azon állások, melyekben a páros szinthez tartozó játékos ellenfele következik lépni. A stratégiák és a nyerőstratégia meghatározásához viszont, mivel mindig egy játékos szemszögéből vizsgáljuk, az

előbbi módon megkapott fát ÉS/VAGY fává kell alakítani. *ÉS/VAGY gráf* esetében egy olyan $< N, HE >$ rendezett elem párról beszélünk ahol:

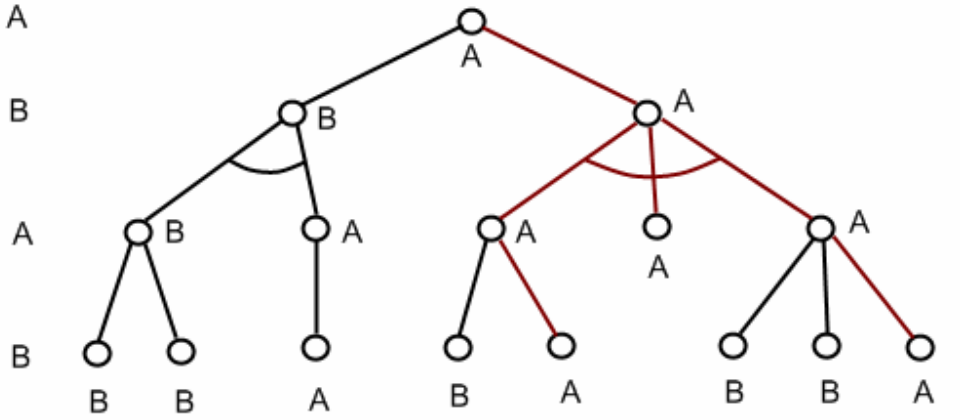
- N a gráf csúcsait tartalmazó nem üres halmaz,
- $HE \subseteq \{(n, M) \in N \times 2^N\}$ a gráf hiperéleinek halmaza, ahol M a gráf csúcspontjainak egy véges nem üres halmaza. [10]

ÉS/VAGY gráfoknál azt mondjuk, hogy két testvér csomópont *ÉS kapcsolatban* van, ha ugyanazon hiperélen keresztül érhetőek el a szülőcsomópontból, illetve *VAGY kapcsolatban* van, ha 2 különböző hiperélen keresztül. Tehát azon hiperéleket, melyek esetén $|M| = 1$ *VAGY éleknek*, illetve azon hiperéleket, melyek estén $|M| > 1$ *ÉS éleknek* nevezzük. Az 1.3.1 ábrán az ívekkel összekötött élek alkotnak ÉS éleket. *Hiperútról* beszélünk egy olyan hiperél-sorozat esetén, amely sorozatban minden a hiperélek által érintett csúcs különbözik, illetve minden hiperél szülő csúcsa, kivéve a sorozat első elemét, szerepel a sorozatban őt megelőző hiperélek M utódhalmazainak valamelyikében. *ÉS/VAGY fáról* értelemszerűen akkor beszélünk, ha egy ÉS/VAGY gráf minden csomópontjába pontosan egy hiperút vezet. Egy játékfából, egy játékos szemszögéből, úgy kapunk ÉS/VAGY fát, hogy azon állásoknál, ahol a vizsgált játékos következik lépni, a lépések egy-egy VAGY éleknek, illetve azon állásoknál, ahol a vizsgált játékos ellenfele következik lépni, az összes lehetséges lépés egy ÉS élnek felel meg. Egyértelmű hogy egy játékfát kétféleképpen is át lehet alakítani ÉS/VAGY fává, attól függően, hogy melyik játékos szemszögéből vizsgáljuk a stratégiákat. Egy játékos szemszögéből az összes stratégia szemléltethető olyan hiperutakkal, melyek a fa gyökércsúcsából indulva a fa levélcsúcsaiban végződnek. Ezen stratégiák közül nyerő stratéga az a hiperút, melynek összes levélcsúcsa a játékos számára nyerő állást tartalmaz. Ilyen hiperút minden játékfa esetén létezik, valamelyik játékos számára. Ennek bizonyítása képen egy tetszőleges játékfa esetén címkézzük fel a levélcsúcsok véletlenszerűn *A*, vagy *B* címkékkel a szerint, hogy a végállás melyik játékos számára nyerő állás. A levélcsúcsoktól felfele haladva a szülőcsúcsok úgy kapnak címkét, hogyha a szülőcsúcs állásánál *A* következik lépni, és van olyan utóda, amely már rendelkezik *A*



1.3.1 ábra: ÉS, VAGY élek

címkével, akkor a szülőcsúcs is *A* címkét kap. Ellenkező estben pedig a szülőcsúcs *B* címkét kap. Ugyanez a címkézés menete azon szülőcsúcs esetében is ahol *B* következik lépni, azzal a különbséggel, hogy a *B* játékos szemszögéből kell vizsgálni az utódok címkéjét. Mivel a fa véges, végül a gyökércsúcs is kap címkét, amely megadja, hogy a játékban mely játékosnak van nyerőstratégiája. A 1.3.2 ábrán látható pirossal jelzett hiperút *A* játékos nyerőstratégiáját mutatja. Abban az esetben, ha a döntetlen a játékban megengedett, annak bizonyítására, hogy valamely játékosnak van legalább nem vesztő stratégiája, alkalmazható az előbbi módszer minimális módosításokkal. A különbség az, hogy egy szülőcsomópont akkor kap *D* címkét, amely a döntetlen állást jelöli, ha van legalább egy olyan utóda, amely *D* címkével rendelkezik és minden más utóda, a szülőcsomópontban lépni következő játékos, ellenfelének címkéjét viseli, illetve ha minden utóda döntetlen címkével rendelkezik.



1.3.2 ábra: Nyerő stratégia létezésének bizonyítása

1.4. Játékelméleti alapfogalmak

A kétszemélyes zérusösszegű játékokkal már olyan matematikusok is foglalkoztak, mint Emile Borel, Neumann János és John Forbes Nash, illetve a magyar származású közgazdász Harsányi János. Nem utolsó sorban Neumann János volt az, aki 1928-ban lefektette a játékelmélet alapjait. Majd 1944-ben Oskar Morgensternnel közösen megírták a *Theory of Games and Economic Behavior* című könyvet, amelyet az első igazi játékelméleti munkának tekintenek. Az olyan játékokat, mint a sakk, vagy dáma a játékelmélet *extenzív formában adott játékoknak* nevezi. Ezen játékok mindig felírhatóak normál formában. Legyen egy játék *normál formában*

való felírása $G = \{X, Y; f\}$, ahol az *X*, és *Y* a játékosok a játék során megjátszható stratégiáinak halmazai, valamint mivel két stratégia egyértelműen meghatároz egy játszmát, ezért $f : X \times Y \to \Re$ a játszma végállásának „hasznosság" értékét meghatározó függvény az első játékos szemszögéből. Elegendő csak az egyik játékosra megadni a „hasznosság" függvényt, ugyanis a másik játékos „hasznosság" értékei, az első játékos „hasznosság" értékeinek negatívumai lesznek a játék zérusösszegű tulajdonsága miatt. *Nash-egyensúlypontnak* akkor és csak akkor nevezzük azt az (x^*, y^*) stratégiai párost, ha $f(x^*, y^*) \geq f(x, y^*)$ és $f(x^*, y^*) \leq f(x^*, y)$ minden $x \in X$ és $y \in Y$ esetén teljesül. Ezen stratégia páros olyan, hogyha bármely játékos eltér az x^*, vagy y^* stratégától úgy, hogy az ellenfél nem vált stratégiát, akkor a játék folyamán már csak rosszabb, vagy legjobb esetben ugyanolyan eredményt érhet el. A játékelméletben egy másik fontos tulajdonsága az olyan játékoknak, mint a dáma, hogy minden játékos *tökéletesen informált*. Vagyis mindkét játékos ismeri a játékot leíró fát, valamint tisztában van azzal, hogy éppen melyik állásnál tart a játék, és azzal a lépéssorozattal, amely elvezetett az aktuális álláshoz. Ebből következik, hogy mindkét játékos információs halmazai rendre egyetlen elemet tartalmaznak. Az *i* játékos egy *információs halmaza* alatt egy olyan U_i halmaz U_i^t részhalmazát értjük ahol:

- U_i a játékfa azon csúcsainak halmaza ahol az *i* játékos következik lépni,
- U_i^t minden csúcsából ugyanannyi él indul ki, és az élek az *i* játékoshoz tartozó csúcsok felé irányulnak,
- A fa bármely útjának legfeljebb egy közös csúcsa van U_i^t-vel,
- Az U_i^t halmazok egy partícióját adják U_i halmaznak. [11]

Mivel a játékban szereplő mindkét játékos tökéletesen informált, ezért maga a játék is *tökéletes információs játék*, ugyanis a játékban minden információs halmaz egyetlen csúcsot tartalmaz. Egy extenzív formában adott kétszemélyes játék *részjátéka* alatt egy olyan részjátékot értünk, amely egy egyelemű információs halmazt tartalmaz a gyökércsúcsában, és részfája az eredeti játékfának. Mivel esetünkben minden információs halmaz egy elemű, ezért az eredeti játék összes részfája egy-egy részjátéknak felel meg. Egy játék egy egyensúlypontját *részjáték tökéletesnek* nevezzük, ha azt egy részjátékra korlátozva továbbra is egyensúlypont marad. *Kuhn tétele*

kimondja a játékelméletben, hogy minden véges tökéletes információs játéknak van részjáték tökéletes egyensúlypontja [11]. A Nash-egyensúlypontban vett „hasznosság” értéket szokás a *játék értékének*, valamint az egyensúlypont által meghatározott stratégiákat *optimális stratégiáknak* nevezni. Egy *G* játéknak akkor és csak akkor van Nash-egyensúlypontja, ha:

$$\max_{x\in X}\inf_{y\in Y} f(x,y) = \min_{y\in Y}\sup_{x\in X} f(x,y)$$

Mivel csak véges játékokkal foglakozunk, ezért X, Y halmazok is végesek, és legyenek $|X| = m$ és $|Y| = n$. Ekkor a játék megadható egy $m \times n$ dimenziójú *A* mátrixban, melynek a_{ij} eleme visszaadja az *i* és *j* stratégiák által adott játszma „hasznosság” értékét az első játékos számára, ha az első játékos *i*, illetve a második játékos *j* stratégia szerint játszik. Jelölje xAy az *A* mátrixban az *x* és *y* stratégákhoz tartozó sor és oszlop által meghatározott „hasznosság” értéket. Így tehát a fenti egyenlet megadható a következő formában:

$$\max_{x\in X}\min_{y\in Y} xAy = \min_{y\in Y}\max_{x\in X} xAy$$

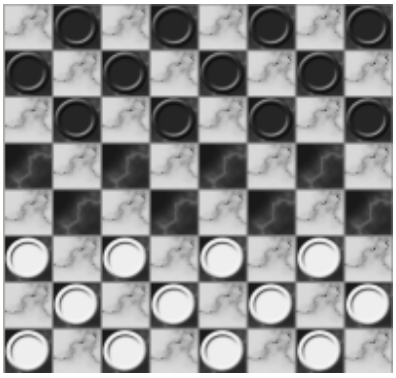
Az így kapott egyenletnek a kétszemélyes zérusösszegű játékokra való általánosabb megfogalmazását nevezzük Neumann János közismert *minimax tételének*.

A\B	y_1	y_2	y_3	y_4	
x_1	6	8	6	8	6
x_2	6	2	6	2	2
x_3	-10	8	-10	8	-10
x_4	-10	2	-10	2	-10
x_5	7	7	11	11	7
x	1	1	11	11	1
x	7	7	-8	-8	-8
x	1	1	-8	-8	-8
	7	8	11	11	

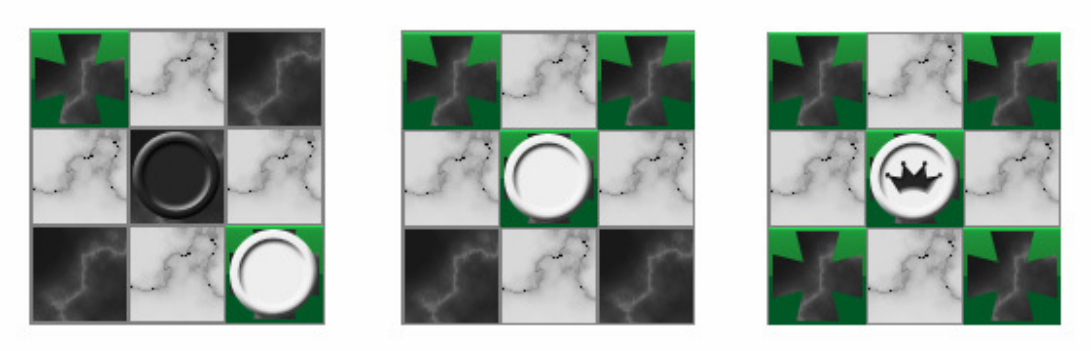
1.4.1 táblázat: Nash-egyensúlypont meghatározása

1.5.Dáma szabályok

A dámajáték Egyiptomból származik és az ókori Görögökön és Rómaiakon keresztül jutott el Európába, Ázsiába és Afrikába. Európában igen nagy népszerűségre tett szert, olyannyira hogy a játék hozzátartozott a „hét lovagi erényhez". Mára már a dáma által bejárt hosszú útnak köszönhetően több mint tíz válfaja létezik a játéknak és azoknak is több szabályrendszere. Az általam választott dáma, az Angol Dáma játékszabályait követi. A játékot már a sakkból is jól ismert négyzetrácsos fekete fehér táblán, illetve a sakkhoz hasonló módon fekete és fehér bábukkal játsszák. A játékban két figurát különböztetünk meg, a *gyalogot* és a *dámát*. Ahhoz hogy a két figurát elkülönítsék egymástól, a dámát valamilyen módon jelölni kell. Általános az a módszer, hogy egy azonos színű gyalogot helyeznek a dámává vált gyalog tetejére, vagy olyan készletet használnak, ahol a dáma már jelölve van, például egy koronával. Mindkét játékos 12-12 gyaloggal kezdi a játékot, amelyeket a tábla azonos színű mezőire állítanak fel úgy, hogy a tábla 4. és 5. sora üres marad, ahogy az 1.5.1 ábrán is látható. A szabályok szerint a sakkal ellentétben mindig a fekete játékos kezd. A játék célja hogy az ellenfél összes bábuját eltávolítsuk a tábláról, vagy olyan állásba kényszerítsük, ahol már nincs több lépése. A lépéseket tekintve



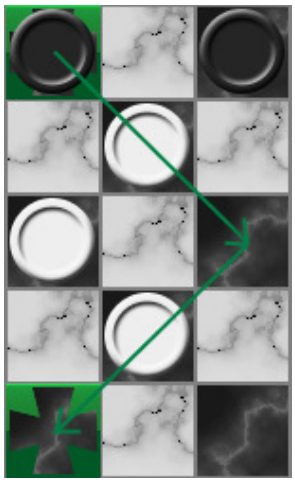
1.5.1 ábra A dámajáték kezdőállása



1.5.2 ábra: Szabályos ütés-, gyalog- és dámalépések.

minden gyalog csak átlósan léphet és kizárólag egyet az ellenfél irányába. Nyilvánvaló, hogy a játék során, ezáltal csak a tábla egyszínű négyzeteit használják a játékosok. Amennyiben egy

gyalog eléri a számára legtávolabbi sorát a táblának, dámává alakul, de csak a lépés elvégzése után, melynek később lesz jelentőssége. A dáma a gyaloggal ellentétben rendelkezik azzal a helyzeti előnnyel, hogy négy irányban léphet átlósan egyet. Az ellenfél bábuját a tábláról ütéssel lehet eltávolítani. *Ütéshelyzetnek* nevezzük a dámában azt a helyzetet, amikor két ellentétes színű figura átlósan, valamelyik figura haladási irányának megfelelően szomszédos, és velük egy vonalban a haladási iránynak megfelelően létezik üres hely. Azt a lépést, amikor egy bábu az ellenfél bábuját átlépve átkerül az előbb említett üres helyre, *ütésnek* nevezzük, és az átugrott bábut el kell távolítani a tábláról. Nyilvánvaló, hogy a dámának az ütések tekintetében is előnye van a gyalogokkal szemben, hiszen a lépések mintájára ugyancsak négy irányban üthet. A játék során *ütéskényszer* van. Tehát ha egy bábu ütéshelyzetbe kerül, akkor a játékosnak kötelező az ütést meglépnie. Abban az esetben, ha ugyanazon bábu, az ütés után ismét ütéshelyzetbe kerül, az ütéskényszer miatt az ütést folytatnia kell. Ebből kifolyólag *ütéssorozatról* beszélünk az így egymásután kapcsolt ütések esetén. Ütéssorozat mindig csak egy bábu szemszögéből létezik, tehát ha egy adott színű bábuval elvégzett ütés után egy megegyező színű másik bábunak is ütéshelyzete van, az nem számít ütéssorozatnak. Minden ütéssorozat egy lépésnek minősül. Abban az esetben, ha egy gyalog ütéssorozattal éri el a tábla legtávolabbi sorát, és lehetősége lenne dámaként folytatnia az ütéssorozatot, akkor sem folytathatja, mivel a gyalog csak a lépés elvégzése után válhat dámává, és gyalogként a haladási iránnyal ellentétesen nem üthet. Amennyiben egy bábunak több ütéshelyzete is van, a játékosnak kötelező azt az ütést, vagy ütéssorozatot választania, amely több bábut távolít el a tábláról. Amennyiben megegyező számú bábu eltávolítására van lehetősége, akkor tetszőlegesen választhat a két irány közül. Ütéssorozat esetén csak akkor kerülnek le a leütött bábuk a tábláról, ha az ütéssorozat befejeződött. Tehát például, egy dáma az ütéssorozatot nem folytathatja, ha az ütéssorozatban olyan helyre lépne, amelyen egy már leütött bábu tartózkodik. Minden bábut legfeljebb egyszer lehet leütni, és saját bábu leütése szabálytalan. Ugyancsak nem megengedett, egy bábu ütés nélküli átugrása. Ahhoz hogy a játékmenetet lerövidítsem az esetleges olyan helyzetek kialakulása esetén, amikor tartósan egyensúlyi helyzet alakul ki a két fél között, a szabályok közzé felvettem egy sakkban jól ismert



1.5.3 ábra: Ütéssorozat

szabály dámára alakított változatát. Az *50-lépés szabály* kimondja a sakkban, hogyha a játszma 50 lépése során, sem ütés, sem gyaloglépés nem történik, és bármelyik játékos igényli, akkor a játszma döntetlen. Dámára a szabályt úgy módosítottam, hogy abban az esetben, ha 50 lépésen keresztül, sem dámává alakítás, sem ütés nem történik akkor a játszma döntetlen. A dáma számítógépes megközelítést tekintve a játékfája által tartalmazható összes állás száma $5*10^{20}$. Viszont ez a szám egy kicsit csalóka lehet, ugyanis így a fa olyan állásokat is tartalmaz, amelyek szabályos lépéssorozatokat követve nem érhetőek el a játék folyamán. Vegyük példának csak azon állásokat, amelyek 24 bábut tartalmaznak. A táblán való 24 bábu elhelyezése megközelítőleg $9*10^{19}$ módon lehetséges, ám ezeknek az így előállítható állásoknak csak egy kisebb csoportjába tartozó állások azok, amelyek egy szabályos játék keretei között előfordulhatnak. Ugyanis elhelyezhetünk 24 bábut úgy is a táblán, hogy az mind dáma. Belátható, hogy nincs olyan szabályos lépéssorozat, amely elvezetne egy ilyen álláshoz. Azon állások száma tehát, amelyek szerepelhetnek egy ésszerű játék folyamán az összes állást figyelembe véve, megközelítőleg 10^{18}. [7]

2. Lépésajánló algoritmus

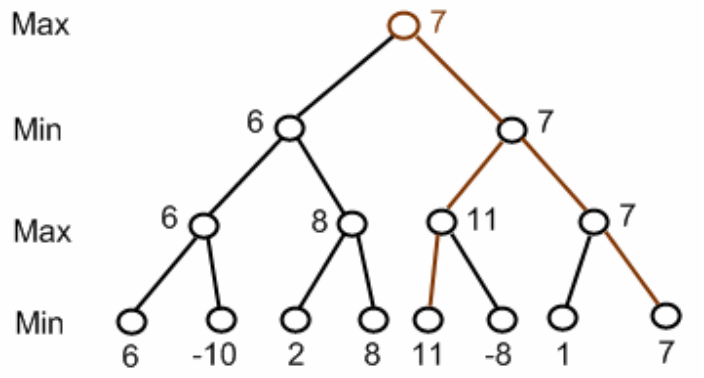
2.1. Minimax algoritmus

A minimax algoritmus az egyik legrégebbi és leggyakrabban alkalmazott lépésajánló algoritmus, valamint alapjául szolgált a későbbi erősebb algoritmusoknak. Az eredeti elképzelés szerint az algoritmus a következő részekből épült fel:

1) A teljes játékfa előállítása a végpontokig.
2) Az összes végállásra alkalmazni a hasznosság függvényt.
3) A végállások hasznosságát felhasználva meghatározni az egy szinttel feljebb lévő állások hasznosságát attól függően, hogy ott az ellenfél vagy a támogatott játékos következik lépni.
4) A 3) lépést alkalmazni egészen addig, amíg el nem éri a fa gyökerét, és ott azt az operátort választani, amely a legnagyobb hasznosság felé vezet.

Az algoritmus az 1.4 fejezetben tárgyalt minimax tételen alapul, vagyis a játék Nash-egyensúlypontja által adott játszma hasznosságértékét határozza meg, és ezen érték alapján ajánl

lépést a támogatott játékosnak. Ebből kifolyólag általánosan vett szokás a játékfa azon szintjeit, ahol a támogatott játékos következik lépni, tehát a páros szinteket, MAX szintnek, illetve ahol az ellenfele következik lépni, tehát a páratlan szinteket, MIN szintnek elnevezni. Értelemszerűen, mivel a fa gyökerében mindig a támogatott játékos a soron következő játékos, ezért a hozzá tartozó szint is MAX szintnek minősül. Jelölje a fa végállapotainak értékét $v(a_k) = f(a_k)$, ahol $f(a)$ a hasznosság függvény. Ha az algoritmus során egy olyan szülőcsomópontnak kell értéket adni, amely MAX szinten helyezkedik el és utódai rendre $a_1,...,a_n$, akkor a szülőcsomópont értéke $v(a) = \max\{v(a_1),...,v(a_n)\}$. Amennyiben a szülőcsomópont a fa MIN szintjén helyezkedik el, akkor értéke $v(a) = \min\{v(a_1),...,v(a_n)\}$ módon alakul. Tehát az algoritmus úgy keresi a legnagyobb hasznosság felé vezető lépést, hogy feltételezi, hogy az ellenfél is a lehető



2.1.1 ábra: A játékfa csúcsértékei a minimax algoritmus végrehajtása után

legjobban fog játszani. Így a támogatott játékosnak ajánlott lépés az optimális lépés lesz, valamint az algoritmus meghatároz egy nyerőstratégiát. A 2.1.1 ábrán egy fiktív játék teljes játékfája látható a minimax algoritmus végrehajtása után. A gyökércsúcsból kiinduló pirossal jelzett él szemlélteti azt az operátort, amely a legjobb lépésnek minősült az algoritmus számára. Ha összevetjük az ábrát a korábban látott 1.4.1 táblázattal, amely e játék mátrixát tartalmazza, látható hogy az érték, amit a gyökércsúcs kapott a Nash-egyensúlypont által meghatározott hasznosság érték. Valamint az is látszik, hogy azon stratégia, amelyet a pirossal jelzett élek által alkotott hiperút határoz meg, egy nyerő stratégiája a támogatott játékos számára. Viszont ahogy a dáma esetében is, úgy a játékok többségénél a játékfák igen nagy méretekkel rendelkeznek, ezért

egyértelmű, hogy az 1) lépés nem kivitelezhető belátható időn belül. A probléma megoldását Claude Elwood Shannon vetette fel egy 1950-ben megjelent sakkprogramokkal foglalkozó tanulmányában. Shannon ötlete az volt, hogy az algoritmus által bejárandó fa méretét csökkenteni lehetne azzal, hogyha az algoritmus a teljes játékfa csak egy részfáját járná be, és a részfa levélelemeire egy úgynevezett heurisztikus kiértékelő függvényt alkalmazna, amely egy hasznosság becslés a támogatott játékos szemszögéből. Felmerül a kérdés, hogy az algoritmus fent említett változtatásai befolyásolják-e az optimális lépésajánlást. A válasz az, hogy a kiértékelő függvény pontosságától függ. Az egyértelmű, hogy az eredeti fa levélelemein kiszámolható értékeket abból kifolyólag, hogy csupán becsléseket lehet alkalmazni a köztes állásokra, pontosan nem lehet előre meghatározni. Viszont minél közelebbi értéket szolgáltat a kiértékelő függvény a levélelemek értékeihez, annál közelebb lesz az algoritmus által ajánlott lépés az optimális lépéshez. Ezek alapján az algoritmus, amit ma minimax algoritmus néven ismerünk a következő lépésekből áll:

1) Valamilyen korlát alapján a bejárandó részfa előállítása,
2) A részfa levélcsúcsaira alkalmazni a heurisztikus kiértékelő függvényt,
3) A heurisztikus értékeket felhasználva meghatározni az egy szinttel feljebb található szülők értékét aszerint, hogy azok MIN vagy MAX szinten találhatóak,
4) A 3) lépést alkalmazni egészen addig, amíg el nem éri a gyökércsúcsot és ott MAX módon értéket adni. Ezután meghatározni azt az operátort, amely a legnagyobb heurisztikus érték felé vezet.

A részfákra való leszűkítés felveti azt a problémát, hogy az algoritmus már nem határozza meg a teljes nyerő stratégiát, amit a támogatott játékos követhetne. Egy ésszerű megoldásának tűnhet, hogy a részfa azon levélcsúcsában, amelytől a heurisztikus értéket örökölte a gyökércsúcs, az algoritmust újra el kell végezni azon részfára, amelyben már a korábbi részfa levélcsúcsa szerepel, mint gyökércsúcs. Viszont az algoritmus bár feltételezi, hogy az ellenfél a lehető legjobban játszik, arra nincs garancia, hogy ténylegesen ez is történik a játszma folyamán. Előfordulhat például, hogy az algoritmus egy másik lépésajánló algoritmus ellen játszik és más heurisztikát használ, vagy ha emberi ellenfél ellen játszik, akkor az egyszerűn hibázik, és az ellenfél lépése más lesz, mint amit neki optimálisnak véltünk. Tehát nem elegendő a bejárt részfa levélelemeinél újra végrehajtani az algoritmust. Ahhoz, hogy a játék folyamán a támogatott

játékos számara, a megfelelő lépéseket tudja az algoritmus ajánlani, az ellenfél minden lépése után újra végre kell hajtani a lépésajánlást. Azért, hogy az algoritmus időt takarítson meg, a fa előállítását mélységi keresés alapján végzi, így nem kell a fát többször bejárni, mivel lehetőség van arra, hogy a 2), illetve 3) lépéseket már a fa előállításakor alkalmazza. A részfa korlátját tekintve, alkalmazhatunk idő-, valamilyen a játékból szerzett plusz információn alapuló, illetve fix mélységi korlátot. Általánosan a fix mélységi kotlát a jellemző. Ha a részfa mélysége *m* és minden szülőcsomópontnak *b* darabszámú utóda van, akkor az algoritmus időigénye $O(b^m)$, és tárigénye lineáris m-ben és b-ben. Az algoritmus hátránya, hogy az egész részfát bejárja, megvizsgálva azon állásokat is, amelyeknek nincs hatásuk a játék kimenetelére, és ez nagyobb játékok esetén időigényes, mivel a részfák már kis mélységekben is, az állásokon alkalmazható operátorok száma miatt, meglehetősen sok állást tartalmaznak. Ezért számítógépes játékok esetében tisztán csak nagyon ritkán, és kisméretű játékok esetében jelenik meg ez az algoritmus, viszont a játékok matematikai elemzésére még ma is használják.

2.2. Heurisztikus kiértékelő függvény

Mint már az előző fejezetben is tárgyaltam, ahhoz hogy a minimax algoritmus belátható időn belül befejezze futását, a teljes játékfát korlátozni kell részfákra, és a részfák levélelemeiről valamilyen módon el kell dönteni, hogy az a támogatott játékos szemszögéből mennyire hasznos. Ehhez az kell, hogy a program tisztában legyen a játék olyan jellemzőivel, melyek ezt a hasznosságot meghatározzák. Ha sakkot vesszük példának, akkor jellemzők lehetnek, a bábuk fajtánkénti darabszámai, illetve az hogy mekkora esély van arra, hogy a király sakkba kerüljön. A játék e jellemzőit együttesen *heurisztikának* nevezzük. Fontos hangsúlyozni, hogy a heurisztika csupán egy becslés arra, hogy az állás mennyire hasznos a támogatott játékos számára. A bonyolultabb játékok esetében, mint a sakk és a dáma, ezeknek a jellemzők a meghatározása már nem olyan egyszerű feladat. Ha a heurisztika nem a megfelelő jellemzőket felhasználva becsül értéket az adott állásokra, annak az a következménye, hogy a program nem játszik majd hatékonyan, és hibákat követ el, ugyanis a minimax algoritmus által ajánlott lépés nem fogja megközelíteni az optimális lépést. Az nyilvánvaló, hogy a tényleges végállapot értékek meghatározás nem lehetséges, ám törekedni kell arra, hogy a heurisztika minél pontosabban megközelítse ezen értékeket az előző fejezetben tárgyalt okok miatt. A heurisztika által

tartalmazott jellemzők felírhatóak egy olyan függvény formájában, melyet a részfa levélelemeire alkalmazva meghatározhatjuk egy számértékben az állapotok hasznosságát. Ezt a függvényt nevezzük *heurisztikus*, *vagy statikus kiértékelő függvénynek*. Általánosan bevett szokás, hogy azon állapotokra melyek a támogatott játékos számára előnyösek pozitív, azon állapotokra melyek az ellenfél számára előnyösek negatív, és a döntetlen, vagy ahhoz közeli állapotokra pedig nullához közeli értékeket határoz meg a kiértékelő függvény. Az sem mindegy a heurisztika terén, hogy az egyes jellemzőknek mekkora befolyása van a játék kimenetelére. Például a sakkban nem mindegy, hogy a támogatott játékos egy gyalogját, vagy a királynőjét ütötte le az ellenfél az előző lépésben. Vagyis e jellemzőket valamilyen módon súlyozni kell. Ezért általában a kiértékelő függvényeket szokás:

$$f(a) = \sum_i w_i h_i$$

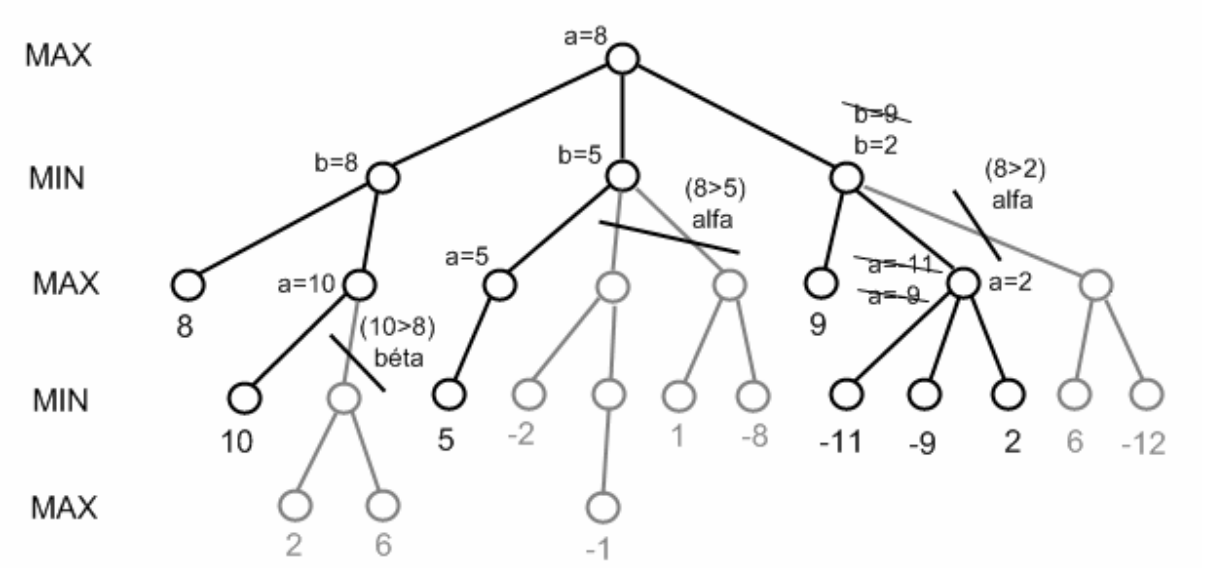
súlyozott lineáris formában megadni, ahol h_i a heurisztika i jellemzője által meghatározott számérték, és w_i az i jellemzőhöz tartozó súly. Már magukat a jellemzőket is elég nehéz megtalálni, de a hozzájuk tartozó megfelelő súlyok meghatározására és optimalizálására már csak gyakorlatban van lehetőség. Először Arthur Samuel alkalmazta öntanuló dáma programjánál azt a módszert, hogy a program két másolatát egymás ellen játszatta, hogy egymástól tanuljanak. A módszer a súlyok meghatározásánál sem különbözik. A lényeg, hogy a programot hagyni kell játszani önmagával és figyelni a lejátszott játszmák győzelem-vereség arányát. Ezen arány segít eldönteni, hogy két heurisztika közül melyik a hatékonyabb, illetve azonos heurisztika esetén milyen súlyok a célravezetőbbek.

3. Hatékonyság növelő módszerek

3.1. $\alpha\beta$ vágás

A minimax algoritmus legnagyobb hátránya, hogy a teljes részfát bejárja, és olyan állapotokat is kiértékel, amelyeket nem lenne szükséges. Az $\alpha\beta$ vágás egy olyan optimalizációs technika a minimax algoritmushoz, amely felismeri a bejárás közben szerzett információk segítségével, hogy mely csúcsok azok, melyeken a fa előállítása során, a legjobb lépés meghatározása érdekében, érdemes továbbmenni. Így az $\alpha\beta$ vágással kibővített minimax

algoritmus képes gazdálkodni a lépésajánlásra fordítható erőforrásokkal. A módszert azért hívják *vágásnak*, mert azon ágait a részfának, melyek biztosan nem lesznek hatással a lépésajánlásra, egyszerűen elhagyja, más néven levágja. Az $\alpha\beta$ vágás azért közkedvelt, mert az eredeti minimax algoritmusban csupán a 3) és 4) lépéseket kell módosítani a vágás megvalósításához. A csúcsok heurisztika értékét nevezzük el α , és β értékeknek, melyekről a vágás a nevét kapta. α, és β értékeket attól függően, hogy MIN, vagy MAX szinten szerepel, minden csúcs kap leszámítva a levélcsúcsokat, ugyanis azokra, ahogy a minimax algoritmusnál is láttuk, a heurisztikus kiértékelő függvényt alkalmazzuk. Az α érték egy MAX szinten szereplő csúcs esetén az alatta eddig bejárt részfa legnagyobb heurisztika értékét tartalmazza, és az alatta meghatározandó további heurisztika értékekre egy alsó korlátot határoz meg. A β érték értelemszerűen egy MIN szinten szereplő csúcs eddig alatta kiértékelt legkisebb heurisztika értékét adja, és felső korlátot jelent a csúcs alatt felmerülő további heurisztika értékekre. Minden csúcs akkor kap α, vagy β



3.1.1 ábra: A vágások ,és az α, β értékek alakulása az $\alpha\beta$ vágás végrehajtásakor

értéket, ha már van legalább egy olyan utóda, amely már rendelkezik értékkel. A vágást tekintve egy MIN csúcs alatti kiértékelés tovább már felesleges, ha a hozzá vezető lépéssorozat által meghatározott úton a részfában létezik legalább egy olyan MAX őse, melynek α értéke nagyobb vagy egyenlő, mint a MIN csúcs β értéke. Ezt a vágást nevezzük α *vágásnak*. Tehát arról van szó, hogyha már korábban a részfa bejárása közben találtunk valahol egy olyan lépést, amely a

MIN csúcs MAX őse számára jobb értékkel rendelkezik, mint a MIN szinten felmerülő érték, amely már biztos, hogy nem lesz nagyobb, akkor elmondható, hogy MAX azt az utat nem fogja választani, amely a MIN csúcshoz vezet. Egyértelmű, hogy egy MAX csúcs alatti kiértékelés tovább már felesleges, ha a hozzá vezető lépéssorozat által meghatározott úton a részfában létezik olyan MIN őse, amely β értéke kisebb vagy egyenlő, mint a MAX csúcs α értéke. Ezt β *vágásnak* nevezzük. Tehát röviden összefoglalva a vágás feltétele $\alpha \geq \beta$. Az $\alpha\beta$ vágással így módosított minimax algoritmus az eredeti algoritmussal egyenértékű, esetenként megegyező lépést határoz meg. A vágással megvalósított minimax algoritmus hatékonyságát tekintve akkor lenne a leghatékonyabb, ha először azokat a csúcsokat értékelné ki, melyek a legnagyobb értékekkel rendelkeznek. Ez sajnos nem lehetséges mivel a részfa a bejárása közben jön létre, ezért a rendezéshez szükséges értékeket nem ismerjük előre. Ha feltételezzük, hogy ez mégis megtehető, akkor az algoritmus $O(b^{m/2})$ időigénnyel rendelkezik. Ez jóval kevesebb, mint a tiszta minimax esetében felmerülő időigény. Tehát a vágással mélyebb vizsgálatot lehet végezni, mélyebb vizsgálat esetén több állást érint az algoritmus, ezért több információja lesz, hogy a megfelelő lépést meghatározza. Egy mélyebb keresés nem csak a lépésajánláshoz felhasználható többlet információ előnyével rendelkezik, hanem azzal az előnnyel is, hogy egy jól működő heurisztika minél közelebb van az eredeti fában a levélcsúcsokhoz, annál pontosabb becslést ad a támogatott játékos nyerési esélyeire. 1975-ben az algoritmust mélyebb vizsgálat alá vetették, ahol kiderült, hogy aszimptotikus komplexitása $O((b/\log b)^m)$, ami már nem olyan jó eredmény. Viszont a keresési problémák komplexitásának meghatározásához ideális famodellt alkalmaznak. Tehát olyan fákra vizsgálják az algoritmust, amelyekben minden csúcs *b* utóddal rendelkezik, az összes út eléri a mélységi korlátot, valamint a fa levélcsúcsai véletlenszerűen oszlanak el a fa legutolsó rétegeiben. Belátható, hogy egy játék fája csak igen ritkán rendelkezik az összes fent említett jellemzővel. Összegezve az algoritmus időbonyolultsága erősen függ a kiértékelt utódok sorrendjétől, vagyis a gyakorlatban $O(b^{m/2})$ és $O(b^m)$ időigények közé esik minden lépésajánlás időigénye. A gyakorlatban előforduló játékok fáira az is igaz, hogy a csomópontok értékei erősen korreláltak a rokoncsomópontjaik értékeivel, mely korreláció erősen függ a játéktól, valamint a gyökérben elhelyezkedő állapottól. Elméletben azt is igazolták, hogy attól függetlenül, hogy mekkora a fa elágazási faktora, átlagban egy adott csúcspont első két utódja

kiértékelése után levágás következik. Tehát elmondható, hogy az esetek többségében az $\alpha\beta$ vágással megvalósított minimax algoritmus jobb hatékonysággal rendelkezik mint a tiszta minimax algoritmus.

3.2. Cáfoló lépés elve

Az $\alpha\beta$ vágás legnagyobb hátránya, hogy a csúcsok kiértékelési sorrendjének függvényében erősen ingadozik a hatékonysága. A vágás hibájának kiküszöbölésére találni kell egy olyan módszert, amely segítségével a csúcsok rendezhetőek a heurisztikájuk alapján. A probléma, amely már az előző fejezetben is felmerült, hogy a rendezést úgy kellene elvégezni, hogy nem ismerjük előre az összes felmerülő heurisztika értéket. Így csak az eddig bejárt részfa kiértékelései közben felmerült értékekre lehet hagyatkozni. Azon hatékonyság növelő technikákat, amelyek ilyen módon megkísérlik rendezni a részfa csúcsait a bejárás alatt, *előrendezéseknek* nevezzük. Ilyen előrendezés a cáfoló lépés elve is. Az elv lényege, hogyha meghatároztuk a bejárás egy pontján, hogy egy adott szinten melyik lépés volt az, amelyik eddig a legtöbb vágáshoz vezetett, akkor azon a szinten egy másik állásnál is nagy valószínűséggel vágáshoz vezet. Ez azért lehetséges, mert az azonos szinten elhelyezkedő állások csupán egyetlen lépésben különböznek, ami nem elég nagy eltérés ahhoz, hogy a lényegesebb jellemzőikben is eltérjenek. Tehát bejárás közben minden új csúcsnál, mielőtt az összes alkalmazható operátort megvizsgálná az algoritmus, megnézi hogy van-e a szinthez cáfoló lépés, és ha van, akkor az alkalmazható-e az aktuális állapotra. Ha igen akkor a következő csúcsot a cáfoló lépéssel terjeszti ki először abban a reményben, hogy a cáfoló lépés vágáshoz vezet. A további operátorokat tehát csak akkor kell alkalmazni, ha a cáfoló lépés nem vezet vágáshoz. Általánosan bevett szokás a megvalósításuk tekintetébe, hogy nem egy, hanem két cáfoló lépést is nyilvántartanak, így csökkentve annak az esélyét, hogy ezek a lépések nem minősülnek szabályos lépésnek egy adott állásra nézve.

3.3. History heuristic

A cáfoló lépés hátránya, hogy minden mélységhez csak egy vagy két legjobb lépést tart nyilván és azokat alkalmazza, ha az aktuális állapoton alkalmazhatóak. A probléma ezzel az, hogy meg van az esély arra, hogy egy szinten több olyan állás is előfordulhat, amelyre nem

alkalmazható egyik cáfoló lépés sem. Az is gondot jelenthet a cáfoló lépés elvének alkalmazásakor, hogyha a cáfoló lépések nem vezetnek vágáshoz. Ugyanis akkor a fennmaradó operátorokkal kapcsolatban az algoritmus számára azon a szinten már semmilyen információ nem áll rendelkezésre, ezért a további lépésekre már nem valósíthat meg előrendezést. Ezen következmények egyértelműen hatékonyság csökkenéshez vezetnek. A history heuristic a cáfoló lépés általánosított változata. Nem csak egy-két kitüntetett lépésről tárol információt mélység specifikusan, hanem az összes lehetséges lépésről és az egész játékfára vonatkozóan. A history heuristic esetében alkalmazott elv hasonló a cáfoló lépés elvéhez. Tételezzük fel, hogy egy *a* állapoton *o* operátor tűnik a legjobb lépésnek. Ekkor a keresés folyamán előfordulnak olyan állapotok, amelyek csak kis mértékben különböznek *a* állapottól. Például ha tekintünk egy a' állapotot amely *a* állapottól csak két-három lépés távolságra van, akkor a két állás elég hasonló egymáshoz, hogy feltételezzük *o* még mindig a legerősebb lépés. A cáfoló lépésnél csak a vágások darabszáma az, amely meghatározza, hogy egy lépés mennyire erős. A history heuristic esetében egy állapoton alkalmazható operátorok közül az a legerősebb, amely vágást eredményez, vagy ha az állapotnál nem volt vágás, akkor az összes operátor alkalmazása után a legnagyobb heurisztika értékhez vezető operátor lesz a legerősebb lépés. Itt viszont nem azt tartjuk nyilván a bejárás folyamán, hogy hányszor volt egy operátor a legerősebb, hanem minden operátor rendelkezik az állapotok heurisztikus értékéhez hasonlóan egy értékkel. Ezen érték úgy alakul, hogy ha egy operátor egy állapot esetében a legerősebbnek bizonyult, akkor értékét egy súlyozott értékkel megnöveljük. Mivel minél mélyebben vannak a bejárás közben a kiértékelt levélcsúcsai a részfának, annál pontosabbak lesznek a heurisztika értékek, valamint annál nagyobbak lehetnek a magasabb szinten és a mélyebb szinten lévő állapotok között felmerülő különbségek. Ezért célszerű az operátorok súlyozását is mélységhez kötni. Lényegben arról van szó, hogy minél mélyebben tűnik hatékonynak egy operátor, annál biztosabb, hogy a továbbiakban is az marad. A mélységhez köthető súlyok közül a módszer kialakítása közben a leghatékonyabbnak a 2^d súly tűnt, ahol *d* jelöli az operátor alkalmazásának mélységét. Tehát az operátorok értékei dinamikusan változnak a lépésajánlások alatt. Ebből kifolyólag ahhoz, hogy mindig az aktuálisan leghatékonyabb operátort alkalmazza az algoritmus egy állapotnál, az alkalmazható operátorokat, minden operátor alkalmazás előtt csökkenő sorrendbe kell rendezni. A módszer esetében a cáfoló lépések nagyon gyorsan igen nagy értékekre tesznek szert, így

várhatóan azokat próbálja ki először egy adott állásra, ezért elmondható, hogy a módszer magában foglalja a cáfoló lépés elvét is.

3.4. Nyitány könyvtárak

Ahhoz hogy a program a dáma állások részleteiben rejlő előnyöket felismerje a játék folyamán, legkevesebb 15 mélységig kellene bejárnia a részfákat. Ez fokozottan igaz a játék kezdetekor, ahol a fent említett előnyökhöz megközelítőleg több mint 25 mélységig kellene előre látnia az algoritmusnak. Az olyan dámaprogramok esetében, melyek már világszínvonalú játékot képesek produkálni, ezért elengedhetetlen a nyitány könyvtárak alkalmazása. A 25 mélységű részfák bejárása még a legjobb és leghatékonyabb algoritmusok számára is túlságosan időigényes ahhoz, hogy egy versenykeretek között lezajló játszma folyamán kivitelezzék. Viszont a részletekben rejlő többletinformáció hiányában a program szinte biztos, hogy a játszma elején hibát követ el. Ez nem okoz gondot egy átlagos dámajátékossal szemben, viszont a profi játékosok ellen már szinte biztos a program veresége. Ezért valamilyen módon azon lépéssorozatokat, melyek a program szemszögéből előnyös, vagy biztonságos állapotokba vezetnek, ezen állásokkal együttvéve, előre össze kell gyűjteni egy statikus adatszerkezetbe. A lépéssorozatok lehetnek a program szemszögéből jó nyitásokat alkotó lépéssorozatok, vagy az ellenfél, egy adott nyitása esetén meglépendő válaszlépések. A nyitány könyvtárakat a játékkal foglalkozó szakirodalmak, illetve a profi játékosoktól gyűjtött információk alapján állítják össze. Mielőtt egy nyitás bekerülne az adatbázisba, előtte számítógépes környezetben tesztelés alá vetik, és a tesztek eredményei alapján, ha szükséges, akkor módisítják. 1996-ban a Chinook névre keresztelt program nyitány könyvtára megközelítőleg 60000 állást tartalmazott, az állásokon meglépendő lépésekkel, melyek által alkotott nyitások közül megközelítőleg százas nagyságrendben módosított a jobb eredmény elérése érdekében.

3.5. Végjáték adatbázisok

Még 1986-ban alkalmazták először a végjáték adatbázisokat a sakkprogramoknál, és ezen adatbázisok mintájára alkalmazták a dámaprogramok esetében is. A végjáték adatbázisok alkalmazásának lényege, hogy egy, a végállapotokból visszafele haladó keresés, a játékfa egy kis részére, amelynek állapotaiban csak kevés számú bábu található a táblán, még elég belátható időn

belül elvégezhető. Így előzetesen össze lehet gyűjteni egy adatbázisba, hogy a játék vége fele mely állapotokból, milyen végállapotok érhetőek el a játékfában, figyelembe véve az ellenfél lépéseit. Ahogy a 2.1 fejezetben is tárgyaltam, az egyértelmű hogy pontos becslést csak a fa végállapotaira lehet adni, tehát azon állapotokra melyek állásain valamelyik játékos már nem rendelkezik egyetlen bábuval sem, vagy valamelyik játékosnak már nincs több lépéslehetősége. Egy visszafelé haladó kereséssel ezen értékek több lépéssel előrébb hozhatóak a játékfában a játék végétől, ezáltal lecsökkentve azon lépések megtételének számát, melyek szükségesek ahhoz, hogy a lépésajánló algoritmus pontos és nem csak becsült értékek alapján tudjon dönteni. Tehát ha egy játszma, a lépésajánló algoritmus által korábban ajánlott lépéseken keresztül, a végéhez közeledik, és az aktuális lépésajánlás közben végrehajtott, eddigi bejárás alatt talál egy olyan állapotot, amely már szerepel a végjáték adatbázisban, akkor az állapot pontos értéke már meghatározható a végállapotokhoz vezető út meghatározása nélkül. Ebből kifolyólag a program hatékonysága megnő a végjátékokban, a pontos értékékek korábban való felhasználásának köszönhetően. Viszont ez nem csak azzal az előnnyel jár, hogy egy adott állapot heurisztikus értéke már pontos érték lesz. Hanem azzal is, hogy az adatbázisból meghatározható az állapoton alkalmazandó legkedvezőbb végállásba vezető lépéssorozat is, mely a bejáráshoz szükséges idő csökkenésével jár. Ugyanis a lépésajánló algoritmusnak azt az utat a részfában már nem kell bejárnia.

4. Megvalósítás

4.1. Operátorok

Az AutoCheckers összesen kilenc operátorral dolgozik. Négy mozgató operátorral, négy ütő operátorral, és egy ütéssorozatokat megvalósító operátorral. Mivel ezek az operátorok elég sok azonos tulajdonsággal, és viselkedéssel rendelkeznek, ezért ezen tulajdonságok és viselkedési formák az operátorokra általánosan is megadhatóak voltak. A megvalósításhoz az öröklődést felhasználva elegendő volt a `BaseOperator` absztrakt osztályt implementálni, és minden további operátor osztály ebből az osztályból származtatni. Az már a játék reprezentációjából is kiderül, hogy minden operátor objektumnak, viselkedését tekintve, meg kell tudni mondania, hogy alkalmazható-e egy adott állapot objektumon, illetve az operátor végrehajtásakor tudnia kell

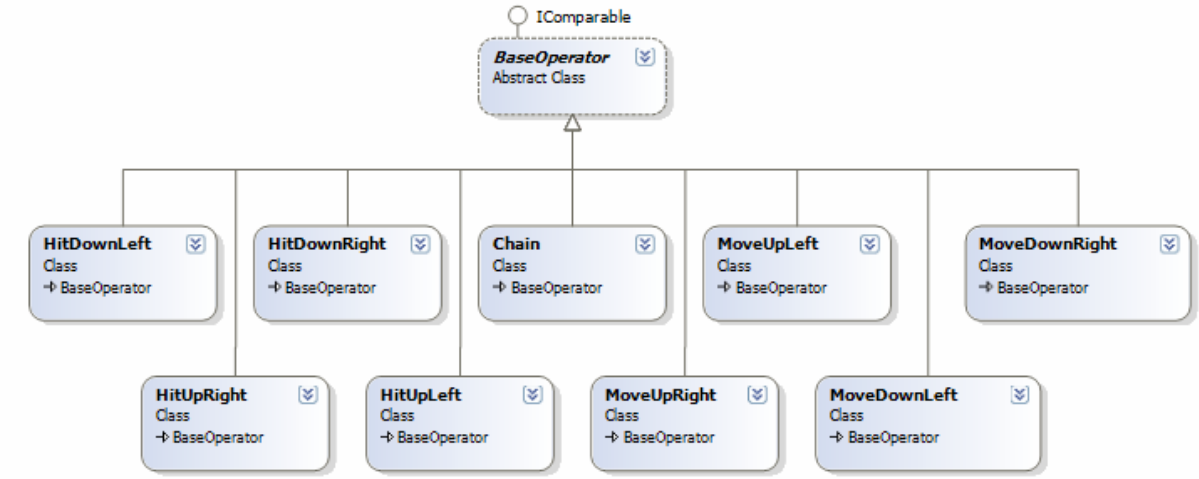
módosítani az állapot objektumban található állást az általa megvalósított lépés alapján. Mivel viselkedésről beszélünk, ezért az operátorokhoz valamilyen formában olyan tagfüggvény specifikácókat kellett meghatározni, melyeket a fejlesztés alatt később az operátoroknak kötelezően implementálniuk kellett. A `BaseOperator` tehát ebből kifolyólag tartalmaz két absztrakt tagfüggvényt, melyek a `UseOperator` és az `IsUsableOnStand`. Az `IsUsableOnStand` egy állapotot vár paraméterként, és 1.1 fejezetben említett *előfeltétel(l,s)* függvénnyel egyenértékű. A `UseOperator` ugyancsak egy állapotot vár paraméterként és visszatérési értékként az operátor által megvalósított lépés végrehajtásával módosítva visszaad egy új állapotot. Ezen felül minden operátor objektum a következő közös tulajdonságokkal rendelkezik:

- A lépés kiinduló mezőjének sor-, és oszlopindexe,
- A lépés célmezőjének sor-, és oszlopindexe,
- A lépés history heuristic értéke,
- A rajzoláshoz szükséges forrás-, és célkoordináták.

Mivel az állapotban az állást egy mátrixként implementáltam, melyet majd az állapotok implementálásának tárgyalásakor részletezek, ezért egy kivételével minden operátorjellemző egyértelmű. A rajzolási koordinátákra a grafikus felhasználói felület megvalósításához volt szükség, és részletesebben majd annak tárgyalásakor térek ki rá. A `BaseOperator` absztrakt osztály alkalmazása nem csak azzal az előnnyel rendelkezik, hogy magába foglalja az operátorok közös tulajdonságait, hanem az operátorokhoz egy elérési felületet is biztosít más osztályok számára, így azok az összes operátort egységesen kezelhetik. Az absztrakt tagfüggvényeken kívül még az absztrakt osztály tartalmazza `HitOnI` és `HitOnJ` paraméter nélküli `virtual` módosítóval rendelkező tagfüggvényeket, melyek az operátor objektum által eltávolított bábu sor-, és oszlopindexeit határozzák meg. A két tagfüggvény azért került az absztrakt osztályba annak ellenére, hogy a mozgató operátorok nem távolítanak el bábut, mert minden operátorhoz a későbbi munka megkönnyítése érdekében egy egységes felületet kellett biztosítanom, ezért olyan jellemzőket is fel kellett venni, melyek nem az összes operátorra jellemzőek, hanem azoknak csak egy kisebb csoportjára. Mivel nem minden operátorra jellemző, hogy bábut távolít el, ezért az absztrakt osztály `HitOnI` és `HitOnJ` tagfüggvényei olyan értékkel térnek vissza, amelyek kívül esnek az állást reprezentáló mátrix indexhatárain, ezért nem megfelelő használatuk kivételhez vezet. Értelemszerűen e tagfüggvényeket az ütő operátorok osztályaiban túl kellett terhelni, hogy

a megfelelő értékekkel térjenek vissza. A `BaseOperator` osztály ezen felül magába foglal még két statikus `List<BaseOperator>` típusú `Moves` és `HitMoves` generikus listát. Mivel absztrakt osztály nem példányosítható, ezért értelemszerűen a listák a `BaseOperator` osztály leszármazott osztályainak objektumait tartalmazzák. Ugyanis véges számú operátor létezik a játékban, és ezek a játék elejétől a végéig változatlanok maradnak. Ezért érdemes ezen operátorokat még a program indításakor példányosítani, hogy ne lassítsa le a lépésajánlás algoritmusát a bejárás közben való példányosításokkal. Annak megvalósítására, hogy már a program indításakor az összes mozgató és ütő operátor példányosítása végbe menjen, három konstruktor megvalósítása volt szükséges minden ütő és mozgató operátort megvalósító osztályban. Az első konstruktor egy privát konstruktor, mely két paramétert vár, amelyek a tábla azon pozíciói, amelyek a lépés kiinduló pozíciójának sor-, és oszlopindexe. Majd e paraméterekből, a tábla mátrixának tartalmi tulajdonságai miatt, melyet majd az állapotok tárgyalásakor részletezek, a privát konstruktor a példányosításkor számolja ki, hogy melyik pozíció a lépés célpozíciója, illetve az ütő operátorok esetében mely pozícióról távolít el bábut. A második konstruktor egy statikus konstruktor, mely az osztályra való első hivatkozáskor fut le. A konstruktor feladata, hogy sorra vegye azon kiinduló pozíciókat a táblán, melyekkel az operátor, alkalmazása során nem lép le a tábláról, és az összes ilyen pozícióhoz létrehozzon egy operátor objektumot, amit elhelyez az absztrakt osztály megfelelő statikus listájában. A harmadik egy paraméter nélküli publikus konstruktor melynek a törzse üres. A publikus konstruktor első hívása után lefut a statikus konstruktor, ami az osztály által megvalósított összes lehetséges operátor objektumot a privát konstruktorok felhasználásával lepéldányosítja. Így a nyolc ütő és mozgató operátor publikus konstruktorának első hívása után a `BaseOperator` statikus listái tartalmazni fogják a játék során megléphető 98 mozgató, és 72 ütő operátort. Más volt a helyzet az ütéssorozatot megvalósító `Chain` osztállyal, ugyanis a 72 alkalmazható ütő operátor miatt, az összes ütéssorozat megtalálása már sokkal több időt vett volna igénybe, és az ütéssorozatok előfeltétele túlságosan összetetté vált volna. Az osztály azért is speciális eset, mert az ütéssorozatok egyszerű ütésekre bonhatóságának tulajdonsága miatt, ütő operátorokat tartalmaz egy `OperatorsInChain` adattagban, amely egy `List<BaseOperator>` típusú generikus lista. Mivel ezen operátorok dinamikusan alakulnak a részfák bejárása folyamán, ezért az osztály tartalmaz egy plusz tagfüggvényt a többi operátorhoz képest. Az `InsertInChain` elhelyezi a listában a megfelelő helyre a paraméterül kapott ütő

operátort objektumot. Az operátor minden más adattagját az ütő operátorok listájának elemeitől kapja úgy, mint a kiinduló mező sor-, és oszlopindexét, és a history heuristic értéket a lista első elemétől, valamint a célmező sor-, és oszlopindexét a lista utolsó elemétől. Az ütéssorozat végrehajtása, tehát egyértelműen egyenértékű a listában szereplő ütő operátorok sorrendben történő végrehajtásával, valamint egy ütéssorozat akkor alkalmazható, ha az első ütés alkalmazható. Az ütéssorozat végrehajtásakor arra külön kellett figyelnem, hogy az ütéssorozat 1.4 fejezetben tárgyalt tulajdonságai miatt külön értékek jelzik a mátrixban a leütött bábukat, és ezeket az ütéssorozatokat megvalósító operátorok alkalmazása után az üres mező értékére kell állítani. Valamint mivel minden gyalog csak a lépés megtétele után válik dámává, ezért miden operátor végrehajtása után ebből a szemszögből is meg kell vizsgálni az állást.

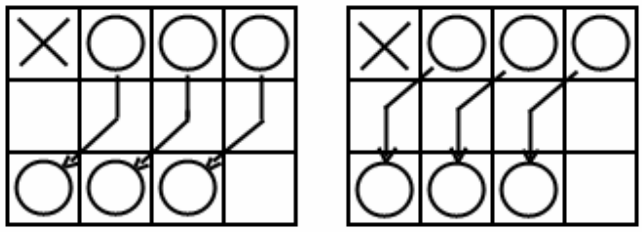


4.1.1 ábra: Operátorok osztályhierarchiája

4.2. Állapotok

Már a játék reprezentációjában is látható, hogy egy állapotnak tartalmaznia kell a játék egy állását, és hogy melyik játékos következik lépni abban az állásban. A `CheckersStand` osztály ebből kifolyólag tartalmaz egy 8×4-es `byte` típusú elemeket magába foglaló `table` mátrixot. Annak ellenére, hogy a játék táblája 8×8 darab mezőből áll, a mátrix mérete csak fele a tábla méretének. Ez azért volt szükséges, mert ugyan a tábla 64 mezőből áll, ám az átlós lépések folyamán, ahogy már utaltam erre a 1.4 fejezetben is, csak a tábla felét használják a játékosok, ezért felesleges az egész táblát letárolni. A `table` úgy reprezentálja a játék tábláját, mintha az

horizontálisan össze lenne nyomva. Ez viszont azzal a következménnyel járt, hogy az egyes mezők a szomszédságaikat tekintve elcsúsztak az eredeti táblához képest, amely az operátorok tekintetében jelentett némi változtatást, ahogy a 4.2.1 ábrán is látható. Mivel az operátorok már a



4.2.1 ábra: Balra lefele történő ütés az átalakított tábla páros és páratlan soraira

program indításakor létrejönnek, ezért az így felmerült szomszédsági problémából adódó számítások, mint ahogy az már felmerült az előző fejezetben is, az operátorok privát konstruktoraiban előre elvégezhetőek, így a tábla ezen módosítása a lépésajánlás algoritmusát nem lassítja. A tábla értékeit tekintve a következőket tartalmazhatja:

- 0 jelzi az üres mezőket,
- 1 és 3 jelzi a fekete és fehér gyalogokat
- 2, és 4 jelzi a fekete és fehér dámákat,
- 5 jelzi a leütött bábukat,

A leütött bábuk külön jelölésének szükségességét az ütéssorozatok 1.5 fejezetben tárgyalt tulajdonságai magyarázzák. A játékosok azonosítására a `Players`, egy `enum` felsorolásos típus szolgál, mely `Player_A` értékből, ami a fekete oldalt, és `Player_B` értékekből áll, ami a fehér oldalt jelöli. Az állapotban soron következő játékost egy állapot objektum a `NextPlayer` adattagjában tárolja, és értéke a `Players` értékeiből kerül ki. A megvalósítás tekintetében nem volt értelme külön-külön megvalósítani a kezdőállapotot és a végállapotokat. A kezdőállapotot a `CheckersStand` konstruktora valósítja meg, az általa létrehozott objektum `table` mátrixának, a kezdőállásnak megfelelő feltöltésével, és a `NextPlayer` értékének a fekete játékost jelölő értékre állításával. Ahhoz, hogy állapotról el tudjuk dönteni, hogy az végállapot-e, meg kellett valósítani a 1.2 fejezetben tárgyalt *végállás(s)* függvényt. A játék végét meghatározó tulajdonságok miatt, amit szintén a fent említett fejezet tartalmaz, minden állapotnak tisztában kellett lennie, az általa tartalmazott állás következő jellemzőivel. Melyik oldal hány figurával rendelkezik, hány lépés

van még hátra egy döntetlen állásig, illetve, hogy van-e az állapoton alkalmazható operátor, amit az `Operators`, egy `List<BaseOperator>` típusú, generikus lista elemszáma adja meg, amely az összes az állapoton alkalmazható operátort tartalmazza. Egyértelmű hogy a végállapot vizsgálatot végrehajtó paraméter nélküli `CheckTerminal` tagfüggvény abban az esetben tér vissza igaz értékkel, ha az állás fent említett jellemzői közzül, legalább egynek nulla az értéke. Az összes alkalmazható operátor összegyűjtése a paraméter nélküli `GetUsableOperators` tagfüggvény feladata. Az alkalmazható mozgató és ütő operátorok összegyűjtésének megoldása egyértelmű, viszont az alkalmazható ütéssorozatok összegyűjtése már nem. Ráadásul amennyiben létezik legalább egy ütéssorozat is egy operátor esetében, a 1.5 fejezetben tárgyalt ide vonatkozó játékszabályok miatt az egyszerű ütő és mozgató operátorok szabálytalan lépésnek minősülnek. Ebből kifolyólag mivel az ütéssorozatok nincsenek előre meghatározva, ezért valamilyen módon, ha már egy ütés is alkalmazható az állapoton, akkor meg kell hozzá határozni a leghosszabb ütéssorozatokat. Tehát a `GetUsableOperators` úgy működik, hogy előbb egy `List<BaseOperator>` típusú `OperatorsForChains` generikus listába összegyűjti az alkalmazható ütéseket. Ha e lista üres marad, akkor már csak a mozgató operátorok alkalmazhatóak az állapoton, amelyek már előre adottak. Ha van legalább egy alkalmazható ütés, akkor az állapotot átadja egy mélységi keresőnek, melyet a `DephtFirstSearch` osztály valósít meg. A keresőnek az a feladata, hogy megtalálja az összes alkalmazható leghosszabb ütéssorozatot az összes alkalmazható ütő operátorhoz. A mélységi kereső a játékfa egy részfáját úgy járja be, hogy csak azokat az ütéseket megvalósító operátorokat alkalmazhatja a keresés folyamán a részfa összes állapotára, amely operátorok alkalmazhatóak az egyes állapotokon és kiinduló mezőik megegyeznek, az ezen állapotokat létrehozó operátor célmezőjével. Ebből kifolyólag minden állapot objektumnak nyilván kell tartania az azt létrehozó operátort objektumot. Mivel, ugyanúgy egy részfa bejárásáról van szó, mint a lépésajánló algoritmusok esetén, itt is meg kell vizsgálni, hogy egy állapot célállapot-e. A mélységi kereső célállapot tesztje abban különbözik az eredeti célállapot teszttől, hogy nem kell vizsgálni a döntetlen lehetőségét, ugyanis az ütéssorozatok egy lépésnek minősülnek. Mivel az ütéssorozatok maximális hossza eleve ad egy mélységi korlátot ezért a keresésnek külön nem kell ezt megszabni. Amennyiben a bejárás eléri a mélységi keresés részfájának levélelemét, akkor a levélállapottól a gyökérállapot közvetlen utódjáig terjedő úton elhelyezkedő állapotokat létrehozó

operátorok fordított sorrendű felírása adja az ütéssorozatot. Tehát ezen operátor sorozat alapján a kereső a levélállapotokban létrehoz egy-egy `Chain` típusú objektumot. Ehhez viszont az kell, hogy minden állapot objektum meg tudja mondani, hogy melyik állapot objektum a szülőállapota. Hogy a mélységi kereső a leghosszabb ütéssorozatokat határozza meg, minden ütéssorozat objektumot egy `List<BaseOperator>` típusú `Chains` generikus listában helyez el. Ha a lista üres akkor az újonnan megtalált ütéssorozatot felveszi a listába. Ha nem üres a lista, akkor megnézi a lista első ütéssorozatának hosszát. Ha ez a hossz, nagyobb mint az új ütéssorozat hossza, akkor az ütéssorozat már nem szabályos, és nem kerül a listába. Ha megegyezik az új ütéssorozat hosszával, akkor felveszi azt a már meglévő ütéssorozatok közzé. Ha az új ütéssorozat hossza nagyobb, akkor a korábban talált ütéssorozatok már nem szabályos lépések, ezért kiüríti a listát és felveszi az új ütéssorozatot. A mélységi kereső a keresés befejeztével ezt a listát adja át, mint visszatérési érték. Mivel a kereső célszerűen az egy lépésből álló ütéseket is ütéssorozatnak ismeri fel, ezért a visszaadott lista soha nem lesz üres. Felmerülhet a kérdés, hogy ez a keresés mennyire lassítja le a lépésajánlást. Az ütéssorozatok elvi hosszkorlátja gyalogok esetében három, és dámák esetében pedig tizenkét ütés. Az viszont belátható, hogy ezek az ütéssorozatok annyira speciális esetek, hogy gyakorlatban csak ritkán, vagy szinte soha nem fordulnak elő. Ha egy felső becslést kell nézni, akkor az általam tapasztaltak alapján maximum kettő ütéshelyzetet taralmaz egy állás, és ebből maximum két ütéssorozat léphető meg, melyek maximum három hosszúságúak. Így elmondható hogy az esetek többségében a legrosszabb eshetőségben a mélységi keresőnek 6*72 operátort kell megvizsgálnia. A lépésajánlások és keresések közben az állapotok csak minimális változásokon esnek át, ezért nincs értelme, hogy minden új állapot objektumot újra példányosítson a program és beállítsa az adattagok értékét az operátor hatásának megfelelően. Ezért a `CheckersStand` osztáy megvalósítja a .NET keretrendszer beépített `ICloneable` interface-ét, mely lehetővé teszi egy objektumról való másolat készítését. Az interface megvalósításához csupán az általa specifikált `Clone` tagfüggvényt kell implementálni, mely egy objektumra meghívva annak egy másolatával tér vissza. Így az operátorokat elegendő ezen másolatokra végrehajtani ahhoz, hogy új állapotokat kapjunk a bejárások során. Azt viszont figyelembe kellett venni, hogy a referencia típusok csak referencia szinten másolódtak, ezért gondoskodnom kellett arról hogy az eredeti, és a másolt állapot ne osztozzon az általuk tartalmazott objektumokon. Az $\alpha\beta$ vágásasal megvalósított

minimax algoritmus a heurisztika értékek alapján dönt a lépést illetően, ezért minden állapot objektum a hozzá tartozó aktuális heuriszika értéket az `AlfaBetaValue` adattagjában tárolja. A bejárások során az is fontos az operátorok history heuristic értékét tekintve, hogy az egyes állapot objektumok meg tudják mondani, hogy a rajta alkalmazott operátorok közül melyik tűnt a legjobbnak. Ezért minden állapot objektum a rajta alkamazott legjobb operátor objektumot a `BaseOperator` típusú `BestForStand` adattagjában tárolja. A lépésajánlás mélységi korlát ellenőrzéséhez szükséges volt még, hogy minden állapot objektum tisztában legyen azzal, hogy a játékfa milyen mélységében helyezkedik el, melyet az állapotok `Depth` adattagjának értéke határoz meg.

4.3. Heurisztika

A heurisztika megvalósítását tekintve a heurisztikus kiértékelő függvények a `CheckersStand` osztály statikus tagfüggvényeiként kerültek implementálásra. A `BlackHeuristic` a fekete oldal, pedig a `WhiteHeuristic` a fehér oldal szemszögéből értékeli ki a paraméterként kapott állapotot. Azt, hogy a lépésajánlás közben melyik tagfüggvény használandó, a `HeuristicMethod` képviselő objektum mondja meg. Mint már tárgyaltam a 2.2 fejezetben egy hatékony heurisztika kialakításához a legjobb út ha a gép önmagával játszik. Az AutoCheckers ennek eredményeképpen öt különböző jellemzőjét veszi figyelembe az állásoknak. Csak említésként a Chinook program összesen 25 jellemzővel dolgozik. Az első és legkézenfekvőbb jellemző, hogy minél több bábuja van a támogatott játékosnak a táblán és az ellenfélnek minél kevesebb, annál jobbnak kell lennie a heurisztikus értéknek. Viszont egyértelmű az is, hogy két bábu típus létezik a játékban, melyek nem egyenértékűek. Ebből következően az alap heurisztika a fekete odal szemszögéből:

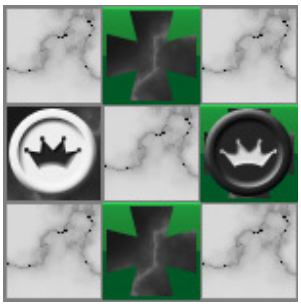
$$f(a) = w_1 \sum_{i,j} g_1(a_{ij}) - w_1 \sum_{i,j} g_2(a_{ij}) + w_2 \sum_{i,j} d_1(a_{ij}) - w_2 \sum_{i,j} d_2(a_{ij})$$

ahol $g_1(a_{ij})$ függvény a tábla fekete gyalogokat tartalmazó a_{ij} mezője esetén, és $g_2(a_{ij})$ függvény a tábla fehér gyalogokat tatalmazó a_{ij} mezője esetén egyet vesz fel értékül, egyébként nullát, és w_1 a gyalogokhoz tartozó súlyérték, valamint $d_1(a_{ij})$ függvény a tábla fekete dámákat tartalamzó a_{ij} mezője esetén, és $d_2(a_{ij})$ függvény a tábla fehér dámákat tatalmazó a_{ij} mezője

esetén egyet vesz fel értékül, egyébként nullát, és w_2 a dámákhoz tartozó súlyérték. Az *i*, *j* indexek értkei pedig a tábla sorait, és oszlopait jelölik, ami fenn áll a későbbi heuriszitka függvények esetében is. Tehát az első amit ki kellett deríteni hogy mennyit ér egy dáma, és mennyit a gyalog ahhoz, hogy a program hatékonyan játsszon. A súlyértékek terén való későbbi szabadabb mozgás érdekében a gyalogok súlyértékét 100-ra rögzítettem. A dáma súlyértékeire a 140-től 200-ig terjedő, tizessével növekedő értékskálát próbáltam ki. A legkisebb érték már az első heurisztika verzió esetén azért eset ki, mert ezen súly esetében a dáma alulértékelődöt, és hajlamos volt a program egy gyalogért cserébe feláldozni. A legnagyobb súly következménye hasonló volt, ugyanis a dáma túlértékeltsége miatt, ha kellett, a program a gyalogokat maradéktalanul feláldozta a dáma túlélése érdekében. A következő jellemző a gyalogok rangja volt. Egy gyalog *rangja* a dámában egy olyan érték, amely megmutatja, hogy mennyire áll közel a dámává váláshoz. Tehát az ellenfél térfelének legelső sorától egy sor távolságra lévő, a támogatott játékoshoz tartozó gyalogok 6. ranggal, és attól a sortól legtávolabbi sorban található ugyanahoz a játékoshoz tartozó gyalogok 0. ranggal jellemezhetőek. Ez azért is hasznos, mert a kis mélységű keresések folyamán a dámává váláshoz vezető lépéssorozatok végei a játék elején még a bejárandó részfák mélységkorlátján túl helyezkednek el. Viszont ha figyelembe veszi a gyalogok rangját, akkor a program arra fog törekedni már a játék elejétől, hogy haladjon az ellenfél felé, így hamarabb jutva dámákhoz. Minden dámával kapcsolatos forrás a $rang^2$ érték használatát javasolta, de mellette kipróbátam a rangot önmagában is. Az eredmény a $rang^2$ jobb hatékonyságát igazolta. Mivel ezen jellemző megemelheti a gyalogok értékét maximum 136-ra, ezért az fentebb bemutatott alulértékelési probléma ismét megjelent, és kiesett a 150-es és a 160-as súly a dámák esetében. Ugyancsak megjelent a dámák felülértékelésének problémája is egy kicsit átalkult formában. A program magasabb dámasúlyok esetén törekedett arra, hogy a legmagasabb ranggal rendelkező gyalogból minél hamarabb dáma legyen még akkor is, ha a dámává válsá felé vezető úton elvesztette a gyalogot. Ezért a dáma súlyok közül kiesett a 190-es érték. Tehát a heurisztika a ranggal egybevéve a következő formában volt megadható a fekete oldal szemszögéből:

$$f'(a) = f(a) + \sum_{i,j} g_1(a_{ij}) * rang(a_{ij})^2 - \sum_{i,j} g_2(a_{ij}) * rang(a_{ij})^2$$

ahol $f(a)$ a korábban megadott heurisztika függvény, és $rang(a_{ij})$ függvény, ha a tábla a_{ij} pozíciója nem üres akkor a bábu rangját veszi fel értékül, egyébként pedig nullát. A programnak viszont a dámával kapcsolatban még mindig nem volt használható információja. Egy dáma akkor van rossz pozícióban, ha a tábla szélén áll, ugyanis akkor csapdába ejthető. Ebből kifolyólag minden olyan a támogatott játékoshoz tartozó dáma, amely a tábla szélén áll, veszít az értékéből és minden olyan, az ellenfélhez tartozó dámának, amely a tábla szélén áll, megnő az értéke. A legrosszabb helyen tehát a támogatott játékos azon dámája van, amelyik a tábla valamelyik sarkában helyezkedik el, ugyanis ekkor mindeössze egy lépése van hogy elhagyja azt a mezőt amelyen éppen áll, így tehát az ilyen dáma kétszer is veszít az értékéből. Ezért az értékvesztést úgy kellett meghatározni, hogy még a legnagyobb értékvesztés után is a dáma értékesebb maradjon mint egy gyalog. Ezen érték az AutoChechkers esetében 15 lett, ugyanis így a maximális értékvesztéssel rendelkező dáma még a legkisebb súly esetén is 140-es értékkel rendelkezik, a gyalog maximális 136-os értéke ellen. Tehát a fekete oldal heurisztikus függvénye a következő képpen alakult:

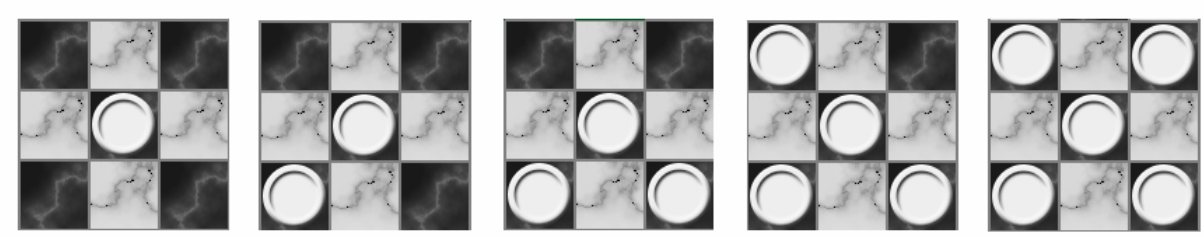


4.3.1 ábra: Dáma csapda

$$f''(a) = f'(a) - w_3 \sum_{i,j} s_1(a_{ij}) + w_3 \sum_{i,j} s_2(a_{ij})$$

ahol $f'(a)$ a korábban felírt heurisztikus függvény. $s_1(a_{ij})$ függvény minden a tábla szélén álló fekete dáma esetén egy értéket vesz fel, minden a tábla valamely sarkában található dáma esetén kettő értéket vesz fel, egyébként pedig nulla értéket. $s_2(a_{ij})$ függvény pedig megegyezik az $s_1(a_{ij})$ függvénnyel azzal a különbséggel, hogy fehér dámák esetén veszi fel a fent említett értékeket. w_3 pedig a szélen található dámák súlya, amely a korábban említett 15-ös értéket veszi fel. Itt már a program elég jól játszott, viszont a dámává válás érdekében esetenként még mindig hajlamos volt egy gyalogot értelmetlenül feláldozni. Valamint az esetek többségében a program vereségével ért véget a játék, ha ez ellenfél a saját térfele legelső sorából a játék folyamán nem mozdította el a gyalogjait. A fent említett heurisztika alkalmazása mellett a program hajlamos volt nagyobb gyalogáldozatokra azért, hogy megbontsa az ellenfél legelső sorát. Viszont mire sikerült azt megtennie, már akkora hátrányba került a bábuk számát tekintve, hogy nem tudta azt

behozni. Ennek az volt az oka, hogy a program, a minél nagyobb rangok elérésére törekedve, olyan lépéseket alkalmazott, ahol a gyalogoknak nem volt megfelelő „támogatottságuk”, így az ellenfél azokat már könnyebben eltávolíthatta a tábláról. Ebből kifolyólag vettem fel a heurisztikus jellemzők közé az egyes bábuk védettségi szintjét. Az elv az, hogy minél több azonos színű bábu vesz körül egy bábut, azt annál nehezebb az ellenfélnek a tábláról eltávolítani.



4.3.2 ábra: Védettségi szintek értékük szerint balról jobbra növekvő sorrendben

Ehhez egy olyan súlyt kellett találni, ami elég nagy ahhoz, hogyha kell, felülbírálja a rangot, de a jellemzők között korában kialakított rendszert ne borítsa fel. Az AutoCheckers esetében e súly értéke 20 lett. Tehát egy bábu értéke minden azonos színű közvetlen szomszédja esetén, 20-as értékkel megnövelendő. Természetesen ugyanezen értékek az ellenfél bábuira számolva negatív tényezőként szerepelnek a heurisztikában. Kérdés, hogy mi a helyzet a tábla szélén álló bábuk esetén. A dámák esete egyértelmű az előző heurisztikajellemző miatt. Viszont egy szélen álló gyalog ugyanúgy csapdába eshet, ahogy a dáma. A játék folyamán mivel a szélen álló gyalogok mindössze két azonos színű közvetlen szomszéddal rendelkezhetnek, ezért ezeket a szélen álló gyalogokat tartalmazó állásokat, a program nem fogja felhasználni, mert alacsonyabb értékekkel rendelkeznek mint a stabilabb helyzeteket tartalmazó állások. A jellemző alkalmazásának eredménye az lett, hogy a program igyekezett egységben mozgatni bábuit úgy, hogy az ellenfél lépéslehetőségeit tekintve egy bábuval vagy ne tudjon lépni, vagy ütéshelyzetbe kényszerüljön. Így tehát az ellenfél a játék során egy idő után arra kényszerült a program ellen, hogyha el akarta kerülni a gyalogjai gyors elvesztését, meg kellett bontania a térfele legelső sorát, a dámák kialakításának lehetőségét nyújtva ezzel a program számára. Így tehát a végső heurisztika függvény a következő formában adható meg a fekete oldal számára:

$$f'''(a) = f''(a) + w_4 \sum_{i,j} t_1(a_{ij}) - w_4 \sum_{i,j} t_2(a_{ij})$$

ahol $f''(a)$ a korábban felírt heurisztika függvény. $t_1(a_{ij})$ függvény a tábla a_{ij} pozíción elhelyezkedő fekete bábu esetén rendre 1, 2, 3, 4 értékeket vesz fel aszerint, hogy a bábunak 1, 2, 3, 4 azonos színű közvetlen szomszédja létezik a táblán, egyébként pedig a nulla értéket. $t_2(a_{ij})$ függvény értékeit tekintve megegyezik az előbbivel, azzal a különbséggel, hogy a fehér bábuk esetén veszi fel a fent említett értékeket. $w4$ pedig a védettségi szinthez tartozó súlyérték. Az így kapott fekete oldalon játszó játékost támogató heurisztika innen már könnyen átalakítható, hogy az ellentétes oldalt támogató heurisztikát kapjunk. Mindössze annyi változtatást kell alkalmazni, hogy a fekete és fehér játékosra vonatkozó jellemzők számértékeit a fentebb említett képletekhez képest, ellentétes előjellel kell szerepeltetni.

4.4. Lépésajánlás

Az $\alpha\beta$ vágás és a minimax algoritmus pszeudokódját szinte minden mesterséges intelligenciával foglalkozó könyvben megtaláltam és ezen kód alapján írtam meg az AutoCheckers lépésajánló algoritmusát, az `AlphaBeta` osztály keretében. A lépésajánlást az osztály `SearchForStep` tagfüggvénye valósítja meg, melyet az ugyancsak egy paraméteres `ComputersTurn` tagfüggvénye hív meg. Mindkét tagfüggvény az aktuális állást kapja meg paraméterként. A `SearchForStep` befejeztével a paraméterül kapott aktuális állapot objektum `BaseOperator` típusú `BestMove` adattagja tartalmazza a minimax algoritmus által legjobbnak vélt lépést. A `ComputersTurn` ezt az operátort alkalmazza az aktuális állapot objektumon, és az így létrejött új állapot objektumot adja vissza. A lépésajánló algoritmus megvalósításában a komolyabb problémát az okozta, hogy a lépésajánlás nagyobb mélységekben történő bejárás esetén akár több másodpercet is igénybe vesz a megfelelő lépés megtalálásához. Addig viszont a program, a felhasználó semmilyen interakciójára nem válaszol. Ezért szükséges volt, hogy maga a lépésajánlás szálként a háttérben fusson, a felhasználó és a program esetleges kommunikációja fenntartása érdekében. A szálkezelést és a hozzá kapcsolódó műveleteket a .NET keretrendszer `System.Threading` névtere tartalmazza. Egy szál létrehozása az ugyanezen névtér által tartalmazott `Thread` osztály egy objektumának példányosításával egyenértékű. Ezen osztályhoz létezik olyan konstruktor, amely paraméterül egy tagfüggvény nevét várja, mely a szálban elvégzendő utasításokat tartalmazza. Egyértelmű, hogy a lépésajánlás háttérben futtatásához a

`ComputersTurn` tagfüggvényt kellett ezen konstruktor paraméteréül átadni. A szál a `Thread` objektum `Start` tagfüggvényével indítható, mely paraméterül az aktuális állást kapja. Ezt a paramétert, majd már új szálként futó `ComputersTurn` tagfüggvénynek adja át. Viszont a program főszála arról nem rendelkezik információval, hogy mikor ér véget a háttérben a lépésajánlás, így nem tudja, mikor kell a lépésajánlással módosított állapot objektumot lekérdezni a mellékszáltól. Ahhoz hogy a két szál információt cseréljen, szükséges volt egy `SearchDoneEvent` saját esemény implementálása, amelyet a háttérben futó lépésajánlás által megadott operátorral módosított állapot objektum átadása váltja ki, és kiváltódása után eseménykezelőjébe ezen állapotra módosítja a fő szál aktuális állapotát. Így a főszálban szükséges, és az aktuális állapot változásához köthető módosítások már elvégezhetőek. Figyelni kellett viszont arra, hogy a program és a felhasználó interakciói módosíthatják a háttérben futó keresést. Ilyen interakciók például a térfélváltás, és a nehézségi fok beállítása. Ezen műveletek egyértelműen új játszmát jelentenek, ezért mind a térfél váltás, mind pedig a nehézségi fok megváltoztatása esetén a korábbi lépésajánlás értelmét veszti, ezért a szálat, amennyiben még fut a háttérben, a `Thread` objektum `Abort` tagfüggvényével le kell állítani. Az is lényeges, hogy mivel a főszál és a lépésajánlás szála is kezeli az aktuális állapot objektumot, ezért ennek kezelése kritikus szekciónak minősül. Tehát ahhoz, hogy a két szál ne kezelje egy időben az aktuális állapotot, amely előre nem látható hibákhoz vezethet, mindkét szálnak az aktuális állapot felhasználása előtt, egy zárat kell arra elhelyeznie. A lépésajánlások hatékonyságát nézve a 4.4.1 táblázaton látható a tiszta, az $\alpha\beta$ vágással megvalósított és az $\alpha\beta$ vágással és előrendezéssel megvalósított minimax algoritmus által bejárt csúcsok száma a kezdőlépésre. Látható hogy a tiszta minimax algoritmus esetén a 10 mélységű keresés már belátható időn belül nem végez a bejárással. Bár az $\alpha\beta$ vágás hatékonysága ingadozó az operátorok rendezetlensége miatt, ahogy a 3.1 fejezetben is tárgyaltam, viszont a mélység növekedésével nő a hatékonysága is. Ugyanis a minimax által bejárt csúcsoknak két mélységen még 63%-át járja be, de már nyolc mélységen ez az arány mindössze 4%. Ennek az oka, hogy a vágás több levélállapotot érint mélyebb keresések esetén, tehát több heurisztikus értékkel dolgozik, ezért nagyobb az esély a vágásra. Az látható, hogy a legjobb hatékonyságot az $\alpha\beta$ vágással és előrendezéssel módosított algoritmus érte el. Az eredeti algoritmus által bejárt csúcsoknak két mélységben mindössze 47%-át járja be, és nyolc

mélységben pedig 2%-át. Az is látható, hogy az $\alpha\beta$ vágás által bejárt csúcsoknak is csupán a 75%-át járja be két mélységben és 57%-át tíz mélységben. Az, hogy nagyobb mélységekben ez az arány csökken, annak tudható be, hogy több operátorral dolgozik az algoritmus, és így pontosabb képet kap arról, hogy melyik operátor a hasznosabb, ezért hatékonyabban képes azokat sorba rendezni, így több lehetősége van vágásra, mint az eredeti $\alpha\beta$ vágással megvalósított minimax algoritmusnak. Bár a nagyobb mélységben az állapotok közötti eltérések is nagyobbak, viszont ez a probléma a program által elért maximális mélységben még a mélységhez kötött súlyozással kompenzálható. A táblázatban szerepeltetett értékek a vágással és előrendezéssel megvalósított minimax algoritmus esetén több lejátszott játszma után sem változnak drasztikusan. Öt játszma után, melyek átlagos lépés száma megközelítőleg 83 lépés, két mélységben bejárt csúcsok átlagos száma megközelítőleg 22, négy mélységben 207, hat mélységben 2218, nyolc mélységben 12106, és tíz mélységben pedig 149605.

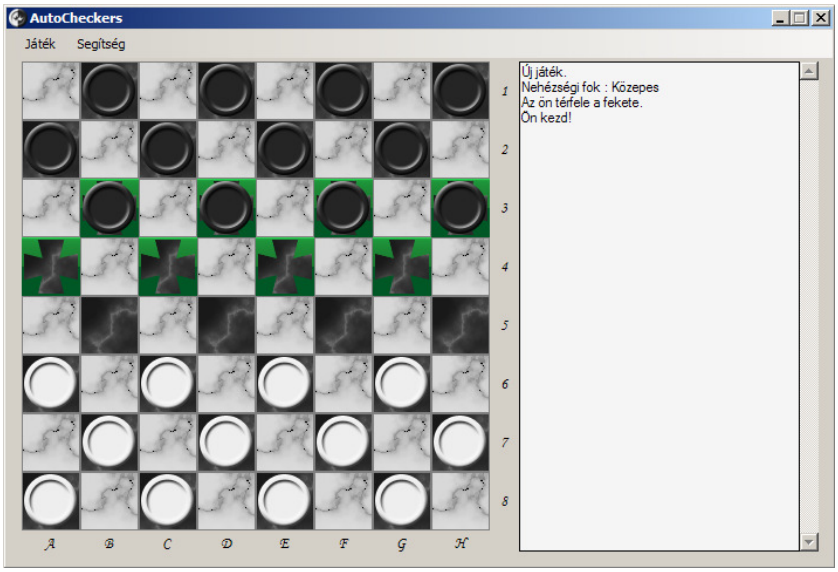
Minimax algoritmus	2 mélység	4 mélység	6 mélység	8 mélység	10 mélység
Tiszta	57	1828	45662	1051977	
$\alpha\beta$ vágással	36	669	4428	37827	295530
$\alpha\beta$ vágással és előrendezéssel	27	191	2756	22414	170406

4.4.1 táblázat: Lépésajánló algoritmusok által bejárt csúcsok száma az első lépésre.

4.5. Grafikus felhasználói felület

A grafikus felület kialakításánál törekedtem az egyszerűségre és a könnyen kezelhetőségre. A program használatához mindössze az egeret kell megfelelően alkalmazni. A program összesen három ablakból áll, melyek a .NET keretrendszer beépített `Form` osztályából származtatott formok. A főablakot tekintve maga a tábla és a rajta álló bábuk előre megrajzolt képekből állnak. Ahhoz, hogy a táblát a főablaktól az egérkezelés tekintetében elkülönítsem, létrehoztam egy `DBuffPanel` osztályt, amit a .NET keretrendszer beépített `Panel` osztályából örököltettem. Az ősosztályhoz képest annyi változtatást kellett alkalmazni, hogy a tábla rajzolásához felhasznált képeket felvettem a `DBuffPanel` osztályba, mint adattagok, illetve a

rajzolás közben esetlegesen felmerülő villogás elkerülése végett dupla pufferrel láttam el. Minden alkalommal, amikor a program a panelt újra rajzolja a főablakon, először, a már előre megadott képek alapján, összeállítja a táblát, utána, ha nem a számítógép következik lépni, akkor felrajzolja a táblára a lehetséges lépéseket szemléltető jeleket, és végül pedig az így kapott táblára felrajzolja



4.5.1 ábra: A program főablaka

a bábukat az aktuális állásnak megfelelően. Abban az esetben, ha nem a számítógép következik lépni az alapértelmezett kurzort egy fekete, áthúzott körre cseréli a program, ezzel is jelezve a felhasználónak, hogy most nem tudja módosítani az aktuális állást. Ellenkező esetben a programnak meg kell határoznia az egér bal gombjának felengedésekor, hogy a játékos melyik mezőbe kattintott. Ha a felhasználó még nem jelölt ki bábut, akkor megnézi, hogy a koordináták olyan mezőbe esnek-e, ahol egy olyan bábu áll, amelyhez tartozik szabályos lépés. Ha igen akkor eltárolja ezt a mezőt, mint kiválasztott mező, és újrarajzolja a táblát csak azon lépéseket megjelenítve, amelyek a kijelölt bábut helyezik át. Ha már volt a játékos által kiválasztott mező akkor megnézi, hogy a kiválasztott mező és a kattintással érintett mező által meghatározott lépés szerepel-e az állapot szabályos lépései között. Ha igen akkor meglépi a lépést és átadja az így keletkezett új állapotot a lépésajánló algoritmusnak, ha nem akkor törli a kiválasztást. Ehhez viszont, mivel a rajzoláshoz szükséges x, y koordináták által meghatározott indexek pont fordítva szerepelnek, mint az állapot `table` mátrixának indexei, szükséges volt, hogy az operátor meg

tudja mondani, hogy a rajzolást tekintve hova került a táblára. Így fel kellett venni e jellemzőt is az operátorok közös tulajdonságai közzé, mint ahogy a 4.1 fejezetben már említettem. A tábla mellett annak érdekében, hogy a felhasználó nyomon tudja követni, hogy milyen lépések voltak az eddigi játszma során, hány lépés van még a döntetlenig, illetve melyik fél hány bábuval rendelkezik, a program szövegesen is megjeleníti az eddig lejátszott játszma adatait. Erre szolgál a tábla mellett található, a .NET keretrendszer beépített `TextBox` osztályával megvalósított szövegdoboz. A játszma kezdetekor a szövegdobozban megjelennek a játszmával kapcsolatos információk úgy, mint a nehézségi fok, és a felhasználó térfele. Minden lépés után a szövegdoboz tartalma frissül olyan információkkal, mint a korábban meglépett lépés sorszáma a lépések sorában, melyik mezőről, melyik mezőre történt a lépés, illetve a táblán lévő bábuk darabszáma. Minden új játszma esetén a szövegdoboz tartalma kiürül, és az új játszma adatait veszi fel. A játékkal kapcsolatos beállításokat, úgy mint új játszma kezdése és térfél váltás, a főablak menürendszerén keresztül végezhetők el. Ugyancsak a menürendszeren keresztül kaphat a felhasználó a játékkal és a program használatával kapcsolatos információkat. A második ablak a lépésajánlások és a lejátszott játszmák alatt összegyűjtött információ alapján készített statisztikák, megjelenítésére szolgál. Ez elsősorban a szakdolgozat elkészítéséhez volt elengedhetetlen, hogy az egyes minimax algoritmus változatok hatékonyságát bemutassam. Ezen ablak megjelenítése a főablak menürendszerén keresztül lehetséges a felhasználó számára. Ugyancsak a főablak menürendszerén keresztül érhető el a harmadik ablak is, amely a szabályokat és a program használatát írja le a felhasználó számára.

Összefoglalás

Az általam megvalósított dáma program a bevezetésben említett célkitűzéseket maradéktalanul megvalósította. A program képes arra, hogy a dáma szabályainak megfelelően játsszon, és elég gyors, ahhoz hogy belátható időn belül egy tíz mélységig tartó lépésajánlással lejátsszon egy játszmát. A programból az ilyen jellegű játékprogramokra jellemző ember-ember elleni játék lehetősége maradt ki. Viszont idő hiányában, és annak tekintetében, hogy ez a funkciója a programnak nem érinti a szakdolgozat fő témáját alkotó mesterséges intelligencia témakörét, ezért nem tartottam fontosnak a program e funkcióját. A programban ugyancsak nincs lehetőség a játszmák közbeni mentésre. Ezen funkciója a programnak azért nem valósult meg, mert maguk a játszmák nem vesznek túl sok időt igénybe, ezért ennek a funkciónak nincs nagy jelentőssége a dáma esetében, valamint ugyancsak nem a mesterséges intelligencia témaköréhez tartozik. A legfontosabb célkitűzés, hogy a program képes legyen egy emberi ellenfél ellen nyerni, ugyancsak megvalósult. A heurisztika véglegesítése után kettő külső személyt kértem fel hogy teszteljék a programomat. Az ő segítségükkel összegyűjtött adatok és statisztikák alapján több száz lejátszott játszma során mindössze kétszer sikerült megverni a programot a legkönnyebb nehézségi fokon. Ezen adat viszont megtévesztő lehet, ugyanis a tesztalanyok csupán amatőr szinten ismerték a játékot. Meggyőződésem hogy a programomnak ahhoz, hogy képes legyen egy profi dámajátékost megverni, még nagyon sok fejlesztésen kellene átesnie. A program továbbfejlesztésének szükségessége a komolyabb ellenfelek ellen, abban is megnyilvánul, hogy mindössze tízmélységű lépésajánlást képes megvalósítani, és már ahhoz is több időre van szüksége, mint amennyire egy versenyszabályoknak megfelelő játszma során lehetősége lenne. Mint ahogy a 3.4 fejezetben is tárgyaltam ráadásul a programnak legalább 15 mélységig kellene előre látnia az állásokat, ahhoz hogy a profi játékosoknak kihívást jelentsen. A heurisztikát tekintve is fejlődnie kellene a programnak, hogy jobban játsszon. Mint azt már az 4.3 fejezetben is említettem a játék számítógépes egyeduralkodója a Chinook program összesen 25 jellemzővel dolgozik, melyek olyan jellemzőket is tartalmaznak, mint a dámává váláshoz vezető szabad utak száma, amely már komolyabb álláselemzést igényel. A program hatékonyságának növelése érdekében ezeken felül még a nyitány könyvtárakkal és végjáték adatbázisokkal való bővítése is szükséges lenne. Ezek implementálása viszont az adatok elemzésének és

összegyűjtésének időigényessége miatt igen lassú folyamat. Ez a fő oka annak, hogy az AutoCheckers program keretében e funkciók nem valósulhattak meg. A dámát tekintve az informatikában már túl sok kutatási téma nem maradt. A Chinook program 2007-ben érte el azt a fejlettségi szintet, hogy elmondhatta magáról, hogy „gyengén" megoldotta a játékot. Ez azt jelenti, hogy a program már a kezdőállásban az adatbázisa segítségével az összes lehetséges játszma pontos heurisztika értékét meg tudja mondani. Ennek az a következménye, hogy minden játszmát legrosszabb esetben is legalább döntetlenre ki tud hozni, vagyis verhetetlen. A hátra maradt lépés a dáma programok fejlesztésében, a dáma „erős" megoldása, tehát egy olyan program elkészítése, amely ugyanarra képes, mint jelenleg a Chinook program, csak tetszőleges állásra nézve. Ez azzal is járna, hogy a játékhoz már meg lehetne pontosan mondani a nyerőstratégiákat. Az olyan kétszemélyes játékok területén, mint a dáma viszont még maradt bőven probléma. A sakk esetében, a játék bonyolultságát figyelembe véve, még messze áll a mesterséges intelligencia, hogy a Chinook jelenlegi tudásával rendelkező programot hozhassanak létre, annak ellenére is, hogy régóta léteznek már olyan sakk programok, melyek képesek profi játékosokat legyőzni. A legnagyobb rejtély a mesterséges intelligencia számára, viszont a távol keletről származó Go. A játék méreteit tekintve az átlagos lépésszámú játszmákra nézve a játékfájának mérete megközelítőleg $3*10^{511}$. A játékfa óriási méretei miatt, a jelenleg ismert lépésajánló algoritmusok már sokkal nagyobb időt vesznek igénybe egy lépés megtalálásához, mint az ugyanebbe a kategóriába tartozó más játékok esetén. Valamint a játékra még senkinek nem sikerült olyan heurisztikus kiértékelő függvényt találnia, amely felírható a 2.2 fejezetben tárgyalt formában. Ezért a jelenleg ötletszerű heurisztikák lassúsága miatt a játék lépésajánló algoritmusai a mélyebb keresések esetén is csak megközelítőleg 100000 nagyságrendű állapotot képesek bejárni. Még mind a mai napig nem sikerült senkinek olyan Go programot írni, amelynek esélye lenne a profibb játékosok ellen akár egy döntetlen kialakítására is. Tehát a Go játék kutatásával a későbbiekben esetleg új heurisztikus elvek, új optimalizációs megoldások, vagy a már meglévő technológiák fejlesztéseinek megjelenése várható a mestersége intelligencia ezen területén.

Irodalomjegyzék

Könyvek

[1] Stuart J. Russell, Peter Norvig: *Mesterséges intelligencia - modern megközelítésben*. Budapest: Panem, 2005.

[2] Fekete István, Gregorics Tibor, Nagy Sára: *Bevezetés a mesterséges intelligenciába*. 3. hasonmás kiad. Budapest: ELTE Eötvös K., 2006.

[3] Futó Iván: *Mesterséges intelligencia*. Budapest: Aula, 1999.

[4] M. Tim Jones: *Artificial intelligence - A system approach*. Hingham, Massachusetts: Infinity Science Press LLC, 2008.

[5] Alison Cawsey: *Mesterséges intelligencia – alapismeretek*. Budapest: Panem, cop.2002

Tanulmányok

[6] Jonathan Schaeffer: *The History Heuristic and Alpha-Beta Search Enhancements in Practice*. Department of Computing Science, University of Alberta, 1989.

[7] Jonathan Schaeffer, Robert Lake: *Solving the Game of Checkers*. Department of Computing Science, University of Alberta, 1996.

[8] J. Schaeffer, Y. Björnsson, N. Burch, A. Kishimoto, M. Müller, R. Lake, P. Lu, S. Sutphen: *Solving Checkers*. Department of Computing Science, University of Alberta, 2005.

Elektronikus jegyzetek

[10] Várterész Magda: *Mesterséges intelligencia 1*. Előadás fóliák, 2006.

[11] Forgó Ferenc, Pintér Miklós, Simonovits András, Solymosi Tamás: *Játékelmélet*. Elektronikus jegyzet, 2005.

Internetes források

[12] http://web.axelero.hu/penkalo/ZDSE_02.htm
Zalaegerszegi Dámajáték Sportegyesület által megadott dáma versenyszabályok.

[13] http://mek.niif.hu/00000/00056/html/133.htm
Dáma játékszabályok.

[14] http://hu.wikipedia.org/wiki/D%C3%A1maj%C3%A1t%C3%A9k
Dáma játékszabályok.

Köszönetnyilvánítás

Szeretném megköszönni Mecsei Zoltán Tanár Úrnak, hogy lehetőséget adott a szakdolgozat megírására, és hogy annak elkészítése alatt tanácsaival és javaslataival segítette munkámat. Köszönetet kell mondanom még Salamon Editnek, aki a szakdolgozat írása alatt mindenben támogatott, valamint a program tesztelésével, hozzájárult annak magvalósulásához. Ugyancsak köszönet illeti Pongor Leventét, hogy hozzájárult a program grafikus felületének megvalósulásához azzal, hogy elkészítette a program által felhasznált, a táblát alkotó, és a bábukat ábrázoló képeket. Valamint szeretném még megköszönni Tóth Attilának, hogy ugyancsak a program tesztelésével hozzájárult annak megvalósításához. Legvégül pedig mindenkinek szeretném megköszönni a támogatást, akik segítettek, hogy a Debreceni Egyetemen folytathassam tanulmányaimat.

Függelék

Pszeudokód alapján megírt $\alpha\beta$ vágás előrendezéssel:

```
private int SearchForStep(CheckersStand Stand)
{
    NoodNo++;
    if (DepthForSearch == Stand.Depth || Stand.Terminal)
    {
        leefNo++;
        //heurisztika érték
        return CheckersStand.HeuristicValue(CheckersStand.Heuristic, Stand);
    }
    //alapértelmezett alfa/béta értékek
    if (Stand.Player == SearchFor) Stand.Value = int.MinValue;
    else Stand.Value = int.MaxValue;

    int value;

    while (Stand.UsableOperators.Count != 0)
    {
        UsedOpNo++;
        Stand.UsableOperators.Sort();  //history heuristic szerinti érték rendezés
        BaseOperator Move = Stand.UsableOperators[0];
        CheckersStand child = Move.UseOperator(Stand);
        Stand.UsableOperators.Remove(Move);
        value = SearchForStep(child);
        if (Stand.Player == SearchFor && Stand.Value < value) //Max szint
        {
            Stand.Value = value;
            Stand.BestMove = Move;
            if (NeedCutBeta(Stand)) break;  //béta vágás
        }
        else if (Stand.Player != SearchFor && Stand.Value > value) //Min szint
        {
            Stand.Value = value;
            Stand.BestMove = Move;
            if (NeedCutAlfa(Stand)) break;  //alfa vágás
        }
    }
    //a legjobb operátor history heuristic értékenek bállítása
    Stand.BestMove.Value += (uint)Math.Pow(2, Stand.Depth - StartDepth);
    return Stand.Value;
}
```