Tartalomjegyzék

1. Bevezetés ... 3

2. Gépi tanulás ... 6

2.1. A gépi tanulás fogalma ... 6

2.2. Input adatok ... 6

2.3. A reprezentációs nyelv ... 8

2.4. Háttérismeretek ... 9

2.5. Keresési stratégia ... 9

2.5.1. Evolúciós algoritmusok ... 10
2.5.2. Genetikus algoritmusok ... 11
2.5.3. Bakteriális algoritmusok ... 12

2.6. A tanulás helyességét mérő értékelő függvény ... 13

2.7. Felügyelet nélküli tanulás ... 14

2.8. Felügyelt tanulás ... 14

3. A megerősítéses tanulás alapfogalmai ... 16

3.1. A megerősítéses tanulás feladata és alapfogalmai ... 16

3.1.1. Az ügynök-környezet modell ... 17
3.1.2. Hozam ... 18
3.1.3. Markov-tulajdonság, Markov döntési folyamat ... 19
3.1.4. Értékelő függvény, Bellman-egyenlet ... 21
3.1.5. Optimális politika és értékelő függvények. A Bellman-féle optimalitási egyenlet ... 23

3.2. A megerősítéses tanulás elemi módszerei ... 25

3.2.1. Dinamikus programozás ... 25
3.2.1.2. A politika javítása ... 27
3.2.1.3. Politika iteráció ... 29
3.2.1.4. Érték iteráció ... 30
3.2.1.5. Általánosított politika iteráció ... 31
3.2.2. Időbeli differenciák módszere ... 32
3.2.2.1. TD jóslás ... 33
3.2.3. A TD(0) módszer optimalitása ... 34
3.2.3.1. SARSA ... 34

3.3 Emlékeztető nyomok módszere 35
3.3.1. n lépéses TD jóslás 35
3.3.2. A TD (λ) előretekintő változata 37
3.3.3. A TD(λ) visszatekintő változata 39
3.3.4. A TD(λ) módszer alkalmazása logikai játékokban 41
3.4. Kapcsolat a függvény-approximátorokkal 42
3.4.1. Értékelő függvény becslése függvény-approximátorokkal 42
3.4.2. Gradiens keresési eljárás 44
3.4.3. Lineáris eljárás 45
4. A minimax kiértékelés 47
5. A megerősítéses tanulás bemutatása egy tanuló amőba játék implementációjában 50
5.1 Az amőba játék története, szabályai 50
5.2 Állapottér és rendszer terv tanuló amőba alkalmazáshoz 51
5.2.1. A környezet modellje 51
5.2.1.1. Az állapottér reprezentációja 51
5.2.2. Az ügynök modellje 52
5.2.2.1. A következő lépés döntésének meghozatala 52
5.2.2.2. Minimax politika 58
5.2.2.3. Az állapot kiértékelő függvény közelítése 59
5.3. A program felhasználói felületének ismertetése 62
5.4. Futási eredmények, összegzés 66
Melléklet 71
1. melléklet Jelölések 71
2. melléklet Hardware és Software-architektúra 73
Irodalomjegyzék 75
Summary 76

1. Bevezetés

1947-ben Arthur L. Samuel, aki akkoriban villamosmérnök professzor volt az Illinois-i Egyetemen, azzal az ötlettel állt elő, hogy ír egy dáma (Checkers) játékot. A dámajáték, amit általában egy, a sakknál egyszerűbb játékként tartanak számon, tökéletes eszköznek bizonyult arra, hogy bemutassa rajta a szimbolikus számítás hatékonyságát egy programozási projekt keretében. A feladat egyszerű volt: „Írjon egy dámajáték programot, hívja ki a dámajáték világbajnokot egy mérkőzésre, és győzzön ellene." Azt remélte, hogy ezzel a kis projekttel felkelti annyira az emberek érdeklődését, hogy az egyetemi számítógépre nyújtott adományok emelkedni fognak.
Samuel ekkor még nem is sejtette, hogy két évtizeden keresztül fog majd dolgozni ezen a programon, aminek következtében nem csupán elkészít egy mesteri fokú játékost (aki megverte kora egyik legjobb amerikai dámajátékosát), hanem fontos ötletekkel is előáll majd a számítógépes játékok és a gépi tanulás terén. Az a két publikáció (Samuel 1959, 1967) amely kutatásait részletezi, a mesterséges intelligencia szakterületének két meghatározó tanulmánya vált. Ezekben a munkákban nemcsak a keresés alapú számítógépes játékprogramok (úgy mint: minimax, alfa-béta nyesés) területén vezet be kiemelkedő fontosságú újításokat, hanem számos olyan tanulási technikát is feltalál, amelyek alkalmasak a program teljesítményének automatikus fejlesztésére. A dámajátékot tökéletes eszköznek találta a gépi tanulás tanulmányozására, mivel a játékokban felmerülő problémák sokszor a valós világ problémáinak leegyszerűsített esetei, így ezen problémák megfigyelése lehetővé teszi a kutatók számára azt, hogy magukra a tanulási problémákra összpontosítsanak. Eredményeinek köszönhetően számos olyan módszer, ami hozzájárult ahhoz, hogy a gépi tanulás tudományos szintre emelkedjék, közvetlenül Samuel nevéhez kötődik, és az ő, tanulással kapcsolatos ötleteinek többsége valamilyen formában még mindig használatban van.
A dámajátékosa először is emlékezett azokra a lépésekre, amelyekkel gyakran találta szemben magát a játék folyamán. Az ismétléses tanulásnak ezen egyszerű formája lehetővé tette számára, hogy időt nyerjen, és hogy a további játékok folyamán behatóbb vizsgálatokat végezhessen minden olyan alkalommal, amikor egy, már ismert állással találkozott a játékmezőn. Egy másik fontos vívmány, amit

a programmal kapcsolatban fontos megemlíteni az, hogy Samuel játékosa vázolta fel elsőként sikeresen annak a tanulási módszernek az alkalmazását, amit ma megerősítéses tanulásnak nevezünk *(reinforcement learning),* és aminek segítségével az értékelési függvény súlyozása beállítható. A program magát fejlesztette, mégpedig úgy, hogy saját maga stabil mintaváltozata ellen játszott. A kiértékelő függvény súlyait minden lépés után beállította, egy a logikai játékokban ma is rendszeresen használt módszer, az *időbeni differenciák* néven ismert módszer egy változata alapján. Samuel programja nem csak beállította a kiértékelő függvény súlyait, hanem folyamatosan megszerkesztette a kiértékelő függvényt, azon paraméterek alapján, amelyek értékesnek bizonyultak az adott játékállás kiértékeléséhez.

Később megváltoztatta a kiértékelő függvényt, és a paramétereket nem lineáris kombinációban ábrázolta, hanem egy olyan rendszerben, ami egy három rétegű neurális hálózatra hasonlított. Ezt a rendszert összehasonlításos tanulással *(comparison training)* fejlesztették, amely tréning mesteri szintű mérkőzések több ezer állásából állt.

Samuel dámajáték programjának kifejlesztése óta a gépi tanulás, és a játék programázás területe sokat gazdagodott, de számos sikeres új módszer, amelyet időközben fejlesztettek ki ezen a szakterületen, közvetlenül az ő ötleteihez kötődik. Az ő dámajátékosát mai napig e két kutatási ág legjelentősebb vívmányai között tartjuk számon, amely egyben tökéletes példaként hozható fel e területek eredményes szimbiózisára.

A számítógépes játékok terén végzett kutatások valóra váltották AI (Artifical Intelligence, Mesterséges Intelligencia) első álmait, vagyis létrejött egy program amelynek segítségével a számítógép megverte az akkori sakkvilágbajnokot egy játszmában, majd egy évvel később versenyt is nyert ellene. (Schaeffer and Plaat 1997, Kasparov 1998).

A Chinook nevű dámajáték-program volt az első, amelyik egy hús-vér világbajnok ellen bármilyen játékban is nyerni tudott. Ezzel, néhány egyszerűbb, népszerű játék rejtélye, úgy, mint a Connect 4, Gomoku, Nine men's morris megoldódott. A programok ma már komoly ellenfelei a legjobb ember játékosoknak, például olyan játékok esetében, mint az ostábla, az othello, vagy a scrabble. Hatékony kutatások folynak más játékokkal kapcsolatban is, úgy mint póker, bridge, shogi illetve Go, de ezekben a gép még nem győzte le az embert.

E diplomamunka célja Samuel nyomán haladva a gépi tanulás, a megerősítéses tanulás vizsgálata egy intelligens amőba játékos implementációján keresztül.

A dolgozat, második fejezetében áttekintjük a gépi tanulás elméletének legfontosabb fogalmait. A harmadik fejezetben a megerősítéses tanulás alapfogalmait és algoritmusait ismertetem. A negyedik fejezetben a teljes információjú kétszemélyes játékok játékfájának minimax kiértékeléséről lesz szó.

Ezen ismeretekre alapozva az ötödik fejezetben bemutatok egy, az Amőba játékos problémán alkalmazott tanítási módszert, illetve a futtatások eredményeit. Megpróbálok a fejlesztési tapasztalatok és futtatási eredmények alapján következtetéseket levonni, és további kutatási irányt meghatározni.

2. Gépi tanulás

2.1. A gépi tanulás fogalma

A ***gépi tanulás*** Samuel óta a mesterséges intelligencia egyik legfőbb kutatási területévé fejlődött. A gépi tanulás egy adaptív eljárás, amely a rendszerben olyan változásokat hoz létre, hogy a következő ciklusban megjelenő azonos problémát a rendszer már helyesebben oldja meg. A tanulási folyamat paraméterek beállítását, illetve csoportosítást, osztályozást, statisztikailag szignifikáns szabályszerűségek feltárását teszi lehetővé a rendszer számára.
A gépi tanulással mára már nem csak kutatóműhelyekben foglalkoznak, hanem valós működő rendszerekben is sikeresen használják. A tudásbázisú rendszerek és az ismeretszerzés automatizálásán túl, a gépi tanulás módszereit gyakran használják statisztikai elemzések támogatásánál az adatok összefoglalásához, az adatokban rejlő, eddig ismeretlen összefüggések leleplezéséhez. Ez a terület az ***adatbányászat*** *(data minining)*, amelyet vezetői döntéstámogató rendszerekhez alkalmaznak. Sokan úgy tartják, hogy ez a terület lehet a leggyorsabban fejlődő kereskedelmi felhasználása a mesterséges intelligencia technikáknak.
A gépi tanulás és az adatbányászat módszereit további gyakorlati problémák megoldására is használják, mint például telekommunikációs hálózatok elemzésére és irányítására, mobil telefonkészülékekkel kapcsolatos csalások felderítésére, hatékony liftvezérlésre, és számos speciális területre is, mint például orvosi diagnosztikára, illetve kifejezésben gazdag zenét játszó gépek megvalósítására. Ezen kívül végül, de nem utolsó sorban, beállíthatjuk ennek segítségével a világ egyik legjobb ostábla játékosának értékelési függvényét.

2.2. Input adatok

Egy tanulási módszer kiválasztása előtt szükséges mind a probléma, mind a rendelkezésre álló adatok elemzése.
A környezet, az input adatok, a példák a gépi tanulás alapját jelentik. Egy tanulási módszer eredményessége nagymértékben a feldolgozandó adatok minőségén és

mennyiségén múlik. Az adatok véges mennyiségben, illetve adatfolyamként folyamatosan állhatnak rendelkezésre.

Hozzáférhető a környezet, ha a tanuló rendszer számára biztosítva van a hozzáférés a környezet teljes állapotához.

Determinisztikus a környezet, ha jelenlegi állapota és az intelligens rendszer cselekedete határozza meg a következő állapotot. Hozzáférhető, determinisztikus környezetben a rendszernek nem kell a bizonytalansággal törődnie.

Epizódszerű a környezet, ha a rendszer észlel, majd cselekszik, de ez a cselekedet független az előzőektől, azaz csak az adott epizódban érvényes.

Ha a környezet megváltozhat azon időszak alatt, míg a rendszer gondolkodik, akkor a környezet ***dinamikus***, egyébként ***statikus***.

Ha létezik az észlelések és cselekvések elkülönülő, világosan definiált véges elemű halmaza, akkor azt mondjuk, hogy a környezet ***diszkrét***.

Ezek alapján a sakk és az amőba például diszkrét, mivel minden egyes lépésben véges számú lehetséges lépés van, hozzáférhető, mivel a környezet elérhető, determinisztikus, hiszen a véletlennek nincsen szerepe, a lépés határozza meg a következő állapotot, valamint epizodikus, és statikus.

A bemenő adatok tisztaságát a következő jellemzők befolyásolják:

- zajosság
- konzisztencia
- teljesség
- magyarázóképesség
- tényszerűség
- számszerűség

A ***zajosság*** azt jelenti, hogy az adatok között van-e hiba, illetve milyen mértékben fordul ez elő.

A ***konzisztencia*** arra utal, hogy az input adatok nem tartalmaznak-e ellentmondásosságot.

A ***teljesség*** azt vizsgálja, hogy rendelkezésre áll-e minden olyan adat, amely szükséges egy cselekvési döntés kialakítására a rendszer számára. Például egy amőba játék esetén látjuk mind saját, mind ellenfelünk játékállását, így képesek

vagyunk döntést hozni a következő lépésről, míg egy kártyajáték esetén ellenfelünk kártyái számunkra rejtve maradnak, és ellenfelünk kártyáinak ismerete elengedhetetlenül fontos lenne egy jó stratégia kialakításához.

Magyarázóképességen azt értjük, hogy az adott jelenség mennyire van hatással más jelenségekre, így ezen jelenségek ismeretében tudunk-e következtetni más eseményekre. A kevésbé fontos eseményeket, amelyek nincsenek nagy hatással a rendszer működésére a rendszer egyszerűsítése érdekében esetleg nem kell figyelembe venni.

Az input adatok ***tényszerűek***, ha mérési eredményként, vagy például diszkrét játékállásokként állnak rendelkezésre. A tényszerű adatok feldolgozása sokkal egyszerűbb.

Az adatok ***számszerűsége*** arra utal, hogy az adatokat olyan numerikus formába kell hozni, hogy azokon mind numerikus, mind logikai műveleteket el lehessen végezni, mivel ezekre a műveletekre szükség van a tanulási folyamat során.

2.3. A reprezentációs nyelv

Az inputadatok, a tanulás eredménye és a háttérismereteket reprezentálása a feldolgozáshoz elengedhetetlen. Elméletileg az inputadatokat lehet egy, a tanulási eredménytől eltérő reprezentációs nyelven ábrázolni, de gyakran egyszerűbb ezeket inkább azonos alapra helyezni. A különböző adatformátumok az információk reprezentálására különböző mértékben felelnek meg. A megfelelő tanulási módszer kiválasztását nagymértékben befolyásolja a módszer által alkalmazott reprezentációs nyelv.

A gépi tanulási módszerek egyik lehetséges reprezentációs nyelve lehet a ***tulajdonság vektorok***. A tanulási eredményt itt egy előre rögzített struktúrájú vektor reprezentálja. Ezt a reprezentációs formát zömmel a mesterséges neurális hálózatok és az evolúciós algoritmusok esetében használják.

Néhány további fogalom tisztázásra szorul a komplett probléma reprezentációval kapcsolatosan:

Az ***állapot***, a probléma lehetséges adatszerkezeteinek értéke.

Az ***állapottér***, az összes lényeges adatszerkezet előforduló értékeinek halmaza.

A ***műveletek***, illetve ***operátorok***, az állapottéren értelmezett transzformációkat jelentik. Meg kell adni a művelet értelmezési tartományát, valamint a műveletek végrehajtásának költségeit. Ez a költség alapértelmezésben egységnyi. Tulajdonképpen a probléma megoldására irányuló egy lépés, amely egyik állapotot a másikba viszi át.
A ***célfeltétel***, az elérni kívánt célállapotok halmazát írja le. Ezek a halmaz általában nem konkrét állapotok felsorolásával, hanem feltételekkel definiáltak. Része az állapottérnek.
A ***kezdőállapotok*** halmaza is része az állapottérnek. Egy sakk játék esetén például a szabályok szerint való elhelyezése világos illetve sötét bábuinak a 8 x 8-as táblán.
A ***megoldás*** egy olyan műveletsorozat, ami elvezet a kezdőállapotból a célállapotba.
Az ***optimális megoldás*** egy minimális költségű megoldás a lehetséges megoldások közül.

2.4. Háttérismeretek

Az ***adat- és modellvezérelt*** tanulási módszerek közötti különbséget az adja, hogy rendelkezésre állnak-e a problémával kapcsolatos háttérismeretek, illetve ezek felhasználhatóak-e a tanulási algoritmus kialakításában. Amennyiben ezeket a háttér-információkat a rendszer felhasználhatja, előre definiált modelleket alkotva modellvezérelt tanulást valósíthat meg. Amennyiben ezek a háttér-információk nem állnak rendelkezésre egy általános, szakterületektől független, adatvezérelt tanulást megvalósító rendszer építhető.

2.5. Keresési stratégia

A tanulás egy fajta optimum keresésként fogható fel egy absztrakt térben. A keresési stratégia kiválasztásánál ezt az absztrakt keresési teret kell elsődlegesen figyelembe vennünk, mivel az egyes keresési algoritmusok például nem alkalmazhatóak végtelen keresési térben, mások pedig elakadhatnak egy lokális optimum pontnál, így a valódi optimumot nem képesek megtalálni.

Az optimalizálás vált mára a számítógép felhasználás egyik legfontosabb területévé. Ennek segítségével végzik a legtöbb olyan számítást, amikor egy rendszer bizonyos kvantitatív tulajdonságát minden rendelkezésre álló információt kihasználva növelni vagy csökkenteni kell mindaddig, míg egy optimális megoldást találunk.
Az optimalizálás létezett már a számítógépek megjelenése előtt is, de bonyolult problémák esetén számítógép nélkül szinte lehetetlen volt a nagy mennyiségű adat feldolgozása az igényelt óriási számítási kapacitás miatt.
A számítógépek elterjedése előtt még csak analitikus módszereket használtak optimalizációra. Ezek közös jellemzője volt, hogy az optimalizálandó rendszerek csak egy bizonyos részhalmazára voltak alkalmazhatóak, mivel csak szigorú megkötésekkel, például differenciálhatóság esetén lehetett a szükséges számításokat végezni. Ide tartoznak a például a gradiens alapú módszerek, amelyek közül talán a legismertebb az úgynevezett ***hegymászó módszer***. Ennek a lényege az, hogy egy adott állapotból lokális tulajdonságok alapján lépünk a legígéretesebb irányba tovább. Ennek a módszernek azonnal látható a gyengesége, ami a lokális szélsőértékekre való érzékenység.
A fenti probléma kiküszöbölésére fejlesztették ki a véletlenen alapuló, úgynevezett *sztochasztikus* módszereket, amely algoritmusok az analitikus megfontolásokon túl egyéb, véletlenszerű hatásokra is képesek a paraméterek változtatásra. Így képes egy hirtelen ugrással kikerülni a lokális szélsőértékből, és folytatni a keresést a végső optimum felé. Ennek a módszernek óriási számításigénye van, ezért csak számítógépekkel, ezeken is főként elosztott, párhuzamos feldolgozással lehet jó eredményeket elérni. Egyik legérdekesebb sztochasztikus keresési módszer a ***szimulált lehűtés***. Ez belső szerkezetében hasonlít a hegymászó módszerre, de véletlen lépéseket is tartalmaz. A véletlen hatások gyakoriságát az idő múlásával csökkentjük, aminek az a várt eredménye, hogy a helyes út kezdeti gyors megtalálása után képes az optimális megoldás pontos behatárolására.

2.5.1. Evolúciós algoritmusok

Az ***evolúciós algoritmusok*** a földi állat- és növényfajok darwini fejlődésének számítástechnikai modelljei, vagyis a biológiai evolúciót utánozó valószínűségi

kereső eljárások. Ezen eljárások lényege, hogy a természetes folyamatokat alapul véve, populációk elkülönülten önállóan fejlődjenek a természetes kiválasztódás szabályai szerint. A populációk egyedei tudnak szaporodni, elpusztulhatnak, illetve itt jön be a sztochasztikus optimalizálásnál megemlített véletlen szerepe, szaporodásuk közben mutáció is felléphet. Ezt a folyamatot kell algoritmusba kódolni, ahol az egyedből struktúra, az eseményből evolúciós operátor lesz.

A populáció egyedei pontokat reprezentálnak a keresési térben. A kiindulási populációt úgy módosítjuk a kiválasztó, keresztező és mutációs eljárásokkal, hogy az egyedek értékének átlaga növekedjen. Ez az érték felel meg a biológiai alkalmasság fogalmának, így az átlag növekedését hasonlóan érjük el, mint ahogy az evolúció működik: a nagyobb értékű, alkalmasabb egyedeknek nagyobb esélyük van arra, hogy megjelenjenek a következő generációban is, illetve az utódok az alkalmasabb egyedek tulajdonságait próbálják egyesíteni. Kellő számú generáció múlva a populáció legjobb egyedei közel hasonlóak lesznek a lehetséges legjobb egyedekhez.

Az alkalmasság mérése általában egyszerűen a célfüggvény értékével történik, tehát arra törekszünk, hogy az egyedek a célfüggvény optimum helyei körül sűrűsödjenek.

2.5.2. Genetikus algoritmusok

A ***genetikus algoritmusok*** evolúciós technikán alapuló eljárások, amelyek futásuk közben a kereszteződéses szaporodást, mutációt, valamint szelektív pusztulás operátort használnak a populációk megváltoztatásához. Ezeket az algoritmusokat a valós, magasabb rendű élőlények életét meghatározó biológiai folyamatok modellezéseként hozták először létre. A megvalósítás elve a következő:

Az egyedek, akárcsak a természetben, populációkat alkotnak, amelyek egymástól részben vagy egészben elzártan létező szaporodási közösségek. Az egyed reprezentációja egy kód, adatszerkezet, amely úgy tárolja az egyed tulajdonságait, mint ahogy a DNS óriásmolekula egy élőlényét. A DNS szaporodás során átörökíthető, az egyed élete során viszont nem változtatható.

A *szelekció* az élővilágban az egyedek létfennmaradásért folytatott küzdelme. A természet szabályai szerint csak a legmegfelelőbb egyedek hozhatnak létre egy

vagy több utódot, a leggyengébb tulajdonságúak még ivarérett koruk elérése előtt elpusztulnak. Ez a folyamat az optimalizációs algoritmusban a rossz tulajdonságú egyedek DNS-ének törlését, a jó tulajdonságúak DNS-ének megtartását, ezen DNS-ek kombinációiból új DNS-ek építését jelenti.

Az egyedek szaporodásuk során, ***kereszteződéssel*** tudnak még megfelelőbb utódot létrehozni, amit mi a kódsorok részleteinek véletlenszerű cseréjével valósítunk meg. Ez a folyamat hasonlít talán a legjobban a valós, az élővilágban létező módszerre, amikor is ivaros szaporodás esetén a két ivarsejt különböző információanyagából az utód DNS-ébe véletlenszerűen kerülnek részletek.

Mutációnak nevezzük egy születő egyed génkészletének véletlenszerű megváltozását, amelyet az algoritmusban a DNS kód véletlenszerű, de jól szabályozott megváltoztatása jelenti. A biológiában, természetes körülmények között a mutáció általában nagyon ritka esemény, de amint például a növénynemesítés is mutatja, mesterségesen felgyorsítható folyamat, s ami régen természetes módon évmilliókig tartott, az ma sokkal rövidebb idő alatt, meghatározott irányba megtehető. Egy számítógépen futó algoritmus esetén ez az idő természetesen még sokkal rövidebb is lehet.

2.5.3. Bakteriális algoritmusok

Az evolúciós technikák másik fajtája a ***bakteriális algoritmusok***. A véletlen szerepe szintén nagyon fontos ebben az eljárásban is, de az egyedek nem kereszteződéses szaporodással, hanem osztódással szaporodnak. A mutáció és a szelekció ugyan úgy működik, mint egyéb más evolúciós technikáknál. Az algoritmus a rendszer megfelelő leírásával kezdődik. A rendszernek olyan paraméterekkel kell rendelkeznie, amelyek jól és egyértelműen jellemzik. Ezeken a paramétereken hajtjuk végre az algoritmust. Az első lépésben létrehozunk véletlenszerű paraméterekkel egy rendszert. Ez a teljes rendszer lesz a kiinduló egyed. Ezután az egyedet lemásoljuk egy előre meghatározott példányban. Ez a baktériumok szaporodásának megfelelő osztódás. Ebben a megközelítésben tehát az utódokat egy egyed hozza létre, eltérően a hagyományos genetikus algoritmustól. A klónozás után minden lemásolt példányra mutációt alkalmazunk. A mutáció itt is a paraméterek véletlenszerű megváltoztatását jelenti. A paraméterek megváltoztatása különböző keretek közé szorítható, lehetőség van

tehát finomhangolásra is, és arra is, hogy csak egy bizonyos paramétert változtassunk meg. A mutáció után valamilyen szempont szerint kiértékeljük az egyedeket, és meghatározzuk közülük a legmegfelelőbbet. Csak ezt az egyetlen kiválasztott egyedet tartjuk meg. Amennyiben a rendszertől egyéb más, például gradiens módszer alapján újabb információkat kapunk, az egyedünket e szerint is megváltoztathatjuk. Ezután megvizsgáljuk, hogy az egyed mennyire ad jó megoldást a feladatra. Ha a megoldás megfelelő, akkor megáll az algoritmus, ha nem megfelelő, akkor újra klónozást és mutációt alkalmazva folytatjuk azt.
Az algoritmus megállításának másik lehetősége, hogy előre meghatározott számú generáció elérésekor állítjuk le a folyamatot. Egyre jobb egyedek jönnek létre, hiszen a mutáció után a legmegfelelőbb egyedet választjuk ki. A klónozás és mutáció után, a szelekciónál az eredeti egyed egyenrangú lesz a többi egyeddel szemben, és a kiértékelés után lehet, hogy sikeres mutáció hiányában újra az eredeti egyed marad életben. Az eljárás folyamatosan közelíti az optimális rendszert, de egy idő után már nem biztos, hogy javulni fog a legjobb egyed. Ennek a beállásnak az ideje, függ attól, hogy hány paraméterrel jellemezzük a rendszert, hogy hány paraméterre alkalmazzuk a mutációt, hogy hány példányt hozunk létre a klónozáskor, valamint a feldolgozás sebességétől, és párhuzamosságától. Amennyiben nagy számú egyedet klónozunk, kisebb az esély a lokális minimumba ragadásra, de a lefutás is lassabb lesz.
Összehasonlítva a két algoritmust, genetikus algoritmus esetében két szülő hoz létre egy utódot, de ezt ivarérett korukban többször is megtehetik, míg a bakteriális eljárás esetében egy szülő egyed hoz létre sok utódot, és amennyiben egyik utódja életképesebbnek bizonyul, saját maga is kihal.

2.6. A tanulás helyességét mérő értékelő függvény

A gépi tanulás nem más, mint az optimális rendszerrel kapcsolatos hipotézisünk folyamatos finomítása. Hogy egy új hipotézis jobb-e, mint az előző, különböző ***értékelő függvények*** segítségével határozhatjuk meg. Ez lehet:

- logikai, például hogy egyes feltétek teljesültek-e,
- statisztikai, például a kedvező eredmények relatív gyakorisága nőtt-e,

- valószínűségi, például a rendszer eredményes döntésének feltételes valószínűsége nőtt-e,
- információelméleti stb.

Az értékelő függvények lehetnek előre rögzítettek, de a rendszer rugalmasságát javíthatja egy szerkezetileg ugyan rögzített, de változtatható paraméterű értékelő függvény.

A változtatható értékelő függvények esetén a rendszer alkalmazkodóképessége ugyan magasabb lehet, azonban nem biztos, hogy a tanulási eljárás során a hipotézisünk folyamatosan iterál az optimális eredményhez.

2.7. Felügyelet nélküli tanulás

Azt a tanulást, amikor nem áll rendelkezésünkre semmilyen adat a rendszer helyes kimenetével kapcsolatosan, ***felügyelet nélküli tanulásnak*** nevezzük *(unsuprvised learning).* Az unsupervised learning eljárásnál a bemenő adatok nem előosztályozottak, a rendszer az adatok ömlesztett feldolgozását végzi.

A felügyelet nélküli tanulás során nem áll rendelkezésre hasznossági függvény, ezért a rendszer nem képes megtanulni, hogy adott helyzetekben mit kéne tennie, csak arra képes, hogy a rendelkezésére álló adatok között összefüggéseket keressen, illetve ezen összefüggések alapján meg tudja jósolni, hogy egy adott állapot milyen következményekkel jár. Egyik tipikus felügyelet nélküli tanulást alkalmazó Mesterséges Intelligencia (MI) tématerület a már említett adatbányászat is.

2.8. Felügyelt tanulás

Minden olyan szituációt, amelyben egy tanuló rendszernek mind a bemenetét, mind a kimenetét észlelni tudjuk, ***felügyelt tanulásnak*** *(supervised learning)* nevezzük. Gyakran a kimeneti információt egy külső tanító adja. A supervised learning tanulási mód feltétele, hogy az input adatok a felhasználó, tanító vagy más eljárás által előosztályozottak legyenek.

Előosztályozottság szerint megkülönböztetünk csak pozitív példákkal, illetve pozitív és negatív példákkal való tanítást.

3. A megerősítéses tanulás alapfogalmai

3.1. A megerősítéses tanulás feladata és alapfogalmai

A példák alapján való tanulás esetében be-kimeneti adatpárok állnak a rendszer rendelkezésére, és a rendszer feladata, hogy egy olyan függvényt tanuljon meg, amely a példaként megadott bemeneti adatokból, az adatpárok kimeneti részeit generálja.

A megerősítéses tanulás esetében a környezettől rendelkezésre álló adatok sokkal kevésbé kényeztetik el a rendszert, mivel a rendszer nem kap példákat, és nem áll rendelkezésére hasznosságfüggvény sem.

Egy tanuló rendszer képes megtanulni sakkozni, vagy más egyéb kétszemélyes teljes információjú logikai játékot professzionális szinten játszani, amennyiben adott játékállásokhoz megmutatja a tanító, mi a számára legjobb lépés. Amennyiben viszont nem áll rendelkezésre tanító, véletlenszerű lépések sorozatával is feltérképezheti a rendszer az állapotteret, és anélkül, hogy az adott pillanatban tudná melyik lépés volt rossz, illetve jó megtanulhatja milyen lépés lenne számára optimális különböző játékállások esetében. Ehhez azonban egyfajta információra, visszajelzésre szüksége van a rendszernek, ez pedig az, hogy ki nyert a játék végén.

Az emberi tanulási folyamatot alapul véve megállapíthatjuk, hogy tudásunk legfőbb forrása a környezetünkkel való kapcsolat, nagyon gyakran cselekedeteinkre is elsősorban környezetünk velünk szemben támasztott elvárásai vannak hatással. A gyermekek felnőtté válásuk során nem kizárólag egy-egy személytől tanulnak, hanem innen-onnan "ellesnek" dolgokat, az egész környezet együttesen alakítja tudásukat. Nincs tehát egyetlen, mindenek felett álló tanító, aki megmondja, hogy mi a jó és mi a rossz, mégis, intelligenciánk révén képesek vagyunk elsajátítani az ok-okozati viszonyokat, a mélyebb összefüggéseket, és azt, hogy hogyan érhetjük el céljainkat.

A következőkben a fent vázolt tanulási módszer egy jelentősen egyszerűsített modelljét tekintjük át. Ismertetem az egyszerűbb tanulási helyzeteket és a megoldásuknál alkalmazható algoritmusokat. A fenti gondolatmenet alapján a ***megerősítéses tanulást*** *(reinforcement learning)* és feladatát úgy fogalmazhatjuk meg, mint egy olyan módszert, amely kapcsolatok alapján, bizonyos célokra

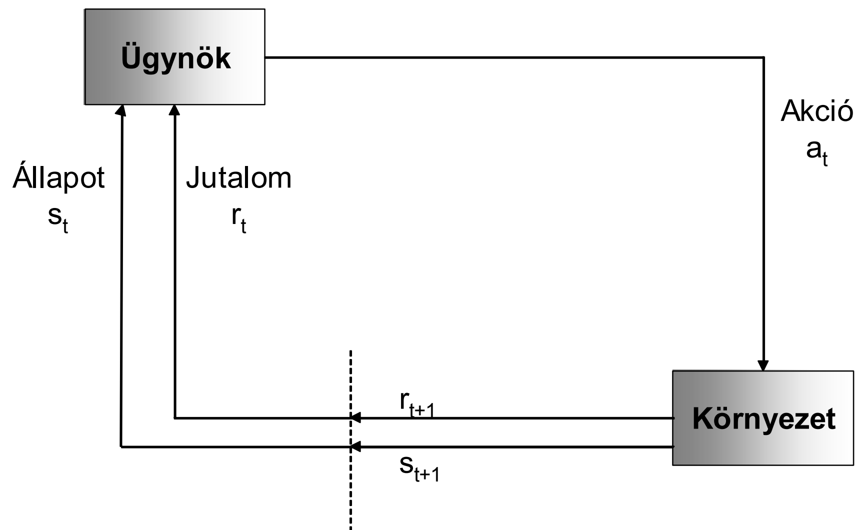
összpontosítva tanul.

3.1.1. Az ügynök-környezet modell

Leegyszerűsített modellünk két alapvető fogalom köré csoportosítható. Ezek közül egyik az aktív döntéshozó fél, amelyet ***ügynöknek*** vagy ***ágensnek*** *(agent)* nevezünk, a másik pedig a vele kapcsolatban álló ***környezet*** *(environment).* Az ügynök és a környezet folyamatos kapcsolatban áll egymással, az ügynök akciókat választ, a környezet pedig ezekre reagál, és új szituációt mutat az ügynöknek. A környezettől érkezik a ***jutalom*** *(reward)* is, amelyet az ügynök maximalizálni kíván. A környezet teljes specifikációja tehát egy megerősítéses tanulás feladatot definiál.

Pontosabban fogalmazva, az ügynök és a környezet diszkrét $t = 0,1,2...$ időpillanatokban kapcsolatba kerül egymással. Minden *t* időpillanatban az ügynök megkapja a környezet $s_t \in S$ állapotleírását, ahol S a lehetséges ***állapotok*** *(state)* halmaza. Ennek alapján választ egy $a_t \in A(s_t)$ akciót, ahol $A(s_t)$ az s_t állapotban megengedett akciók halmaza. A következő lépésben, részben a választott akció függvényeként, kap egy $r_{t+1} \in R \subseteq \mathbb{R}$ jutalmat, és egy új s_{t+1} állapotba kerül. Ennek az interakciónak a szemléletes leírása látható a 1. ábrán.

Minden egyes időpillanatban az ügynök egy leképezést valósít meg az állapotleírások és az egyes akciók választási valószínűségei között. Ezt a leképezést az ügynök ***politikájának*** *(policy)* nevezzük, és π_t-vel jelöljük, ahol $\pi_t(s,a)$ $s_t = s$ esetén $a_t = a$ választásának valószínűségét adja meg. A megerősítéses tanulás különböző módszerei azt írják le, hogy az ügynök hogyan változtatja a politikáját az idő előrehaladtával a tapasztalatai függvényében. Röviden, az ügynök célja, hogy hosszú távon maximalizálja az összegyűjtött jutalmakat.



1. ábra Ügynök-környezet modell a megerősítéses tanulásban

3.1.2. Hozam

A ,,hosszú távon gyűjtött jutalom" fogalmát pontosabban megfogalmazva, az ügynök feladata, hogy a ***várható hozamot*** *(return),* R_t -t maximalizálja, ahol R_t a közvetlen jutalmak sorozatának valamilyen függvénye. A legegyszerűbb esetben a hozam egyszerűen a jutalmak összege:

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + ... + r_T , \qquad (1)$$

ahol T az utolsó időpillanat.

A fenti definíció csak akkor értelmes, ha beszélhetünk az ,,utolsó pillanatról", vagyis ha az ügynök-környezet interakció természetes módon részsorozatokra, úgynevezett ***epizódokra*** *(episode)* bomlik. Ez a helyzet például a kártyajátékok, vagy egy labirintus-feladat esetében. Minden egyes epizód egy úgynevezett *terminális állapotban* ér véget, majd egy kitüntetett kiinduló állapotba, vagy kiinduló állapotok valamelyikébe kerülünk. Néha szükségünk lesz arra, hogy megkülönböztessük egymástól a nem terminális állapotok S halmazát a terminális állapotokkal kiegészített S^+ halmaztól.

Másfelől nagyon gyakran az interakció-sorozat nem bontható értelmes módon epizódokra, gondoljunk csak a folyamatszabályozási vagy robotvezérlési feladatokra. Az ilyen jellegű feladatokat ***folytatható folyamatoknak*** nevezzük. Ekkor a (1)-ben megfogalmazott definíció $T = \infty$ miatt egy végtelen sor alakját ölti,

amelynek az összegét maximalizálni akarjuk. Könnyen lehetséges azonban, hogy a sor divergens, ilyenkor a maximalizálás értelmét veszti. Éppen ezért a (1)-ben adott definíció helyett egy másik, kissé bonyolultabb, de matematikailag jobban kezelhető, definíciót adunk a hozamra, és bevezetjük a ***diszkontált hozam*** fogalmát:

$$R_t = r_{t+1}, + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{i=0}^{\infty} \gamma^i r_{t+i+1} \qquad (2)$$

Itt $0 \le \gamma < 1$ az úgynevezett ***diszkontálási paraméter***. Amennyiben $\gamma < 1$, a (2) sor konvergens, feltéve, hogy a jutalmak r_i sorozata korlátos. Ha $\gamma = 0$, akkor az ügynök ,,rövidlátó" abban az értelemben, hogy csak a közvetlenül következő jutalom maximalizálására törekszik. Ahogy $\gamma \approx 1 (\gamma < 1)$, a későbbi jutalmak egyre nagyobb súllyal jelennek meg, az ügynök egyre inkább ,,előrelátó" lesz.

Az (1) és (2) esetet összefoglalhatjuk egy egységes hozamfüggvény definícióban:

$$R_t = \sum_{i=0}^{T} \gamma^i r_{t+i+1} \qquad (3)$$

Ha T véges, és $\gamma = 0$, akkor az epizodikus, ha $T = \infty$ és $0 \le \gamma < 1$, akkor pedig a folytatható folyamatra megadott definíciót kapjuk.

3.1.3. Markov-tulajdonság, Markov döntési folyamat

A megerősítéses tanulás előzőleg már ismertetett ügynök-környezet modelljében az ügynök döntései a környezet által biztosított állapotjeltől függenek. Az állapot leírása tartalmazza az aktuális szenzoros információkat, de ezen kívül más is szerepelhet benne. Bonyolult módon függhet az előző megfigyelések sorozatától, de az is lehet, hogy az aktuális információ az egész rendszer karakterisztikáját, lényegében az átmenet-valószínűségeket anélkül írja le, hogy a megelőző megfigyeléseket figyelembe venné. Az ilyen rendszereknek, ahol tehát csak a közvetlenül megelőző állapotot kell figyelembe venni, ***Markov-tulajdonsága*** van.

Az alábbiakban megadom a Markov-tulajdonság formális definícióját. Az egyszerűség kedvéért tegyük fel, hogy $|S| < \infty,$ és $|R| < \infty$. A legáltalánosabb esetben a környezet dinamikája csak az összes előző időpillanat figyelembevételével írható le:

$$P(s_{t+1} = s', r_{t+1} = r \mid s_t, a_t, r_t, s_{t-1}, a_{t-1}, r_{t-1} ..., r_1, s_0, a_0 \mid) \qquad (4)$$

minden $(s',r) \in S \times R$-re, és minden lehetséges múltbeli $s_t, a_t, r_t, ..., r_1, s_0, a_0$ esetén. Ellenben, ha a rendszer rendelkezik a Markov-tulajdonsággal, akkor a környezet dinamikáját a következőképpen írhatjuk fel:

$$P_r(s_{t+1} = s', r_{t+1} = r \mid s_t, a_t \mid), \qquad (5)$$

Más szóval az állapotleírás pontosan akkor Markov-tulajdonságú (Markov-állapottér), ha a (4) egyenlő (5)-tel.

Ha a környezet rendelkezik a Markov-tulajdonsággal, akkor (5) alapján megadhatjuk a várható új állapotot és a jutalom értékét. Sőt, ennél több is igaz. A (5) iterálásával látható, hogy minden jövőbeli állapotot és jutalmat előre megbecsülhetünk. Azt is észrevehetjük, hogy az akcióválasztás is Markov-állapotok esetén a legkönnyebb. Ekkor ugyanis Markov-állapottól függő politika éppen olyan jó, mint amelyik az egész előző történetet figyelembe veszi.

Azt a megerősítéses tanulási folyamatot, amely kielégíti a Markov-tulajodnságot, ***Markov döntési folyamatnak*** *(Markow Decision Process, MPD)* nevezzük. Amennyiben az állapotok és akciók halmaza véges, a Markov döntési folyamat véges. A véges Markov döntési folyamatot megadhatjuk az állapotok és az akciók halmazával, továbbá az egylépéses környezeti dinamikával. Az átmenet-valószínűség megadja az s' állapotba kerülés valószínűségét s állapotból a akció választása mellett:

$$P_{ss'}^{a} = P_r(s_{t+1} = s' \mid s_t = s, a_t = a) \qquad (6)$$

Hasonlóképpen a várható jutalom s-ből s'-be kerüléskor a akció választása mellett:

$$R_{ss'}^{a} = E(r_{t+1} \mid a_t = a,\ s_t = s,\ s_{t+1} = s') \qquad (7)$$

$P_{ss'}^{a}$ és $R_{ss'}^{a}$ teljesen leírják a véges Markov döntési folyamatot.

A Markov-tulajdonság és a Markov döntési folyamat áttekintése és megértése azért volt szükséges, mert azokra a feladatokra alkalmazható bizonyítottan a megerősítéses tanulás módszere, amelyre fennáll a Markov-tulajdonság, a rendszer teljesen információjú, azaz az állapotteret teljesen ismerjük. Fontos még az állapottér végessége is.

3.1.4. Értékelő függvény, Bellman-egyenlet

Majdnem mindegyik megerősítéses tanulás módszer ***értékelő függvények*** becslésén alapul. Az értékelő függvény rendszerint az állapotok, vagy az állapot-akció párok függvénye, és azt írja le, hogy mennyire jó egy adott állapot (vagy mennyire jó egy adott állapotban egy adott akciót végrehajtani). A ,,jóság" a hozam várható értékével függ össze. A hozam viszont természetesen függ az ügynök döntéseitől, tehát az ügynök politikájától. Informálisan, egy s állapot π politika melletti ,,jósága", amelyet a továbbiakban $V^{\pi}(s)$-sel jelölünk, az állapotból a politika követése mellett gyűjthető hozam várható értéke. Markov döntési folyamat esetén az úgynevezett ***állapotot értékelő függvény*** formálisan a következő alakban írható fel:

$$V^{\pi}(s) = E_{\pi}(R_t \mid s_t = s) = E_{\pi}\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s \right) \qquad (8)$$

Itt E_{π} jelöli a π politika követése melletti várható értéket, t tetszőleges időpillanat. Megjegyezzük, hogy amennyiben az állapottérben szerepelnek terminális állapotok, azok értéke mindig 0.

Hasonlóképpen, megadjuk az s állapotban a akció választásának értékét a π politika mellett. Ezt $Q^{\pi}(s,a)$-val jelöljük, és azt adja meg, hogy mennyi a hozam várható értéke, ha az s állapotban vagyunk, ahol az a akciót választjuk, majd a π politikát követjük. Formálisan:

$$Q^{\pi}(s,a) = E_{\pi}(R_t \mid s_t = s, a_t = a) = E_{\pi}\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s, a_t = a \right) \qquad (9)$$

Q^{π} elnevezése a π politika melletti ***akciót értékelő*** függvény.

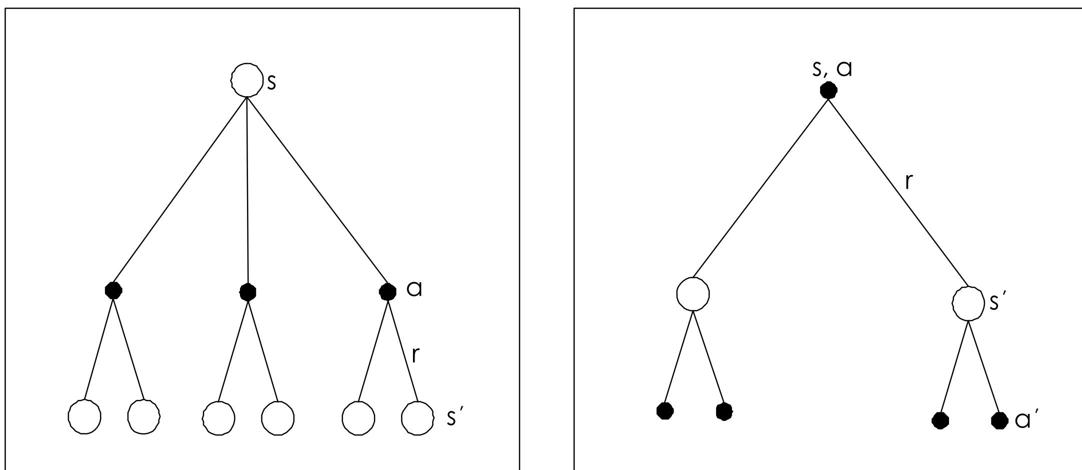
Az állapotot és akciót értékelő függvény fontos tulajdonsága, hogy kielégít egy rekurzív kapcsolatot. Minden π politikára és minden s állapotra, az s és *a* lehetséges rákövetkező állapot értéke között fennáll a következő konzisztencia kapcsolat:

$$\begin{aligned}
V^{\pi}(s) &= E_{\pi}(R_t | s_t = s) \\
&= E_{\pi}\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s \right) \\
&= E_{\pi}\left(r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s \right) \\
&= \sum_a \pi(s,a) \sum_{s'} P_{ss'}^a \left[R_{ss'}^a + \gamma E_{\pi}\left(\sum_0^{\infty} \gamma^k r_{t+k+2} | s_{t+1} = s' \right) \right] \\
&= \sum_a \pi(s,a) \sum_{s'} P_{ss'}^a \left[R_{ss'}^a + \gamma V^{\pi}(s') \right],
\end{aligned} \tag{10}$$

ahol implicit módon feltettük, hogy $a \in A(s)$ és $s' \in S$ (epizodikus folyamatok esetén $s' \in S^{+}$).

A (10) egyenlet a V^{π}-re vonatkozó ***Bellman-egyenlet***, amely a jelenlegi állapot és a lehetséges rákövetkező állapotok értéke közötti összefüggést írja le. Megmutatható, hogy a Bellman-egyenletnek V^{π} állapotot értékelő függvény az egyedüli megoldása, azaz fixpontja.

A 2.(a). és 2.(b). ábrák a Bellman-egyenletben szereplő rekurzív kapcsolat szemléletes megjelenítései. Az ábra értelmezése a következő: az üres körök az állapotokat, a teli körök pedig az állapot-akció párokat szemléltetik.



(a) A V^{π}-t meghatározó felösszegzési gráf *(b) A Q^{π}-t meghatározó felösszegzési gráf*

2. ábra Az értékelő függvényekhez tartozó felösszegzési gráfok.

3.1.5. Optimális politika és értékelő függvények. A Bellman-féle optimalitási egyenlet

Egy megerősítéses tanulás feladat megoldása egy olyan politika megkeresését jelenti, amely követése mellett hosszútávon sok jutalmat gyűjthetünk. Az értékelő függvények a politikák terén egy részberendezést definiálnak a következőképpen: egy π politika pontosan akkor legalább olyan jó, mint egy π' politika, ha a π politikát követve minden egyes állapotban a várható hozam nagyobb vagy egyenlő, mint π' politika esetén: $\pi \geq \pi'$, ha $\forall s \in S$ esetén $V^{\pi}(s) \geq V^{\pi'}(s)$. Ekkor létezik legalább egy olyan politika, amely legalább olyan jó, mint az összes többi. Ezt a politikát ***optimális politikának*** nevezzük. Bár az optimális politika nem egyértelmű, mégis a közös π^* jelölést alkalmazzuk az összes optimális politikára. Egyszerűen meggondolható, hogy ezekhez ugyanaz az optimális állapotot értékelő függvény tartozik, amelyet V^*-gal jelölünk:

$$V^*(s) = \max_{\pi}(V^{\pi}(s)), \forall s \in S \tag{11}$$

Természetesen közös optimális akciót értékelő függvényük is:

$$Q^*(s,a) = \max_{\pi}(Q^{\pi}(s,a)), \forall (s,a) \in S \times A(s) \tag{12}$$

Más szóval, $Q^*(s,a)$ azon hozam várható értékét adja, amelyet úgy kapunk, hogy az s állapotban az a akciót választjuk, majd utána az optimális π^* politikát követjük. Ezért Q^* felírható V^* segítségével a következőképpen:

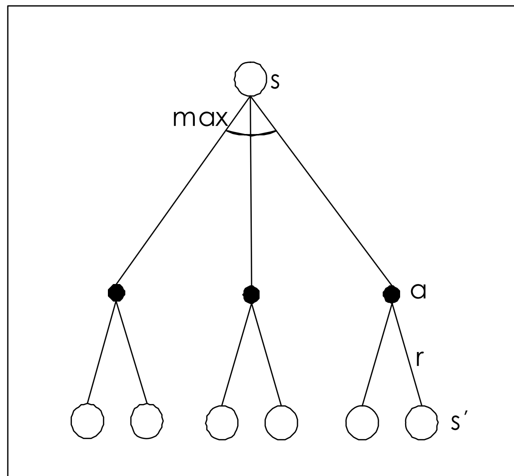
$$Q^*(s,a) = E(r_{t+1} + \gamma V^*(s_{t+1}) | s_t = s,\ a_t = a \tag{13}$$

Mivel V^* egy létező politika értékelő függvénye, így kielégíti a (10)-ben adott Bellman-egyenletet. Ugyanakkor, mivel V^* az optimális állapotot értékelő függvény, a Bellman-egyenletben szereplő rekurzív kapcsolat felírásához nem kell egy konkrét politikára hivatkoznunk:

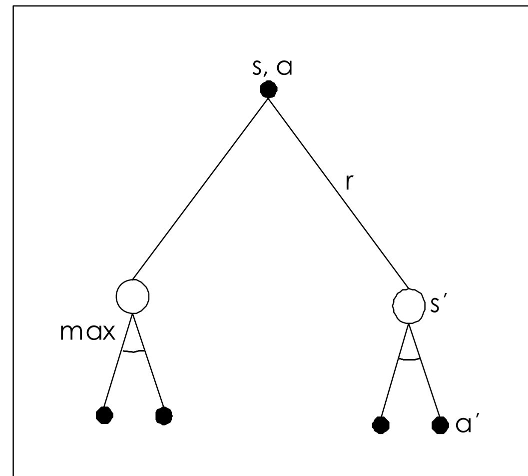
$$
\begin{aligned}
V^*(s) &= \max_a Q^{\pi^*}(s,a) \\
&= \max_a E_{\pi^*}(R_t | s_t = s, a_t = a \\
&= \max_{a\in A(s)} E_{\pi^*}\left(\sum_{k=0}^{\infty}\gamma^k r_{t+k+1} | s_t = s, a_t = a\right) \\
&= \max_{a\in A(s)} E_{\pi^*}\left(r_{t+1} + \gamma\sum_{k=0}^{\infty}\gamma^k r_{t+k+2} | s_t = s, a_t = a\right) \\
&= \max_{a\in A(s)} E_{\pi^*}\left(r_{t+1} + \gamma V^*(s_{t+1}) | s_t = s, a_t = a\right) \\
&= \max_{a\in A(s)} \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + \gamma V^*(s')\right]
\end{aligned}
\tag{14}
$$

A fenti egyenlet a V^*-ra vonatkozó Bellman-féle optimalitási egyenlet. Hasonlóképpen felírhatjuk (14) megfelelőjét Q^*-ra is:

$$
\begin{aligned}
Q^*(s,a) &= E(r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') | s_t = s, a_t = a \\
&= \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + \gamma \max_{a'} Q^*(s', a')\right]
\end{aligned}
\tag{15}
$$



(a) A V^*-t meghatározó felösszegzési gráf



(b) A Q^*-t meghatározó felösszegzési gráf

3. ábra Az optimális értékelő függvényekhez tartozó felösszegzési gráfok (Backup)

Véges Markov döntési folyamat esetén a (14) egyenletnek egyértelműen létezik a politikától független megoldása. A Bellman-féle optimalitási egyenlet tulajdonképpen egy egyenletrendszer, minden állapothoz tartozik egy-egy egyenlet. Tehát, ha az állapotok száma N, akkor van N darab egyenletünk és N

darab ismeretlenünk. Így abban az esetben, ha a környezet dinamikája $\left(P_{ss'}^{a}, R_{ss'}^{a}\right)$ ismert, bármely, nemlineáris egyenletrendszer megoldására felhasználható módszerrel elvileg megoldhatjuk a Bellman-féle optimalitási egyenletet.

V^* ismeretében egy optimális politika megtalálása egyszerű feladat. Minden s állapothoz létezik egy vagy több olyan a akció, amelyre a (14) egyenletben szereplő maximumot elérjük. Bármely politika, amely ezekhez és csak ezekhez az akciókhoz rendel pozitív kiválasztási valószínűséget, optimális. Más szóval, az a politika, amely V^*-ra tekintve mohó, optimális politika is egyben.

A Q^* ismeretében az optimális politika megkeresése még egyszerűbb: minden s állapotra egy olyan akciót kell választani, amely maximalizálja $Q^*(s,a)$-t.

3.2. A megerősítéses tanulás elemi módszerei

Ebben a fejezetben a megerősítéses tanulás azon alapvető módszereit tekintjük át, amelyek a legnagyobb jelentőséggel bírnak mind elméleti, mind gyakorlati szempontból.

3.2.1. Dinamikus programozás

A dinamikus programozás elnevezés azoknak az algoritmusoknak a gyűjteményére vonatkozik, amelyek optimális politikák megkeresésére használhatók, feltéve, ha a környezetről egy teljes Markov döntési folyamat modell áll a rendelkezésünkre. Mivel ilyen erős megkötést igényelnek, és számítási költségük is túl nagy, a dinamikus programozás módszerei a gyakorlatban nem használatosak, elméleti jelentőségük miatt mégis érdemes áttekinteni őket.

A fejezetben azzal a feltevéssel élünk, hogy a környezet véges Markov döntési folyamat. Meg kell jegyeznünk, hogy a dinamikus programozás módszerei folytonos állapot és akcióterű problémák esetén is alkalmazhatók, a pontos megoldás azonban csak speciális esetekben állítható elő.

A dinamikus programozás, és általában a megerősítéses tanulás alapötlete, hogy a jó politikákat az értékelő függvények alapján keressük. Ebben a részben megnézzük, hogyan alkalmazható a dinamikus programozás az előző fejezetben definiált értékelő függvények meghatározására. Amint azt már láttuk, a Bellman-

féle optimalitási egyenletet kielégítő V^* vagy Q^* esetén egyszerűen meghatározható az optimális politika. A dinamikus programozás algoritmusait úgy kaphatjuk meg, hogy a különféle Bellman-egyenleteket az értékelő függvény közelítését javító felülírási szabályokká alakítjuk.

3.2.1.1. Politika kiértékelése

Először azt nézzük meg, hogyan készíthetjük el egy tetszőleges π politika állapotot értékelő függvényét. A dinamikus programozás terminológiájában ezt *politika kiértékelésnek* nevezik. Emlékezzünk az előző részben megadott Bellman-egyenletre:

$$\begin{aligned} V^{\pi}(s) &= E_{\pi}\left(r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots \middle| s_t = s\right) \\ &= E_{\pi}\left(r_{t+1} + \gamma V^{\pi}(s_{t+1}) \middle| s_t = s\right) \\ &= \sum_a \pi(s,a) \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + \gamma V^{\pi}(s')\right], \end{aligned} \tag{16}$$

ahol $\pi(s,a)$ annak a valószínűsége, hogy a π politikát követve az s állapotban az a akciót választjuk. A várható értéket π-vel indexeltük annak jelzésére, hogy ez tulajdonképpen a követett π politikától függő feltételes várható érték. V^{π} egyértelmű létezése garantált, feltéve, hogy $\gamma < 1$ vagy a π politika mellett minden kezdőállapotból véges sok lépés múlva terminális állapotba jutunk.

Ha a környezet dinamikája teljesen ismert, akkor a fenti egyenlet $|S|$ egyenletből álló, ugyanennyi ismeretlenes lineáris egyenletrendszer, amely egyszerűen, bár viszonylag nagy számítási költséggel, explicit módon megoldható. A mi céljainknak inkább az iteratív megoldási módszerek felelnek meg. Tekintsük közelítő állapotot értékelő függvények egy $V_0, V_1, V_2, \ldots : S^+ \to R$ sorozatát. A kiinduló V_0 tetszőleges, csak az esetleges terminális állapotoknak kell 0 értékkel rendelkezniük, és minden következő tagot a (10)-ben adott Bellman-egyenlet alapján kapunk a következőképpen:

$$\begin{aligned} V_{k+1}(s) &= E_{\pi}(r_{t+1} + \gamma V_k(s_{t+1}) \mid s_t = s) \\ &= \sum_a \pi(s,a) \sum P^a_{ss'}\left[R^a_{ss'} + \gamma V_k(s')\right] \end{aligned} \tag{17}$$

$\forall s \in S$-re. Nyilvánvalóan a $V_k = V^{\pi}$ a fenti egyenlet fixpontja, hiszen ekkor a V^{π}-re vonatkozó Bellman-egyenlet garantálja az egyenlőséget. Sőt, meg lehet mutatni,

hogy $\{V_k\}$ konvergál V^π-hez, ha $k \to \infty$, mégpedig ugyanazon feltételek mellett, amelyek V^π létezését garantálják. Ezt az algoritmust ***iteratív politika kiértékelésnek*** nevezzük.

V_k-ból úgy kapjuk a $V_k + 1$-et, hogy minden s állapot értékét felülírjuk egy új értékkel, amelyet az s rákövetkező állapotainak régi értékéből, valamint a kiértékelt π politika melletti azonnali várható jutalmakból határozunk meg. Ezt az eljárást ***teljes felösszegzésnek*** nevezzük, mivel minden egyes s állapot értékét felülírjuk a következő $V_k + 1$ közelítés meghatározásához.

Az alábbi táblázatban megadjuk az iteratív politika kiértékelés algoritmusát:

Input: π policy to be evaluated
Initalize: $V(s) = 0 \forall s \in S^+$
Repeat
$\Delta \leftarrow 0$
For each $s \in S$
$v \leftarrow V(s)$
$V(s) \leftarrow= \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V^\pi_k(s') \right]$
$\Delta \leftarrow \max(\Delta, |v - V(s)|)$
Until $\Delta < \theta$ (small positive number)
Output: $V \approx V^\pi$

1. táblázat. Az iteratív politika kiértékelés algoritmusa

3.2.1.2. A politika javítása

A célunk V^π kiszámításával az volt, hogy jobb politikákat tudjunk keresni. Tegyük fel, hogy meghatároztuk V^π-t egy tetszőleges determinisztikus π politikára. Valamely s állapotban azt szeretnénk eldönteni, hogy érdemes-e megváltoztatni a politikát oly módon, hogy egy $a \neq \pi(s)$ akciót választunk. Egy lehetőség ennek megválaszolására, hogy az s állapotban az a akciót választjuk, majd a π politikát követjük. Az ilyen viselkedés értéke:

$$
\begin{aligned}
Q^\pi(s,a) &= E_\pi\{r_{t+1} + \gamma V_k(s_{t+1}) \mid s_t = s, a_t = a\} \\
&= \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V^\pi_k(s') \right]
\end{aligned}
\tag{18}
$$

A kulcskérdés, hogy ez vajon kisebb vagy nagyobb-e, mint $V^{\pi}(s)$. Amennyiben nagyobb, akkor egy jobb politikát kaptunk, mintha végig a π politika szerint cselekedtünk volna. A fenti az ***általános politika javítás*** egy speciális esete. Igaz ugyanis a következő tétel:

Tegyük fel, hogy π és π' determinisztikus politikák, amelyekre igaz az alábbi:

$$Q^{\pi}(s,\pi'(s)) \geq V^{\pi}(s), \forall s \in S \tag{19}$$

Ekkor a π' politika legalább olyan jó, mint π, azaz:

$$V^{\pi'}(s) \geq V^{\pi}(s) \tag{20}$$

Továbbá, ha a (19)-ben bármely állapotra szigorú egyenlőtlenség teljesül, akkor van olyan állapot, amelynél (20)-ben is szigorú egyenlőtlenség áll.

A bizonyítás alapötlete nagyon egyszerű: kiindulunk a (19) egyenlőtlenségből, és a Q^{π} oldalát kifejtjük (19) ismételt alkalmazásával:

$$\begin{aligned}
V^{\pi}(s) &\leq Q^{\pi}(s,\pi'(s)) \\
&= E_{\pi'}\{r_{t+1} + \gamma V^{\pi}(s_{t+1}) \mid s_t = s\} \\
&\leq E_{\pi'}\{r_{t+1} + \gamma Q^{\pi}(s_{t+1},\pi'(s_{t+1})) \mid s_t = s\} \\
&= E_{\pi'}\{r_{t+1} + \gamma E_{\pi'}\{r_{t+2} + \gamma V^{\pi}(s_{t+2})\} \mid s_t = s\} \\
&= E_{\pi'}\{r_{t+1} + \gamma r_{t+2} + \gamma^2 V^{\pi}(s_{t+2}) \mid s_t = s\} \\
&\leq E_{\pi'}\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 V^{\pi}(s_{t+3}) \mid s_t = s\} \\
&\vdots \\
&\leq E_{\pi'}\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + ... \mid s_t = s\} \\
&\leq V^{\pi'}(s)
\end{aligned} \tag{21}$$

Ezzel az állítást bebizonyítottuk.

Az eddigiekben láttuk, hogy egy adott állapotban a politika megváltozása hogyan hat az akcióválasztásra. Ezen eljárás természetes kiterjesztése, hogy az aktuális politika követése helyett minden állapotban azt az akciót válasszuk, amely $Q^{\pi}(s,a)$ alapján a legjobbnak tűnik, azaz megadhatjuk az alábbi mohó politikát:

$$\begin{aligned}\pi'(s) &= \arg\max_a Q^{\pi}(s,a) \\ &= \arg\max_a E\{r_{t+1} + \gamma V^{\pi}(s_{t+1}) \mid s_t = s,\ a_t = a\} \\ &= \arg\max_a \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + \gamma V^{\pi}_k(s')\right]\end{aligned} \qquad (22)$$

A mohó politika azt az akciót választja, amely V^{π} alapján egy lépésre előretekintve a legjobbnak tűnik. A konstrukcióból adódóan a mohó politika teljesíti az általános politika javítás tétel feltételeit, így tudjuk, hogy legalább olyan jó, mint az eredeti politika. A folyamatot, amely során politikánkat úgy módosítjuk, hogy mohóvá tesszük az állapotot értékelő függvényre vonatkozóan, politika javításnak nevezzük.

Tegyük fel, hogy az új π' politika nem jobb, mint az eredeti π. Ekkor $V^{\pi} = V^{\pi'}$, és a (22) alapján $\forall s \in S$-re:

$$\begin{aligned}V^{\pi}{}'(s) &= \max_a E\{r_{t+1} + \gamma V^{\pi'}(s_{t+1}) \mid s_t = s,\ a_t = a\} \\ &= \max_a \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + \gamma V^{\pi'}(s')\right]\end{aligned} \qquad (23)$$

ami viszont ugyanaz, mint a (14)-ban adott Bellman-féle optimalitási egyenlet. Így $V^{\pi'} = V^{*}$, π és π' optimális politikák. Következésképpen, a politikajavítás szigorúan jobb politikát ad, kivéve, ha a kiinduló politika már maga is optimális.

Az eddigiekben feltettük, hogy a vizsgált politikák determinisztikusak, azonban a bemutatott eredmények általánosíthatók sztochasztikus politikákra is.

3.2.1.3. Politika iteráció

Ha egy π politikát javítottunk V^{π} alapján, és egy jobb π' politikát kaptunk, akkor meghatározhatjuk $V^{\pi'}$-t, és ez alapján egy még jobb π'' politikához juthatunk. Tehát előállíthatjuk politikák és értékelő függvények monoton növő sorozatát:

$$\pi_0 \xrightarrow{E} V^{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V^{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \ldots\ldots \xrightarrow{I} \pi^* \xrightarrow{E} V^{\pi^*},$$

ahol E a politika kiértékelését, I pedig a politika javítását jelöli. Minden következő politika szigorúan jobb az előzőnél, hacsak nem optimális politika. Mivel egy véges Markov döntési folyamat esetében csak véges sok politika létezik, a fenti eljárás véges sok iterációs lépésben az optimális politikához és értékelő függvényhez konvergál.

Az optimális politika ilyen módon történő megkeresése a *politika iteráció*. Az alábbi táblázatban a politika iteráció algoritmusát adom meg:

1. Initalize: $V(s) \in R \ and \ \pi(s) \in A(s) \ for\,all \ s \in S$ 2. Policy evaluation Repeat $\Delta \leftarrow 0$ For each $s \in S$ $v \leftarrow V(s)$ $V(s) \leftarrow= \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V^\pi_k(s') \right]$ $\Delta \leftarrow \max(\Delta, \lvert v - V(s) \rvert)$ Until $\Delta < \theta$ (small positive number) 3. Policy improvement Policy_is_stabil $\leftarrow$ true For each $s \in S$ $b \leftarrow \pi(s)$ $\pi(s) \leftarrow \max_a \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V^\pi(s') \right]$ If $b \neq \pi(s)$, Policy_is_stabil $\leftarrow$ false If Policy_is_stabil then stop; else go to 2.

2. táblázat. Politika iteráció a V*-ra

3.2.1.4. Érték iteráció

A politika iteráció egy nagy hátránya, hogy minden egyes politika kiértékelés igen nagy számítási költségű lehet, és az állapottér többszöri átolvasását igényelheti. Szerencsére, a konvergencia megtartása mellett csökkenthető a politika kiértékelés lépésszáma, és megállhatunk az állapottér egyszeri átolvasása után is. Ezt az algoritmust *érték iterációnak* nevezzük, és egyszerűen felírhatjuk a politikajavítás és a módosított politika kiértékelés figyelembevételével:

$$\begin{aligned} V_{k+1}(s) &= \max_a E\{r_{t+1} + \gamma V_k(s_{t+1}) \mid s_t = s, \ a_t = a\} \\ &= \max_a \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V_k(s') \right], \forall s \in S \end{aligned} \tag{24}$$

Tetszőleges V_0 mellett igazolható, hogy a $\lceil V_k \rceil$ sorozat V^*-hoz tart, mégpedig ugyanazon feltételek mellett, amelyek V^π létezését garantálják.

Az érték iteráció algoritmusa:

Initalize: V is arbitrarily, expect $s \in S^+$, where V(s)=0 Repeat $\Delta \leftarrow 0$ For each $s \in S$ $v \leftarrow V(s)$ $V(s) \leftarrow= \max_a \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V(s') \right]$ $\Delta \leftarrow \max(\Delta, \| v - V(s) \|)$ Until $\Delta < \theta$ (small positive number) Output: deterministic π policy $\pi(s) = \arg\max_a \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V(s') \right]$

3. táblázat. Érték iteráció

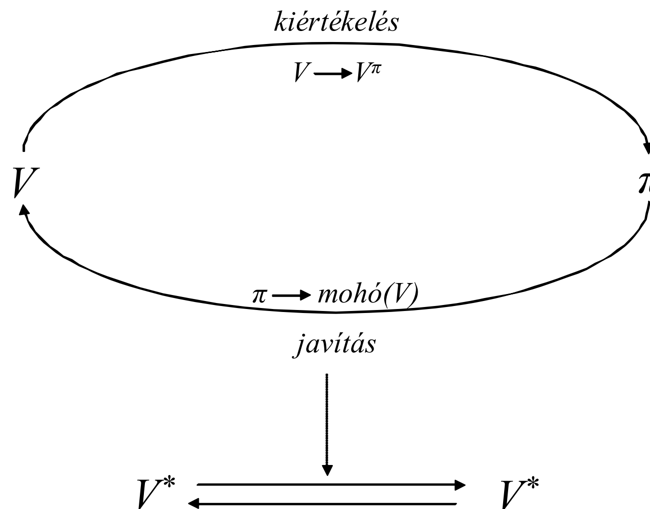
3.2.1.5. Általánosított politika iteráció

A politika iteráció két szimultán, egymással kölcsönhatásban lévő folyamatból áll: az egyik az értékelő függvényt szinkronizálja az aktuális politikával (politika kiértékelés), míg a másik a politikát mohóvá teszi az értékelő függvényre nézve (politikajavítás). A politika iterációnál ez a két folyamat szigorúan váltakozva, egymás után működik, de ez nem szükségszerű. Az érték iterációnál például már egyetlen politika kiértékelési lépés után javítjuk a politikát, sőt ennél még finomabban is feloszthatjuk a működést. Amíg mindkét folyamat minden állapotot frissít, a végeredmény ugyanaz: konvergencia az optimális értékelő függvényhez és politikához.

Az ***általánosított politika iteráció*** elnevezést használjuk akkor, ha a politika kiértékelő és a politika javító folyamat kölcsönhatására utalunk, függetlenül az interakció ,,finomságától", és a folyamatok egyéb részleteitől. Majdnem minden megerősítéses tanulás feladat leírható, mint általánosított politika iteráció: létezik meghatározott politika és értékelő függvény. A politikát mindig az értékelő függvény alapján javítjuk, az értékelő függvényt pedig a politika értékelő függvényének irányába módosítjuk. Ezt az általános sémát láthatjuk a 4. ábrán.

Könnyen látható, hogy ha a kiértékelő és a javító folyamat stabilizálódik, azaz nem végez módosításokat, akkor a kapott értékelő függvény és politika optimális. Ugyanis az értékelő függvény akkor stabilizálódik, ha konzisztens az adott

politikával, a politika pedig akkor, ha mohó az értékelő függvényre nézve. Mindkét folyamat stabilizálódása tehát azt jelenti, hogy találtunk egy politikát, amely mohó a saját értékelő függvényére nézve. Ebből következik, hogy teljesül a (14)-ban adott Bellman-féle optimalitási egyenlet, azaz a politika és az értékelő függvény optimális.



4. ábra. Általánosított politika iteráció

Az általánosított politika iteráció kiértékelő és javító folyamatai egyszerre együttműködők és egymással versengők. Versengők abban az értelemben, hogy ellentétes irányba hatnak. A politika mohóvá tételével az értékelő függvény inkonzisztens lesz a megváltozott politikára nézve, az értékelő függvény konzisztenssé tétele pedig rendszerint a politika mohóságát rontja el. Ugyanakkor hosszútávon ez a két konvergáló folyamat együttműködik az optimális politika és értékelő függvény megtalálásában.

3.2.2. Időbeli differenciák módszere

A megerősítéses tanulás egyik központi koncepciója az ***időbeli differenciák*** *(Temporal Difference, TD)* módszere, amely a környezet dinamikájának ismerete nélkül, direkt módon képes tanulni. A dinamikus programozáshoz hasonlóan a politika frissítése részben előző becsléseken alapul. A ***kontrol probléma*** megoldására, azaz az optimális politika megkeresésére, mindkét módszer az

általánosított politika iteráció valamilyen változatát használja. A lényeges különbség adott π politika melletti V^{π} becslésének módjából adódik.

3.2.2.1. TD jóslás

A TD módszer a jóslási probléma megoldására a tapasztalatokat használja fel. A π politika követése során szerzett tapasztalatait felhasználja a π politikához tartozó V^{π} értékelő függvény V becslésének felülírására. Ha a t időpillanatban a rendszer az s_t állapotban van, akkor a TD módszer már rögtön a következő lépésben módosítja a V -t a megfigyelt r_{t+1} jutalom és $V(s_{t+1})$ korábbi becslésének felhasználásával. Ennek alapján a legegyszerűbb TD módszer, amelyet gyakran TD(0)-nak hívnak, a következőképpen írható le:

$$V(s_t) \leftarrow V(s_t) + \alpha\left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right] \qquad (25)$$

A *dinamikus programozás* egy becslés, de nem a teljes környezeti dinamika ismeretét feltételező várható érték, hanem a $V^{\pi}(s_{t+1})$-et helyettesítő $V_t(s_{t+1})$ felhasználása miatt. A TD többféle szempontból is becslés, egyrészt mintát vesz a várható értékből, másrészt felhasználja a V^{π} értékelő függvény V^{t} becslését.

A 4. táblázatban megadom a TD módszer algoritmusát.

Input: π policy to be evaluated

Initialize: $V(s)$ arbitrarily

Repeat for each episode

    initialise s

    Repeat for each step of episode

        take a,r,s'

        $V(s_t) \leftarrow V(s_t) + \alpha\left[r + \gamma V(s_{t+1} - V(s_t))\right]$

        $s \leftarrow s'$

- until s is terminal

4. táblázat. TD(0) algoritmus V^{π} becslésére

Milyen előnyei vannak a TD módszernek a dinamikus programozással szemben? Egyrészt, nem szükséges a környezeti dinamika ($P^{a}_{ss'}$, valamint $R^{a}_{ss'}$) ismerete,

másrészt nem kell megvárni egy-egy epizód végét az értékelő függvény frissítéséhez, amely jelentős gyorsulást eredményezhet.

3.2.3. A TD(0) módszer optimalitása

Tegyük fel, hogy adott a tanító példák véges halmaza (véges számú epizód vagy lépés). A fokozatos tanítás hagyományos megközelítése, hogy a tanító példákat újra és újra megismételjük, addig, ameddig a módszer nem konvergál. V közelítésekor a (25) egyenlet alapján mindig kiszámítjuk a szükséges változtatás mértékét, de csak egyszer, vagy az epizód végén, vagy ha végigértünk a tanító halmazon, frissítjük ténylegesen V -t, mégpedig a változtatások összegével. Ezért ezt a módszert *kötegelt feldolgozásnak* nevezzük.
Belátható, hogy kötegelt feldolgozás esetén a TD(0) módszer determinisztikusan konvergál az egyértelmű válaszhoz, mégpedig az α lépésköztől függetlenül, annyi megszorítással, hogy α elég kicsi.

3.2.3.1. SARSA (State-Action-Reward-State-Action)

Ebben a szakaszban megvizsgáljuk, hogy a TD módszerek hogyan használhatók fel a kontrol probléma megoldására. Az általánosított politika iterációt használjuk, csak ezúttal a kiértékelést egy TD módszerrel végezzük el.
Az állapotot értékelő függvény becslése helyett az akcióértékelő függvénnyel foglalkozunk. Szerencsére, a TD(0) esetében az állapotot értékelő függvényre vonatkozó konvergencia-kritériumok szimmetrikusan érvényesek az akciót értékelő függvényre is. Ezért a következőt írhatjuk:

$$Q(s_t, a_r) \leftarrow Q(s_t, a_t) + \alpha\left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right] \qquad (26)$$

Ezt a frissítést minden nemterminális s_t állapot esetén végrehajtjuk. Ha az s_{t+1} terminális, akkor $Q(s_{t+1}, a_{t+1})$ definíció szerint 0. A frissítési szabály az átmenetet leíró $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$ mind az öt elemet tartalmazza. Innen származik a módszer neve is.
A *SARSA* alapján magától értetődően konstruálhatunk aktív kontrol algoritmust: Q^π -t becsüljük, π -t pedig Q^π -re nézve mohóvá tesszük. Az algoritmust a 5. táblázatban adom meg.

```
Initalize: Q(s,a) arbitrarily
for each episode
    initialise s
    choose a in s ε – greedy
    Repeat (for each step of the episode)
        execute a take r, s'
        choose a ε – greedy in s'
        take a, r, s'
        Q(s_t, a_r) ← Q(s_t, a_t) + α[r_{t+1} + γQ(s_{t+1}, a_{t+1}) − Q(s_t, a_t)]
        s ← s'; a ← a';
    Until s is terminal
```

5. táblázat. SARSA: aktív TD szabályozási algoritmus

3.3 Emlékeztető nyomok módszere

Ebben a részben egy olyan módszert ismertetek, amely eredményesen kombinálható az előzőekben áttekintett eljárásokkal, és sok esetben hatékonyabb tanuló algoritmus nyerhető a segítségével. Az ***emlékeztető nyomok*** módszerének két alapvető értelmezése lehetséges: az egyik az előre, a másik a visszatekintő változat.

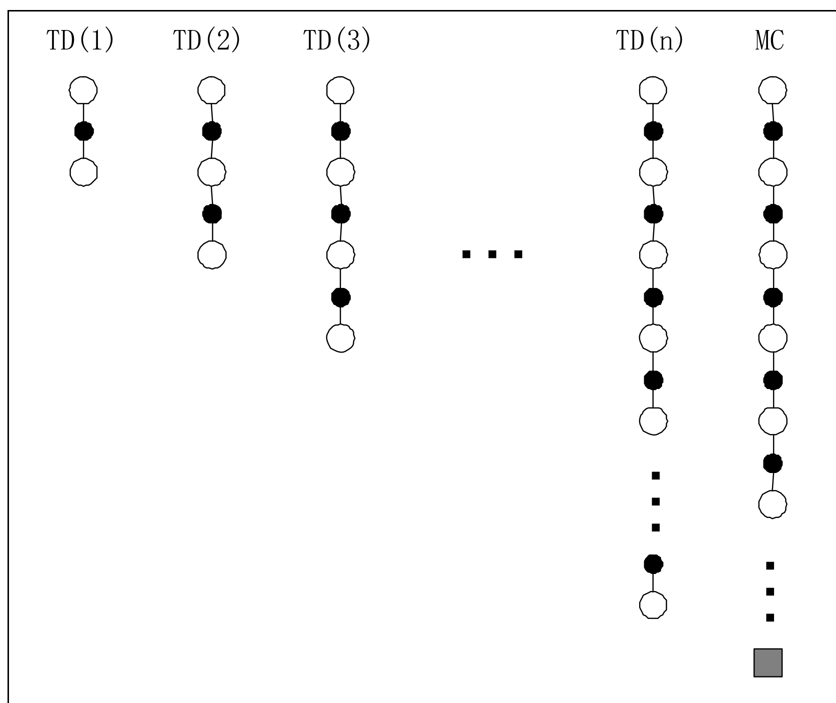
Az előretekintő variáns elsősorban elméleti jelentőségű, és hidat képez a Monte-Carlo módszerek[1] és a TD módszer között.

A visszatekintő változat már sokkal gyakorlatiasabb szemléletű: az emlékeztető nyomok tulajdonképpen hosszú távú memóriaként szolgálnak a tanulás során. Segítségükkel áthidalható a bekövetkező események és tanulási információk közötti hézag.

3.3.1. n lépéses TD jóslás

A legegyszerűbb, 1 lépéses TD módszer csak a közvetlen jutalmat és a következő állapot becsült értékét veszi figyelembe. Vannak "hibrid" TD módszerek, amelyek *n* lépésben képezik a felösszegzési gráf értékét. Ezeket szokás *TD(n)*-nel jelölni.

[1] A Monte-Carlo módszerek szintén az optimális politika és értékelő függvény megtalálására szolgáló eljárások.



5. ábra n lépéses TD felösszegzési gráf

Tegyük fel, hogy az s_t állapotra vonatkozó felülírási értéket szeretnénk meghatározni $s_t, r_{t+1}, s_{t+1}, r_{t+2}, s_{t+2}, \ldots s_T, r_T$ pedig az állapotok és a jutalmak sorozata. Az egylépéses TD módszer esetében a felülírási érték nem más, mint a közvetlen jutalom, plusz a következő állapot diszkontált becsült értéke. Képlettel:

$$R_t^{(1)} = r_{t+1} + \gamma V_t(s_{t+1}) \tag{27}$$

Ez értelmes, hiszen a $\gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots + \gamma^{T-t-1} r_T$ helyét veszi át a $\gamma V_t(s_{t+1})$ becslés. Az ötletet először kettő, majd *n* lépésre általánosítva jutunk az *n* lépéses becsléshez:

$$R_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... + \gamma^{n-1} r_{t+n} + \gamma^n V_t(s_{t+n}) \tag{28}$$

Tulajdonképpen az történik, hogy a hozamot *n* lépés után csonkoljuk, és a hiányzó részt az *n* lépéssel későbbi állapot diszkontált becsült értékével helyettesítjük. Természetesen, ha az epizód előbb véget ér, mintsem elérünk az *n*-edik lépésig, akkor a csonkolást az epizód végén hajtjuk végre. Ebben az esetben a teljes hozamot kapjuk.

Az *n* lépéses felösszegzési gráfot az *n* lépéses hozam irányába eső felülírási értékkel definiáljuk. Az állapotot értékelő esetben a $V_t(s_t)$ felülírását a következőképpen adom meg:

$$\Delta V_t(s) = \alpha\left[R^{(n)} - V_t(s_t)\right] \quad (29)$$

ahol $\alpha > 0$. Természetesen minden $s \neq s_t$ esetén $\Delta V_t(s_t) = 0$. Az *n* lépéses módszert ezzel az egyenlettel adjuk meg a közvetlen felülírási szabály helyett. Ennek az az oka, hogy kétféle felülírást is megkülönböztetünk. Az egyik esetben a felülírás folyamatosan történik *(online updating),* amikor kiszámítjuk a változásokat. Ekkor minden $V_{t+1}(s) = V_t(s) + \Delta V_t(s)$. A másik esetben a felülírás az epizód végén történik az összegyűjtött módosítások összegével *(off-line updating).* Azaz ekkor az epizód folyamán $V_t(s)$ értéke konstans, a végén pedig $V(s) + \sum_{t=0}^{T-1} \Delta V_t(s)$ lesz.

Az összes *n* lépéses hozam várható értéke az igazi értékelő függvény jelenleginél jobb közelítését adja, azaz bármely V-re a V szerinti *n* lépéses hozam várható értéke V^{π}-nek V-nél jobb közelítése. Ez azt jelenti, hogy a legnagyobb hiba az új közelítés esetében legfeljebb akkora, mint a V legnagyobb hibájának γ^n-szerese. Ezt az *n* lépéses hozam hibacsökkentési tulajdonságának nevezzük, és képlettel a következőképpen írhatjuk le:

$$\max_s \left| E_{\pi}\left\{R_t^{(n)} \mid s_t = s\right\} - V^{\pi}(s)\right| \leq \gamma^n \max_s \left| V(s) - V^{\pi}(s) \right| \quad (30)$$

Ennek alapján formálisan is igazolható, hogy mind a folytonos, mind a kötegelt felülírás esetében a TD módszer helyesen működik, a közelítési feltételek mellett.

A hibacsökkentési tulajdonság ellenére a TD(*n*) módszereket implementációs nehézségek miatt a gyakorlatban ritkán használják, inkább csak elméleti jelentőséggel bírnak.

3.3.2. A TD (λ) előretekintő változata

A felösszegzési gráf értékének meghatározása nem csak *n* lépéses hozamokkal, hanem ezek átlagával, sőt, súlyozott átlagával is történhet. Elképzelhető például, hogy a felösszegzési gráf értékét kettő és négy lépéses hozamok átlagával határozzuk meg. Az így előállított értéket a ***komplex felösszegzési gráf*** értékének nevezzük.

A TD(λ) algoritmus speciális esete az *n* lépéses hozamok átlagolásának, ugyanis ez az átlag minden *n*-re tartalmazza az *n* lépéses hozamot, mégpedig λ^{n-1}-nel súlyozva. Annak érdekében, hogy a súlyok összege 1 legyen, $1-\lambda$-val

normalizálunk. Az így kapott hozamot λ-hozamnak (λ Return) nevezzük, és a következőképpen írhatjuk le:

$$R_t^{\lambda} = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)} \tag{31}$$

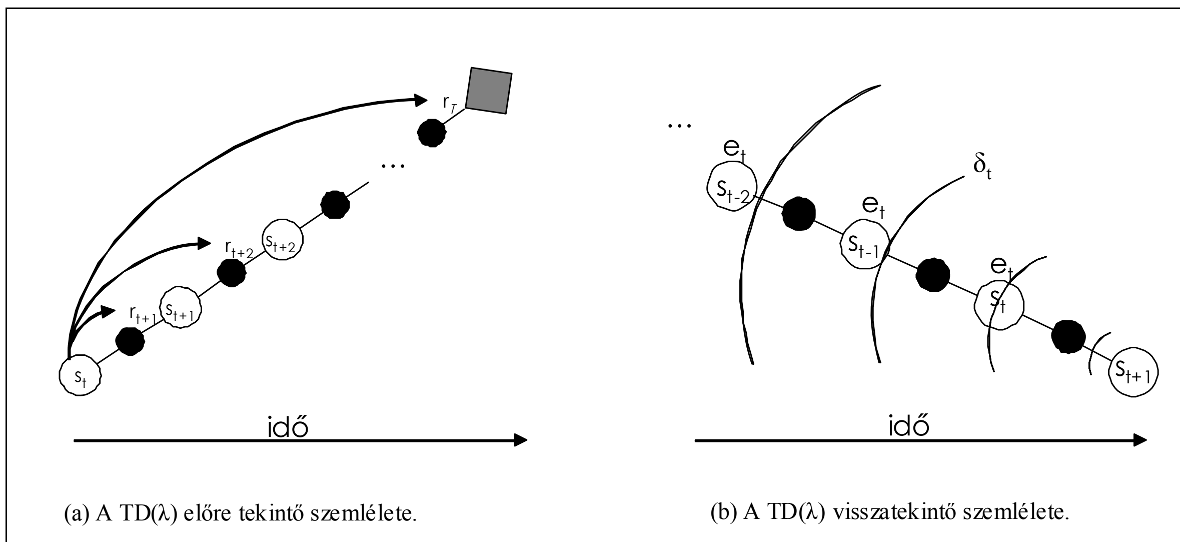
A súlyok minden hozzáadott lépés esetén λ-val felejtődnek el, ennyivel csengenek le. Ha elérjük az epizód végét, azaz terminális állapotba kerülünk, akkor minden ennél nagyobb *n*-re az *n* lépéses hozam R_t-vel egyezik meg. Így átfogalmazhatjuk az előző definíciót:

$$R_t^{\lambda} = (1-\lambda)\sum_{n=1}^{T-t-1} \lambda^{n-1} R_t^{(n)} + \gamma^{T-t-1} R_t \tag{32}$$

Definiáljuk a λ-hozam algoritmust úgy, hogy a λ-hozam alapján határozza meg a felülírási gráf értékét. Azaz minden t lépésben a $\Delta V_t(s_t)$ változtatás mértéke:

$$\Delta V_t(s_t) = \alpha[R_t^{\lambda} - V_t(s_t)] \tag{33}$$

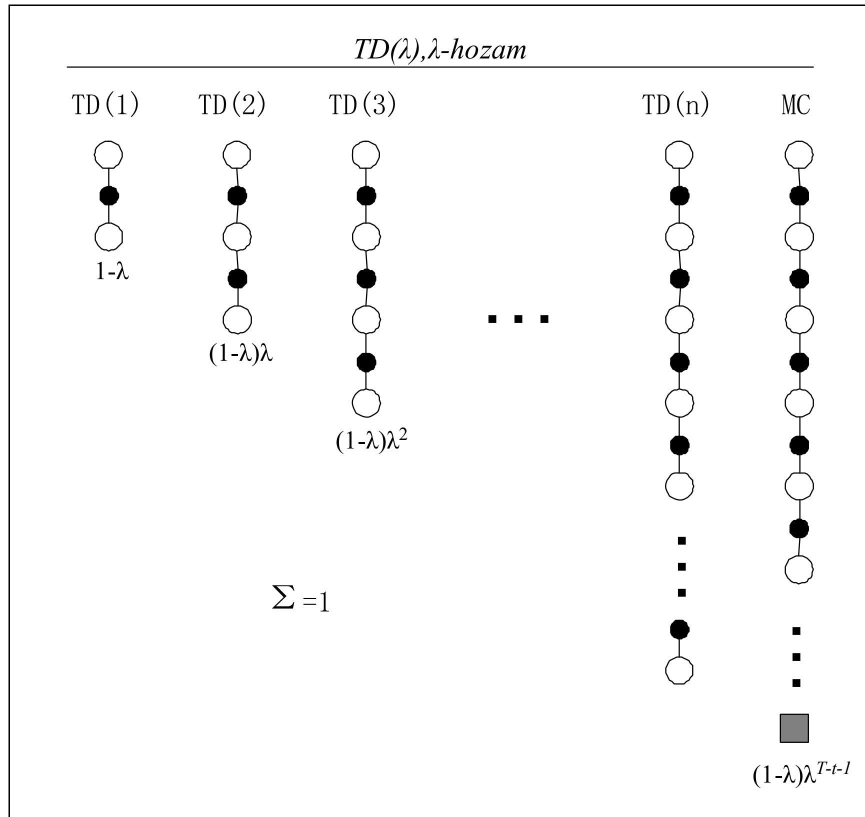
Ez a TD(λ) módszer előre tekintő változata, amely egyaránt alkalmazható mind folytonos, mind kötegelt felülírásra.



6. ábra A TD(λ) kétféle változata.

3.3.3. A TD(λ) visszatekintő változata

A visszatekintő változat az előretekintőnél sokkal könnyebben implementálható. Első lépésként minden egyes állapothoz bevezetünk egy memóriaváltozót, az ***emlékeztető nyomot*** *(eligibility trace).* Ez a változó azt jelzi, hogy "az utóbbi időben" hányszor látogattuk meg az adott állapotot. Pontosítva, t időpontban az s nemterminális állapotra vonatkozó emlékeztető nyom:



7. ábra A TD(λ) felösszegzési gráfja

$$e_t(s)\begin{cases}\gamma\lambda e_{t-1}(s) & ha\ s \neq s_t \\ \gamma\lambda e_{t-1}(s)+1 & ha\ s = s_t\end{cases} \tag{34}$$

Az egyenletben γ a diszkontálási hányados, λ definíciója pedig a TD(λ) algoritmussal foglalkozó részben megtalálható. Az ilyen emlékeztető nyomot gyűjtő nyomnak is nevezhetjük, hiszen az állapot többszöri elérésével fokozatosan felerősödik, majd lecseng, ha nem látogatjuk meg többször az adott állapotot.

A nyom megjegyzi, hogy melyik állapotokat értük el mostanában, és a megerősítendő eseményeket összekapcsolja a TD módszer hibájával, amely az állapotot értékelő esetben a következőképpen írható:

$$\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t) \quad (35)$$

A globális TD hiba segítségével megadhatjuk az értékelő függvény felülírási egyenletét:

$$\Delta V_t(s) = \alpha \delta_t e_t(s), \forall s \in s \quad (36)$$

Az előző részben leírtakhoz hasonlóan itt is elmondható, hogy a becslést akár folyamatosan, vagy az epizód végén kötegelten is módosíthatjuk.

A (35) és a (36) egyenletek megadják a TD(λ) módszer definícióját. Az algoritmust, folyamatos felülírás esetén, a következő táblázatban adom meg:

```
Initialize V(s) arbitarily and e(s) = 0∀s ∈ S
Repeat for each step in episode
    initialize s
    choose a in s using π
    take r, s'
    δ ← r + γV(s') − V(s)
    e(s) ← e(s) + 1
    For each s
        V(s) ← V(s) + αδe(s))
        e(s) ← γλe(s)
    s ← s'
Until s is terminal
```

6. táblázat. A TD(λ) módszer folyamatos felülírás esetén

Minden egyes lépésben megállapítjuk az aktuális TD hibát, és elvégezzük a felülírást minden olyan állapotra, amelynek van nyoma. Ezt szemléletesen a 6(b). ábra érzékelteti.

Az előző részekben megvizsgáltuk a TD(λ) előre- és visszatekintő változatait, és megjegyeztük, hogy míg az előretekintő változat elsősorban elméleti jelentőségű, addig a visszatekintő módszer egyszerűen implementálható, és a gyakorlatban is jól használható algoritmust ad. Érdekes kérdés, hogy vajon ezen két módszer ugyanazon felülírási szabályt valósítja-e meg? A válasz: igen, ugyanis igazolható, hogy a kötegelt λ-hozam módszer ugyanazt a felülírási szabályt valósítja meg, mint a kötegelt TD(λ) algoritmus, vagyis az előre- és visszatekintő módszer ekvivalens.

3.3.4. A TD(λ) módszer alkalmazása logikai játékokban

A megerősítéses tanulás és a mesterséges neuronhálózatok összekapcsolásának egyik leghíresebb és legtöbbet megemlített példája Gerald Tesauro backgammon (ostábla) programja, a TD-Gammon. A TD-Gammon egy öntanuló program, amely a minimális előzetesen belekódolt ismeretek ellenére rendkívül jó szintet ért el csak azáltal, hogy játszmák százezreit játszotta saját magával, és közben ezekből a játszmákból tanult. A tanuló algoritmusa a TD(λ) algoritmus volt és egy többrétegű perceptront használt nemlineáris függvényapproximátorként az állapot-érték függvény közelítésére.

A backgammon játék szabályai bonyolultak, és a játéknak van egy nagyon erős sztochasztikus tulajdonsága is. A 30 báb és a 24 lehetséges pozíció miatt a lehetséges játékállások száma gigantikus: 10^{20} állapot. Ennyi állást külön-külön táblázatban tárolni fizikailag is képtelenség. A kockadobás átlagosan 20 lehetséges lépést eredményez, ami gátat emel a játékfa-kiértékelő heurisztikus keresések komolyabb mélységben való alkalmazásának. A játékban nincs igazi szerepe a tervezésnek, a mesteri szinten játszó emberi játékosoknál is a mintafelismerő képesség és az ismert minták száma kiemelkedő.

A TD-Gammon egy standard többrétegű perceptront használt, az állapotérték függvény becslésére. A hálózat bemenete a „nyers” állás volt, a rejtett réteg 40, a kimeneti réteg egyetlen neuront tartalmazott. A hálózat kimenetén megjelenő értéket úgy értelmezték, mint a játszma adott állásból való megnyerésének a valószínűsége. Ennek megfelelően a közvetlen jutalom a játszma során mindig 0 volt, kivétel ez alól a győztes lépésé, amikor 1.

Tesauro a TD-Gammon első verziója után továbblépett, az újabb verziókba már előzetes backgammon ismereteket is vitt (bizonyos számolható jellemzőkel

belekódolt a hálózat bemenetébe), ezen felül 2-3 lépés mélységű heurisztikus keresést alkalmazott. Az eredmény: a TD-Gammon 3.0 a világbajnokokkal is felveszi a versenyt. Ez a program olyan, eddig ismeretlen megnyitásokat fedezett fel, amiket a legjobb játékosok is átvenni kényszerültek.

3.4. Kapcsolat a függvény-approximátorokkal

Korábban az értékelő függvényt véges sok állapot-akció párossal írtuk le. Így az értékelő függvényünket egy egyszerű táblázat segítségével valósítottuk meg. Ez sajnos sok állapot-akció párosra nem megfelelő szerkezet, mert nemcsak a táblázat helyigénye nagy, hanem a teljes feltöltéséhez szükséges idő is. A kulcskérdés az általánosításban rejlik. Hogyan érhetjük el, hogy korlátozott számú kísérlet eredményének általánosításával is jó közelítését kapjuk az értékelő függvényünk egész tartományán?

Ez nehéz probléma, mivel a legtöbb feladatban, ahol megerősítéses tanulást szeretnénk alkalmazni, kevés számú állapot ismeretében kell olyan állapotokra is becslést adni, amelyekkel még sohasem találkoztunk. Ez a helyzet például a folytonos állapot-akciótérnél, vagy a vizuális kép érzékelésénél is. Így jutunk el a példák alapján történő általánosítási eljárásokhoz, aminek jól kidolgozott algoritmusai vannak.

3.4.1. Értékelő függvény becslése függvény-approximátorokkal

Az általánosításnak azt a formáját, ahol mintavételezésekből általánosítva valósítjuk meg a függvényünket, ***függvény-approximátornak*** nevezzük. Ez a módszer a megerősítéses tanulásnak egy alkalmazása.

Az állapotot értékelő függvényünket a t pillanatban V_t-vel jelöljük, és ebben az esetben nem táblázattal hanem egy $\vec{\theta}_t$ paraméter vektorral adjuk meg, azaz $V_t = V\left(\vec{\theta}_t(\vec{s}_t)\right)$. V_t-t választhatjuk egy mesterséges neurális hálózat kimenetének is. Ebben az esetben a $\vec{\theta}_t$ vektor a hálózat súlyvektoraiból áll. Tipikusan a paraméterek száma (a paraméter vektor komponenseinek a száma) sokkal

kisebb, mint az állapotok száma. Következésképpen, ha változtatunk egy paraméter értékén, akkor sok állapot értékét módosítottuk.

Minden becslést az értékek mentésével valósítunk meg. Jelöljük $s \mapsto v$-vel az s állapothoz tartozó mentést. Ez az, amivel a következő időpontban és állapotban becsülni fogjuk az értékelő függvényünket. Például *DP* esetén $s_t \mapsto E_\Pi\{r_{t+1} + \gamma V_t(s_{t+1}) \mid s_t = s\}, TD(0)$ esetén $s \mapsto r_{t+1} + \gamma V_t(s_{t+1})$ és $TD(\lambda)$ esetén $s \mapsto R_t^\lambda$ alakul.

A táblázatos módszer triviális módon adja vissza a kívánt értékelő függvényt. Az s állapothoz tartozó táblabejegyzést közelítem a tőle elvárt v értékhez. A tetszőlegesen összetett függvény-approximátor tanítása pont ilyen $s \mapsto v$ állapot-érték mentésekből álló tanítási párokkal valósítható meg. Így a megerősítéses tanulás minden módszerét alkalmazhatjuk. A legjobb neurális hálózatok és statisztikus módszerek legtöbbje a megerősítéses tanulással ellentétben statikus és lassú tanulást igényelnek. Ezzel szemben a megerősítéses tanulás képes a környezetével kölcsönhatásban ahhoz igazodni. Ehhez olyan eljárások kellenek, amelyek nem stacionárius célfüggvényt is kezelni tudnak.

Teljesítmény értékelések a függvény közelítő eljárásokra: a legtöbb felügyelt tanulási módszer az ***átlagos négyzetes hiba*** *(Mean-Square Error MSE)* minimalizálására törekszik az állapotok egy bizonyos eloszlásával.

$$MSE(\vec{\theta}_t) = \sum_{s \in S} P(s)[V^\pi(s) - V_t(s)]^2 \tag{37}$$

ahol P az állapotok hibájának a súlyeloszlása. Ez azért fontos, mert általában nem lehet a hibát nullára csökkenteni minden állapotban. Az egyenletes hibaeloszlás érdekében a hibák súlyozásának eloszlását tegyük egyenlővé a tanítási állapotok eloszlásával. Ezt az eloszlást *aktív politika eloszlásnak* hívjuk, ha egy olyan értékelő függvényt becsülünk, ami egy környezettel kölcsönható ügynök politikájához tartozik. Ez egy olyan eloszlást valósít meg, amely segítségével az értékelő függvényt csak azokban az állapotokban tanuljuk, ahol az ügynök-környezet rendszer előfordul.

A hiba minimalizálása egy $\vec{\theta}^*$ vektor megtalálásával egyezik meg, ahol $MSE(\vec{\theta}^*) \le MSE(\vec{\theta})$ minden θ-ra. Ennek elérése egyszerűbb esetekben, mint a lineáris függvény-approximátorok, lehetséges, nem lineárisoknál csak egy lokális

minimum garantált. De a legjobb becslés nem minden esetben a legoptimálisabb a hiba szempontjából.

3.4.2. Gradiens keresési eljárás

A gradiens keresési eljárás egy, a függvény-approximátorok terén széles körben elterjedt eljárás, amely nagyon jól illeszkedik a megerősítéses tanulás koncepciójába is. Ebben az esetben a paraméter vektor $\vec{\theta}_t = (\theta(1), \theta(2), \dots, \theta_t(n))^T$ és $V_t(s)$ egyenletesen differenciálható függvénye $\vec{\theta}_t(\vec{s}_t)$-nek minden $s \in S$-re. A hiba minimalizálását a négyzetes hiba függvény $\vec{\theta}_t$ szerinti gradiensével ellentétes irányba haladva érhetjük el, azaz:

$$\begin{aligned} \vec{\theta}_{t+1} &= \vec{\theta}_t - \frac{1}{2}\alpha\nabla_{\vec{\theta}_t}[V^{\pi}(s_t) - V_t(s_t)]^2 \\ &= \vec{\theta}_t + \alpha\nabla_{\vec{\theta}_t}[V^{\pi}(s_t) - V_t(s_t)]\nabla_{\vec{\theta}_t} V_t(\vec{s}_t) \end{aligned} \qquad (38)$$

ahol α a lépésköz paramétere és $\nabla_{\vec{\theta}_t} f(\vec{\theta}_t)$ az f függvény $\vec{\theta}_t$ paramétervektor szerinti gradiense. Belátható, hogy a negatív gradiens irányába haladva a paraméter vektorok terében csökken a négyzetes hiba.

Konvergencia csak abban az esetben garantált, ha a lépésköz paraméterünk az idővel nullához tart, feltéve, hogy teljesül a standard sztochasztikus feltétel, azaz a nullához való konvergencia nem túl gyors. A hibát természetesen egy lépésben is nullára tudnánk csökkenteni egy adott állapotban, de a többi állapotban ez rontaná a becslésünket.

Általában $V^{\pi}(s_t)$-t nem ismerjük pontosan, annak csak zajos, vagy becsült értéke áll rendelkezésünkre. Ekkor eljárásunkban v_t-t helyettesítünk helyette *(Bootstrapping).*

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha[v_t - V_t(s_t)]\nabla_{\vec{\theta}_t} V_t(\vec{s}_t)$$

Ha v_t ***torzítatlan becslése*** *(unbiased estimate),* $V^{\pi}(s_t)$-nek, azaz $E\{v_t\} = V^{\pi}(s_t)$, minden t-re, akkor az α lépésköz nullához való konvergenciája garantálja a sztochasztikus közelítési feltételt.

V_t^{π} helyett a TD hozamát v_t vagy valamilyen átlagát írva megkapjuk a jövőbe tekintő $TD(\lambda)$ gradiens keresési eljárást:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha[R_t^\lambda - V_t(s_t)]\nabla_{\vec{\theta}_t} V(s_t) \quad (39)$$

Sajnos $\lambda < 1$-re R_t^λ nem torzítatlan becslése *a* $V^\pi(s_t)$-nek*,* és így nem feltétlenül konvergál egy lokális optimumhoz. Ennek ellenére egészen jó eredményeket lehet az ilyen *bootstrap* eljárásokkal elérni.

A jövőbe tekintő eljárásokkal az a baj, hogy csak a hozam összegyűjtése után (epizód) tudunk értékelni, s így csak az epizód után tudjuk az értékelő függvényünket módosítani. Ezzel szemben a visszatekintő eljárásokkal az epizódok közben is lehetséges a hangolás, mert az információ rendelkezésre áll. A $TD(\lambda)$ múltba tekintő változata a következőképpen alakul:

$$\begin{aligned} \vec{\theta}_{t+1} &= \vec{\theta}_t + \alpha\delta_t\vec{e}_t \\ \delta_t &= r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t), s \\ \vec{e}_t &= \gamma\lambda\vec{e}_{t-1} + \nabla_{\vec{\theta}_t} V_t(s_t). \end{aligned} \quad (40)$$

ahol δ_t a szokásos TD hiba, $\vec{e}_t$ pedig az úgynevezett *emlékeztető nyom (eligibility)* vektor amely a múltbeli állapotok egyre csökkenő súllyal vett lenyomata.

3.4.3. Lineáris eljárás

Az egyik legfontosabb eljárás a lineáris eljárás, amelyben V_t lineáris függvénye a $\vec{\theta}_t$ paramétervektornak.

$$V_t = \vec{\theta}_t^T \vec{\phi}_{s_t} \quad (41)$$

Ebben az esetben *a* $\nabla_{\vec{\theta}_t} V_t = \vec{\phi}_{s_t}$ tulajdonságvektorral. Így leegyszerűsödött a $\vec{\theta}^*$ keresésére használt egyenletünk is. A lineáris eljárásnak megvan még az a jó tulajdonsága is, hogy csak egyetlen egy optimális paraméter vektor, vagy tartomány létezik. Így garantált az is, hogy konvergencia esetén a globális optimumhoz fog tartani az eljárás.

Az előző fejezetben tárgyalt $TD(\lambda)$ eljárás is konvergál abban az esetben, ha a lépésköz-paramétere csökken a lépésekkel. Ekkor természetesen nem a globális

optimumhoz fog konvergálni, hanem egy olyan $\vec{\theta}_\infty$ paramétervektorhoz, amelynek a hibája a következő egyenlettel adható meg:

$$MSE(\vec{\theta}_\infty) \leq \frac{1-\gamma^\lambda}{1-\gamma} MSE(\vec{\theta}^*)$$

A lineáris eljárások nemcsak az elmélet egyszerűsége miatt érdekesek, hanem hatékonyságuk miatt is, mind adatmennyiség, mind számítási gyorsaságban is. A lineáris eljárás nem feltétlenül függ kritikusan attól, hogy hogyan alakítjuk ki az állapotokból a tulajdonságvektort.

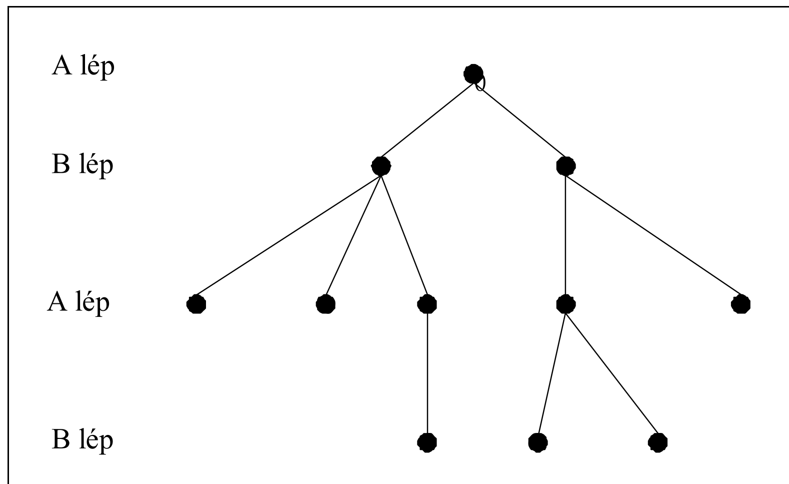
4. A minimax kiértékelés

A kétszemélyes játékok elméletében nagy szerepe van a minimax algoritmusnak, amellyel elméletben minden játék nyerő stratégiája meghatározható és a gyakorlatban is elég jól használható. A ***kétszemélyes, teljes információjú játékok*** osztályát azok a játékok alkotják, amelyekre igaz, hogy:

- A két játékos felváltva lép.
- A játékosok birtokában vannak a játékkal kapcsolatos összes információnak.
- A véletlentől nem függ a játék végeredménye.
- Minden állásban véges sok lehetséges lépés van.
- Véges sok lépésben véget ér a játék.
- A játék nulla-összegű, azaz a nyertes ugyanannyit nyer, amennyit a vesztes elveszít.
- Egyszerre nem győzhet, és nem veszíthet mindkét játékos.

A játékelmélet nagyon fontos tétele, hogy egy teljes információjú kétszemélyes játék esetén mindig létezik az egyik játékos számára nyerő stratégia, feltéve, hogy a játék nem végződhet döntetlennel. Ha előfordulhat döntetlen is, mint például a fix táblaméretű amőbában, akkor az egyik játékos számára létezik legalább döntetlent elérő stratégia.

A kétszemélyes játékokat ábrázolhatjuk fával, amit játékfának nevezünk. A 8. ábrán egy ilyen játékfa látható. Az *A* játékos teszi meg a kezdő lépést. A fa csúcsaiban játékállások vannak, élei az egyes állások közötti szabályos lépéseknek felelnek meg, így a fa gyökerébe a kezdőállás, a levelekbe, pedig a végállapotok kerülnek. Azonos csúcs több helyen is előfordulhat, akár a fa azonos szintjén is. A fa azonos szintjén található csúcsokban levő állásokban mindig azonos játékosnak (*A*-nak vagy *B*-nek) kell lépnie, ezért beszélhetünk a játékfa *A*-szintjéről illetve *B*-szintjéről.



8. ábra Egy fiktív játékfa

A minimax algoritmus a játékfa csúcsaihoz rendel egyet a -1, 0, 1 értékek közül. Ez az érték:

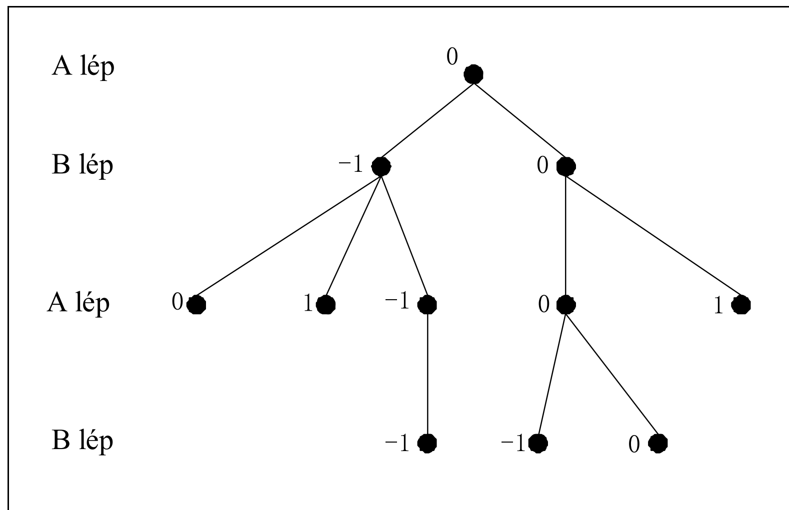
- 1, ha az *A* játékosnak van nyerő stratégiája a csúcsban levő állásból,
- 0, ha az *A*-nak nincs nyerő stratégiája, de döntetlent elérő stratégiája van,
- -1, ha az *A* mindenképpen veszít a csúcsban levő állásból

az optimális ellenfél ellen.

A címkézést a levelektől indulva kezdjük. A levelekhez egyértelműen hozzárendelhető a megfelelő érték, a teljes információjú, kétszemélyes játékok definíciója miatt. A belső csúcsok megcímkézése különbözik, attól függően, hogy azok a fa A-szintjén vagy B-szintjén vannak.

- *A* szinten a közvetlen leszármazottak értékeinek maximuma lesz a csúcs értéke. Minthogy az *A* játékos következik lépésen, van választási lehetősége, ezért léphet a neki legmegfelelőbb irányba.
- *B* szinten a közvetlen leszármazottak értékeinek a minimuma lesz a csúcs értéke. A *B* szinten a *B* játékos lép, várhatóan az *A* játékos számára a legrosszabb lépést választja.

A 9. ábrán egy kiértékelt játékfa látható. A felcímkézés eredményeként a fa gyökerének is lesz valamilyen értéke, ami alapján eldönthető, hogy a gyökérben levő állásból a játék megnyerhető, vagy ha nem, akkor legalább döntetlen elérhető-e.



9. ábra Egy kiértékelt játékfa

A gyakorlatban általában nem építhető fel a teljes játékfa, csak néhány lépés mélységig a kombinatorikus robbanás miatt. Már egy egyszerűbb játék esetén is olyan méretű lehet a teljes játékfa, amely a leggyorsabb számítógépekkel sem értékelhető ki elfogadható válaszidőn belül. A kiértékelés (illetve a fa) egy bizonyos mélységben el van vágva. A levelek ilyenkor nem végállásokat tartalmaznak, ezért nem lehet egyértelműen eldönteni róluk, hogy nyerő vagy vesztő állások.

Statikus kiértékelő függvénynek hívják azt a függvényt, ami számszerűsíti a levelekben levő állások értékét, mégpedig úgy, hogy pozitív értékeket rendel az *A* számára kedvező állásokhoz, 0-hoz közeli értéket a döntetlen-szerű állásokhoz, és negatív számokat az *A*-nak kedvezőtlen állásokhoz. Ez a kiértékelő függvény azért statikus, mert a játékállást csak egyetlen momentumban vizsgálja, nem pedig a játékállások sorozataként.

A levelek kiértékelése után ugyanúgy alkalmazható a minimax értékfelterjesztés, mint a bemutatott elméleti esetben teljes játékfa esetén. Végül a gyökérben levő állásban a legjobb lépés az az él, amely a maximális értékű csúcsba vezet. A megtalált lépés csak a statikus kiértékelő függvény szerint optimális, ezért nagyon sok múlik ennek a függvénynek a helyes megvalósításától, hogy mennyire jól értékeli az állásokat.

5. A megerősítéses tanulás bemutatása egy tanuló amőba játék implementációjában

5.1 Az amőba játék története, szabályai

Az amőba a legismertebb és szabályait tekintve legegyszerűbb táblás játékok egyike. Eredetileg egy 19x19-es Go táblán, de napjainkban általában négyzetrácsos papíron játsszák. A négyzetháló mérete lehet egyenlő a rendelkezésre álló papír méretével, de kisebb terület is kijelölhető. A játékosok felváltva helyeznek egy-egy jelet a tábla valamelyik, még üres négyzetébe; mindenki a saját jelével játszva. A leggyakoribb jelek az X és az O, de másmilyenek is használhatók. Az nyer, akinek sikerül saját jeleiből ötöt egyenes vonalban, vízszintesen, függőlegesen vagy átlósan egymás mellé helyeznie.
A játék ismertsége ellenére, kevesen tudják, hogy az amőba eredetileg kétlépcsős küzdelem volt. Mindkét játékosnak 100-100 bábu állt rendelkezésére, és ha ezek mind úgy kerültek fel a táblára, hogy közben egyiküknek sem sikerült az ötös malmát összehoznia, akkor egy "tologatós" versennyel folytatódott a parti. Ekkor a játékosok felváltva egy-egy saját bábujukat vagy vízszintesen, vagy függőlegesen egy szabadon választott szomszédos üres mezőre áttolhatják.
A kezdő játékos előnye elvitathatatlan, ezért az amőbát versenyszerű szinten űzők különböző megszorító szabályokat találtak ki a kezdő játékos számára, és más-más néven nevezték el az így átalakított játékot (Go-Moku, Pente, Renju stb.). Egy ilyen megszorító szabály például a *hármas szabály* is. E szerint, a kezdő játékos köteles a harmadik lépése megtétele előtt két különböző lépését felajánlani ellenfelének választásra. Szinte bizonyos, hogy a választásra felkínált alternatíva nem egyenértékű lépésekből áll, így a másodiknak lépés miatt hátrányt szenvedő ezzel a szabállyal visszakaphat valamit ellenfele kezdési előnyéből. Az sem kizárt, hogy a szabály alkalmazásával megfordul az előny, többet ad vissza, mint amennyit eredetileg élvezett, ám, az ugyanezen szabály alkalmazásával játszott második partiban, garantáltan megfordulnak az esélyek.
Példaprogramom tervezésénél eltekintek a kezdő játékos előnyétől, és a hétköznapjainkban szeretettel játszott hagyományos amőba szabályait veszem alapul.

5.2 Állapottér és rendszer terv tanuló amőba alkalmazáshoz

Ebben a részben egy konkrét megvalósítását adom az ügynök környezet modellnek. A feladat a következő: az amőba játékhoz olyan kiértékelő függvény találása, amely játékerőben egy emberi játékossal is felveszi a versenyt. A rendszer implementációját Microsoft Developer Studio 6.0 fejlesztőkörnyezettel C++ nyelven végeztem. A részletes Hardware és Software-architektúrát a 2-es melléklet tartalmazza.

5.2.1. A környezet modellje

A környezet modelljének kettős feladata van: az első az, hogy az ügynök felé biztosítsa a rendszer aktuális állapotát, illetve az ügynök által küldött vezérlőjelek alapján változtassa meg a rendszer helyzetét. A környezet egy 17x17 négyzetből álló síkfelület, amelyen X illetve O jeleket helyeznek el két játékos felváltva.

5.2.1.1. Az állapottér reprezentációja

A környezetet alapértelmezésben egy 17 x 17-es mátrixszal reprezentáltam. Mint említettem, az eredeti játékok a 19 x 19-es Go táblán játszották, de a különböző versenyszabályok szerint ennek méretét szűkíteni szokásos. A környezet, így a reprezentációs mátrix mérete viszont az alkalmazás SetupDlg osztályának segítségével változtatható. A reprezentációs mátrix (10. ábra) mezői integer típusú egész számok, amelyek tartalmazhatnak 0-t, ez azt jelenti, hogy a mező üres, illetve 1-t, ha X, 2-t ha O jelet helyeztek el rajtuk a játékosok.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
2																
3																
4																
5																
6																
7						2				2						
8							1		1							
9								1	2	2	1					
10									1	1	2					
11									2	1						
12											2					
13																
14																
15																
16																
17																

10. ábra Reprezentációs mátrix a játékállás értékeléshez

Kezdő állapotban a mátrix csupa 0-t tartalmaz.

Egyetlen lehetséges operáció létezik, miszerint a soron következő játékos saját bejegyzését helyezi el a még üresen álló, tehát 0-t tartalmazó mátrixmezők egyikében.

Célállapot, hogy a reprezentációs mátrix bejegyzései közvetlenül egymás mellett, alatt, illetve átlósan 5, az ügynök által használt azonos bejegyzést tartalmazzanak.

5.2.2. Az ügynök modellje

Az ügynök modellje két részből épül fel: az állapotot értékelő függvény közelítéséből, valamint a kiértékelő függvény aktuális hipotéziseinek alapján a következő lépés döntésének meghozatalából.

5.2.2.1. A következő lépés döntésének meghozatala

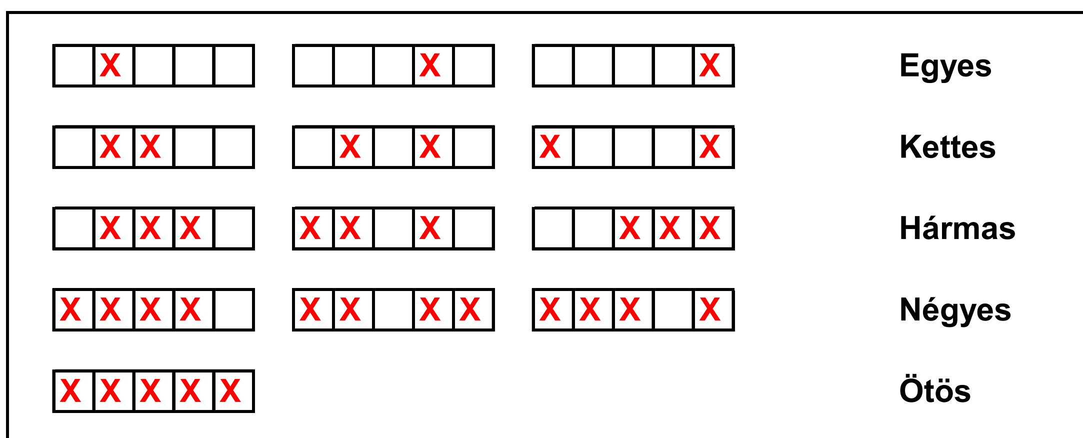
A játékkal kapcsolatos háttérismereteimet felhasználtam, és egy modellvezérelt rendszert hoztam létre. A háttérismeretek felhasználását szükségesnek láttam ennek a játéknak a programozhatóságánál, mivel a játék komplexitása, a nyers játékállások mennyisége olyan nagy méretű állapotteret jelent, amelyet nehezen lehetne mindenféle heurisztika nélkül bejárni.

A környezet reprezentálására alkalmazott mátrixszerkezet alapján a helyes lépés kiválasztásához, valamint a tanuláshoz előosztályozást végeztem, mivel ha csak

azt nézzük, hogy egy játékállás 90 fokkal elforgatva ugyan az lehet, de a mátrix-reprezentációban mégis topológiailag másnak látszik, nagyon sok erőforrást takarítok meg, ha ezeket az azonos játékállásokat valóban azonos módon reprezentálom.

Nem az aktuális játékállás előosztályozását végezem, hanem a következő lehetséges lépések után kialakulót, egy kiértékelő algoritmus segítségével. Ezt a módszert alkalmazta Tessauro Backgammon játékosa is, amelyben a neurális hálózat bemenetére a szabályos lépések mellett kialakítható lehetséges következő nyers játékállások voltak kötve, kimenete, pedig egyetlen neuron volt, amely értéke azt mutatatta meg, hogy mekkora esélye van a rendszernek a nyerésre. Ilyen módon az optimális döntés meghozatalánál ezt az esélyt, hasznosságértéket kell maximalizálnia az ügynöknek. Keresési feladatnál mindenképpen olyan tulajdonságokat tudunk vizsgálni, amelyek maximum, illetve minimum értékének meghatározásával tudunk döntést kialakítani. Ez alól csak egyetlen kivétel lehet, ha a teljes döntési fa kiértékelésére lehetőség nyílik. Ebben az esetben elég bármilyen reprezentációval megadott, akár nem numerikus célértéket keresni, mivel a fa leveleitől felfelé haladva a gyökér felé, az optimális döntés könnyen megtalálható. A kombinatorikus robbanás miatt viszont legtöbb esetben, így az amőba játék esetében sincs lehetőség a teljes játékfa kiértékelésére.

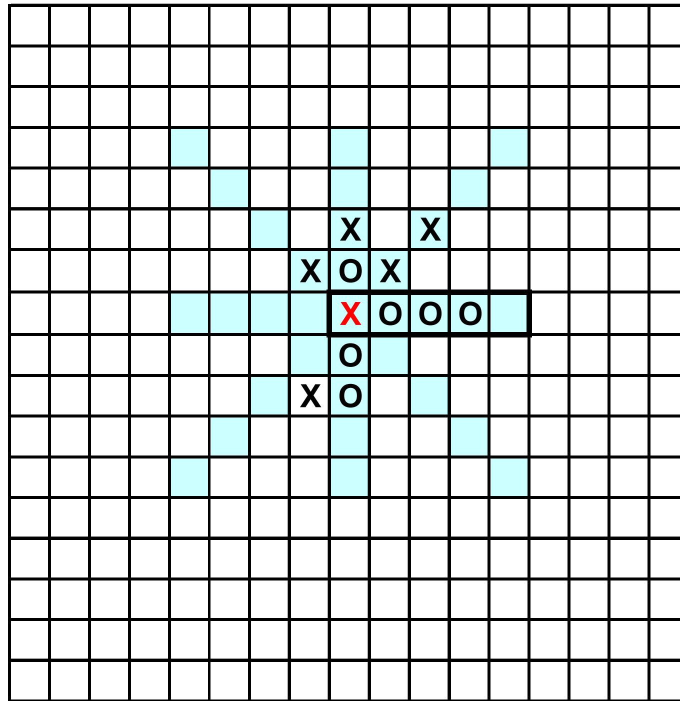
Az előosztályozás során a következő lehetséges lépés helyére leteszek egy virtuális jelet, és ezután elemzem a kialakult nyers játékállást. A lehetséges helyek azok a mezők, amit még nem foglalt el játékos.



11. ábra Amőba játék helyzetei

Az amőba játék során 5 egymás melletti jelet kell kialakítanunk. Ez ugyebár nem történhet meg anélkül, hogy előbb ne legyen 1, 2, 3 majd 4 jelünk egymáshoz viszonyítva maximum 4 távolságra. Nem biztos, hogy ezek már kettes, hármas, négyes "korukban" közvetlenül egymás mellet helyezkednek el, lehet közöttük helyenként üres játékmező is, de mindenképen a függőleges, vízszintes, és a pozitív, negatív meredekségű irányok szerint egy sorban vannak. Ez azt jelenti, hogy ötös építéséhez el kell kezdenünk a legkisebb egységektől kezdve építkezni, folyamatosan egyre nagyobb egységeket létrehozni, közben persze ügyelve arra, hogy ellenfelünk is a nyerésre tör. Emiatt az ő egységeit meg kell törni, le kell zárni ezen kezdeményezések végeit, megállítva építkezését. A 11-es ábra mutat példát az amőba játék helyzeteire.

A kiértékelő függvényt csak azokra a helyzetekre kell alkalmaznom, amelyek a letett virtuális jel környezetében vannak, hiszen azt szeretném megtudni, hogy ha ide tennék, milyen hasznosság értéket érnék el. Az eredményként kapott hasznosság tehát arra a mezőre vonatkozik, amelyre a virtuális jelet leraktam. Az sem torzítaná a kiértékelés hatását, ha a teljes játékteret végignézném, az összes esélyt leszámolva minden egyes virtuális jel letétele után, de ez nagyon lelassítaná a rendszer működését. A játékmező egyéb részét nem érinti lépésem, ezért ezekkel felesleges foglalkozni.
A kiértékelés folyamán a virtuális jel környékén lévő helyzeteket úgy vizsgálom meg, hogy a virtuális jelet csillag középpontnak tekintve, egy 5 mátrixmező hosszúságú gondolatbeli téglalapot mozgatok végig a reprezentációs mátrix minden lehetséges pozíciójában úgy, hogy a virtuális jel minden alkalommal benne legyen ebben a téglalapban. Ez minden irányban 5 lehetséges pozíciót jelent, ami ha összeszámoljuk a vízszintes, függőleges, valamint a pozitív és negatív meredekségű átlókat, összesen húsz. A 12. ábra a virtuális bejegyzés környezetében lévő helyzetek megkeresését szemlélteti. Ezen "mikrokörnyezeten" kívüli játéktér számunkra nem érdekes ebben a pillanatban.



12. ábra helyzetek leszámolás a virtuális bejegyzés csillagközéppontjából

Csak annak a helyzetnek van hasznosság értéke, amely során csupán azonos jelek kerülnek a kiértékelő téglalapba, mert csak ezekből van esély ugyan ebben a pozícióban 5-öt építeni.

Az előosztályozás során tehát a következő lehetséges lépés helyére leteszek egy virtuális jelet, és ennek környezetében leszámolom, ha én tennék ide, hány darab kettes, hármas, négyes stb. helyzetem keletkezne, illetve hány hasonlótól fosztanám meg ellenfelem, ha elfoglalom a mezőt. Ezeknek a helyzeteknek a számát beszorzom a helyzetekhez tartozó kiértékelő függvény súly paraméterekkel, és saját esélyeimre adódó értéket, valamint ellenfelem tönkretett esélyeit jutalmazó értéket összegzem. Egyszerűbben megfogalmazva ezen helyzetek értékeinek lineáris kombinációja adja a hasznosság értéket. A legnagyobb értéket adó üres mezőre teszek. Első átgondolásra furcsának tűnik, hogy miért egyaránt pozitív előjellel veszem figyelembe saját és ellenfelem esélyeinek pontértékét is, de ha belegondolunk logikus, hogy annál többet ér nekünk egy mező elfoglalása, minél több helyzetet építünk saját magunknak, és egyszerre minél nagyobb veszélyeztetéstől védjük meg magunkat ellenfelünk esélyének rontásával.

Ilyen módon nem építek szabálybázist, az algoritmus mégis véd megfelelően beállított paraméterek estében az ellenfél támadásaitól, illetve lehetőséget ad a nyerésre. Nézzük meg például azt az esetet, ha az ellenfelemnek van egy darab egy oldalt nyitott négyese. Nem azt kódolom be egy szabállyal, hogy zárjuk ezt le, hanem amennyiben az 5 egymás mellett lévő bejegyzés (ebből egy virtuális, „mi lenne ha ide raknék" jel), tehát a célfeltételként megfogalmazott játékállás értéke a kiértékelő függvényben megfelelően nagy, a rendszer az ellenfél esélyének elrontása miatt lezárja a nyitott négyest, véglegesíti az ideiglenesen lerakott jelet. Más helyeken akár *gyilkos nyitott négyes* (két oldalt nyitott négyes) építésének értéke sem lehet ennél magasabb. Egy esetben kell, hogy magasabb értéket kapjuk, ha saját magunknak van esélyünk befejezni a játékot, tehát lerakni az ötödik jelünket egy sorba. Ebből látszik, hogy a támadó szellem megtartása érdekében saját esélyeink pontértékét egy korrekciós szorzóval, úgynevezett támadó szorzóval kell növelnünk.

A kiértékelő függvény felépítése:

E	Egyes esély darabszám
K	Kettes esély darabszám
H	Hármas esély darabszám
N	Négyes esély darabszám
O	Ötös - célállapot darabszám
DNS_1	1 egyedül álló bejegyzés értéke
DNS_2	2 azonos bejegyzés értéke
DNS_3	3 azonos bejegyzés értéke
DNS_4	4 azonos bejegyzés értéke
DNS_5	5 egymás melletti azonos bejegyzés értéke
s	Saját esély indexe
e	Ellenfél esélyének indexe
T	Támadószorzó

$$V=(E_s*DNS_1+K_s*DNS_2+H_s*DNS_3+N_s*DNS_4+O_s*DNS_5)*T+$$
$$+(E_e*DNS_1+K_e*DNS_2+H_e*DNS_3+N_e*DNS_4+O_e*DNS_5)$$

A kiértékelő függvény lehetne cizelláltabb is, hiszen nem mindegy, hogy a jelek rajzolata X-XX vagy XXX, illetve, hogy egy vagy két oldalon nyitottak-e. De mivel így akár 20 különböző esélyt is külön kezelhetnék, a tanulás túl sok paraméter állításával nagyon hosszadalmas lenne. Jó közelítéssel azonban ez a megoldás is optimális eredményt adhat számunkra, és csak 5 paramétert kell behangolnunk. Nézzük meg az 5 paraméter, hogy tudja lefedni az ennél sokkal magasabb számú különböző lehetséges játékhelyzetet.

Pirossal jelölve a virtuális bejegyzést, nézzük meg példaként a 13-as ábrán, hogy egy darab egy oldalt zárt, valamint egy darab mindkét oldalt nyitott 3-as között milyen különbségeket tesz ez az eljárás:

*OXXX =0 OXXX * =0 OXXX * * =3 OXXX * * * =2 OXXX * * * * =1

Összegzés: 1x3-as, 1x2-es, 1x1-es

* * XXX =3 * XXX * =3 * XXX * * =3 * XXX * * * =2 * XXX * * * * =1

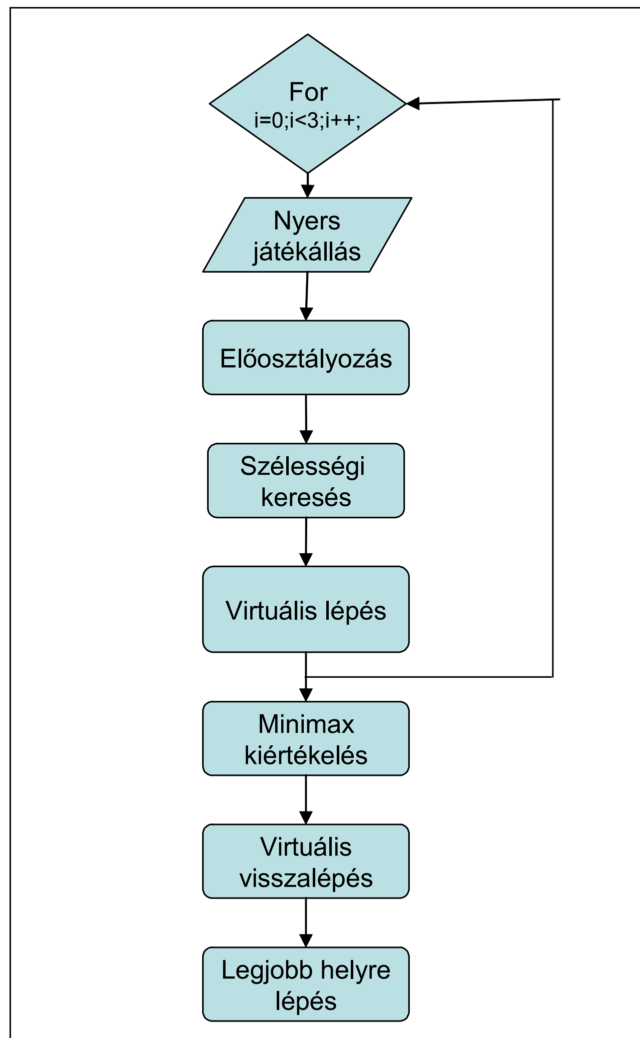
Összegzés: 3x3-as, 1x2-es, 1x1-es

13. ábra Játékállás kiértékelés

A 13. ábrán látható, hogy ugyan csupán egy paraméterrel jellemezzük a 3-as esélyt, mindkét oldalon nyitott 3-as esetén a paramétert 2-vel többször vesszük figyelembe a kiértékelő függvényben, így jelentősen magasabb értéket kapunk, mint egy oldalon zárt 3-as esetén. A következő lépés döntés meghozatalának algoritmusát a 14. ábra szemlélteti.

A minden üres mátrixmezőt csillagközéppontnak tekintő, és így a játékteret végigpásztázó függvényt az "amobaprg.cpp" állomány "CAmoba" osztályában található "Haszon" függvény implementálja. A fejlesztés során azonban észre vettem, hogy ezeket a képzeletbeli téglalapokat a mátrix közepén 9-szer ugyan abba a pozícióba fektetem, ezért a hasznosság kiszámolására új függvényeket implementáltam, amelyek csak egyszer fedik le a játékteret. Ezek a függvények a "Haszon5H" a horizontális irányban, "Haszon5V" vertikális irányban, "Haszon5Z" a főátló irányában, valamint a "Haszon5Y" a mellékátló irányában járja be a táblát, úgy, hogy amennyiben csak azonos jeleket talál a téglalap által lefedett 5

mátrixmezőben, a keresőmező minden egyes üres mezőjéhez hozzáadja ennek értékét a "Haszon1" illetve "Haszon2" tömb aktuális értékének alapján. Így a tábla a négy függvény által minden irányban való bejárása után előállnak ugyan azok az értékek, amelyeket a "Haszon" függvény állított elő, de csaknem tizedére csökkentve az erőforrás felhasználást.



14. ábra A következő lépés döntésének meghozatala

5.2.2.2. Minimax politika

Az algoritmus során minimax politikát követve 3 lépés mélységben vizsgálom a kialakult játékállást. Mivel a játékfa teljes szélességben való kifejtése felesleges, és nagyon sok erőforrást igényelne, először az aktuális játékállás szélességi keresését végzem el, és 10 leghasznosabb mezőjét fejtem ki tovább.

Amint az aktuális horizont 10 leghasznosabb mezőjét megkaptam, sorban ezek mindegyikére valóban lehelyezem bábumat, természetesen megjegyezve, hogy ezt vissza kell vonnom a játékfa kiértékelése után. Az így kialakult fiktív játékálláson ismét szélességi keresést végzek, és a 10 legígéretesebbnek látszó mezőt kigyűjtöm. Ezeket a pontokat sorban véglegesítve ismételten mélységi keresést végzek. Az így a 3 mélységű játéksíkban kialakult hasznossági értékek legjobbikát a minimax kiértékelés szabályai szerint visszaterjesztem a gyökér felé. Ez a gyakorlatban úgy történik, hogy a 3. szint értékeinek maximumát adom át a 2. szint számára, mivel ez saját játékosom szintje. A 2. szint minimumát adom át az 1. szint számára, mivel ez ellenfelem szintje, aki logikusan úgy választ, hogy nekem a lehető legrosszabb játékállást engedje át. Az 1. szinten így kialakult hasznosság értékek maximumát tartalmazó lépést kell ezek után meglépnem.

5.2.2.3. Az állapot kiértékelő függvény közelítése

Mint az előzőekben bemutattam, a kiértékelő függvény felépítését statikusan alakítom ki, tehát szerkezetét nem változtatom, csupán ennek súlyait keresem.

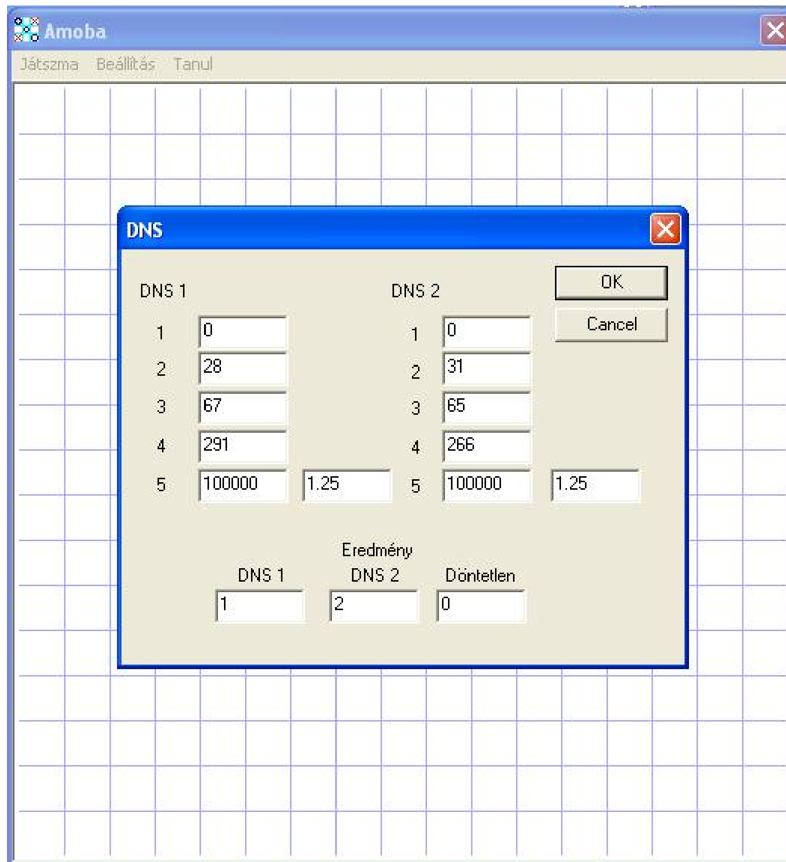
Keresési stratégiaként bakteriális algoritmust választottam az előosztályozással kapott egyszerűsített állapottér bejárására. Különböző hipotéziseket versenyeztetek, a játékot önállóan, egymás ellen játszva, amely segítségével megpróbálok külső tanító nélkül optimális megoldást találni a problémára.
A bakteriális algoritmusok jellemzője, hogy csupán egy szülő egyed létezik. Mivel az általam implementált rendszer egyszerre csupán két hipotézist tud összehasonlítani, hiszen a jósági függvény az, hogy egy kétszemélyes játékban ki nyer többször, nem lehet egy nagy populáció egyedeit egyszerre értékelni. Csak egy szülő, a játékban nyertes génkombináció lehet. Mivel egy szülőből nem lehet kereszteződéses szaporítást, csupán osztódást, és ezen új egyeden mutációt megvalósítani, a bakteriális algoritmus választása kézenfekvő.

A bakteriális algoritmus DNS-e, egy vektor a "Haszon1" illetve "Haszon2" integer tömb, amelyek a következő képen épülnek fel.

- 1 egyedül álló bejegyzés értéke
- 2 azonos bejegyzés értéke

- 3 azonos bejegyzés értéke
- 4 azonos bejegyzés értéke
- 5 egymás melletti azonos bejegyzés értéke

Az egymás ellen versengő DNS-ek értékeit a tanulási folyamat során a 15. ábra szemlélteti.



15. ábra DNS vizualizálása a tanulási folyamat során

Ezek az értékek a játékállás kiértékelő függvény súlyai, amelyeket bakteriális algoritmus alapján folyamatosan változtatok, és keresem az optimális értékek kombinációját.

A bakteriális algoritmusokra jellemző módon a "Genetikus" függvény az előző lépésben nyertes génkombinációt lemásolja, majd ezen mutációt hajt végre.

Szimulált lehűtést alkalmazva a mutáció során történő véletlenszerű paraméterváltoztatások mértékét folyamatosan csökkentem, így a rendszer egyre finomabb hangolását elérve. A szimulált lehűtést az biztosítja, hogy a mutációt végző "Genetikus" függvény második bemenő paramétereként átadjuk a "lehut"

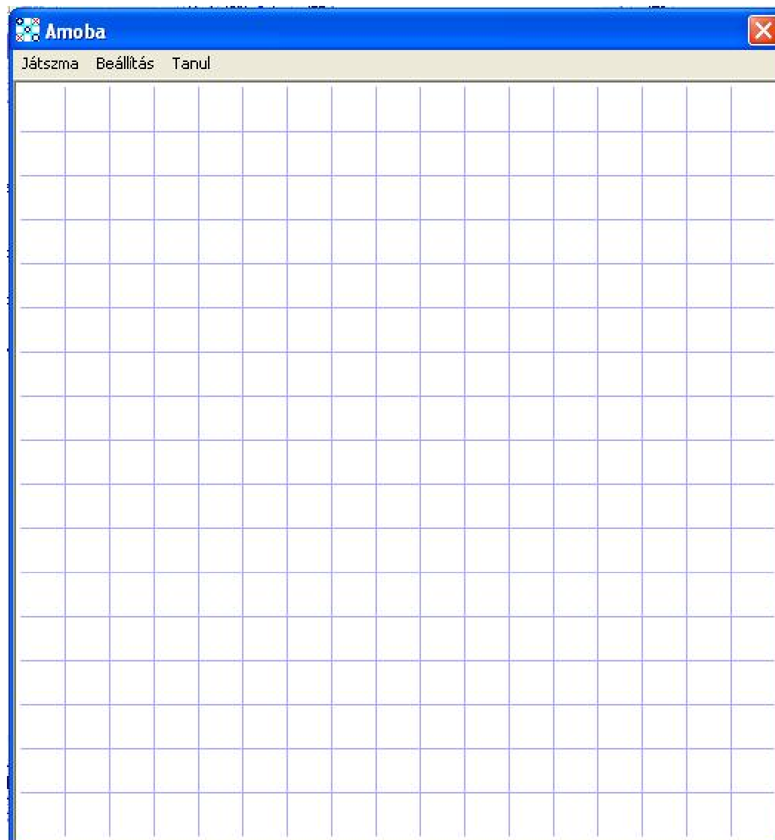
lebegőpontos változó értékét, amely 1-el kezdődik a tanulási folyamat elején, és 0-hoz közelít a tanulási folyamat végén. Ez tehát biztosítja azt, amit a szimulált lehűtés elméletének ismertetésénél megemlítettem, hogy a kezdeti nagy ugrásokkal megtaláljuk az optimális megoldáshoz vezető legjobb utat, majd ezt az optimalizálás végéig egyre kisebb lépésekkel finomítsuk. E nélkül túl nagy lépésközöket választva a keresési térben folyamatosan ugrálnánk, míg túl kis lehetséges lépésközt választva a beállás sokkal tovább tartana.

A mutáció lehetséges mértékét az határozza meg, hogy a "Haszon1" illetve "Haszon2", tehát a DNS-t reprezentáló tömb szomszédos elemei között mekkora a távolság. A mutációjánál az algoritmus megnézi az aktuális paraméter előtt lévő tömbelemhez viszonyított távolságát, és a rand() függvény segítségével generált, 0 és a kiszámolt távolság közötti véletlen számot levonja az eredeti paraméterből, természetesen a szimulált hűtést megvalósító változóval korrigálva. Ugyanezt a műveletet pozitív irányban is elvégzi, vagyis az aktuális elem, valamint a következő elem közötti távolságot maximális értéknek tekintve, generál egy véletlen számot, amelyet most az aktuális elemhez hozzáad. Ezzel az eljárással a DNS elemek közötti távolság véletlenszerűen változtatható új hipotézist kialakítva, ami szavatolja, hogy az egymás melletti DNS értékek szigorúan monoton növekedő sorban helyezkedjenek el. Ennek magyarázata ugyancsak egy logikus tapasztalati tényből fakad, miszerint minél több jel helyezkedik el egymás mellett, annál nagyobb ennek hasznossága a célfüggvény szempontjából, vagyis hogy 5 darabot gyűjtsünk egymás mellé. Amennyiben a DNS értékeinek szigorú sorrendiségét nem kötnénk ki, most sem követnénk el logikai hibát, csupán jelentős számú olyan hipotézist vizsgálnánk meg, játszmák 100-ait feleslegesen lejátszva, amelyek biztosan nem lehetnek számunkra optimális megoldások, ezzel pazarolva az erőforrást.

A kiértékelő függvény súlyai tehát úgy alakulnak ki a tanulási folyamat végére, hogy a hipotézisek egymás ellen versenyeznek, az egymás elleni meccsek eredményéből nyertesen kijövő DNS lemásolásra kerül, és egyre kisebb mértékű mutációt hajtunk rajta végre. Az új hipotézist újra versenyeztetjük a szülő egyeddel, mindaddig, míg a tanulási ciklus végére nem érünk. Az utolsó fázisban nyertes génkombináció lesz a kiértékelő függvényünk paraméterlistája.

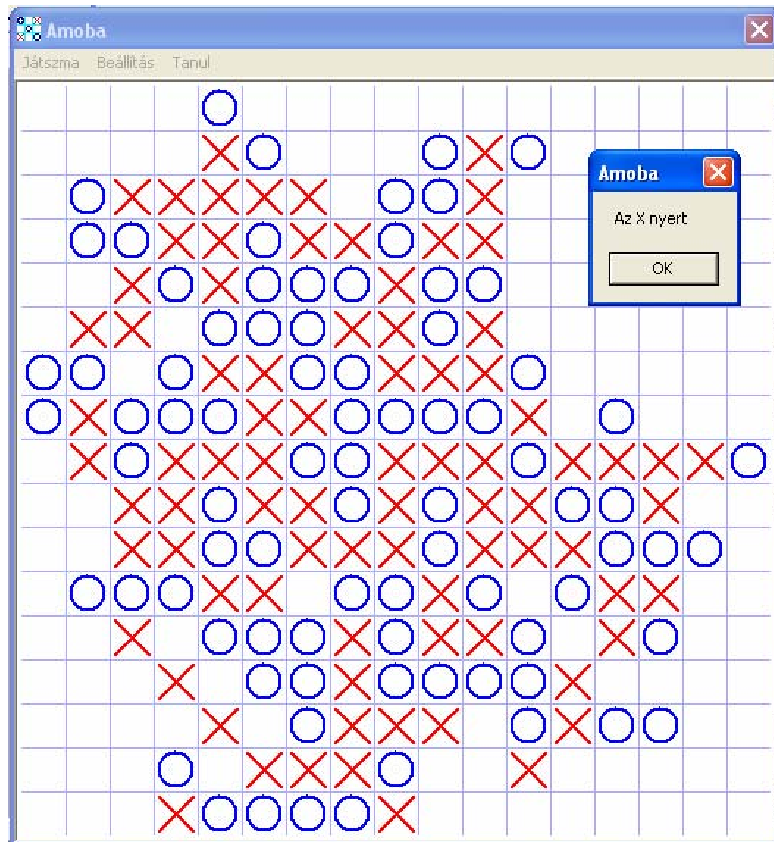
5.3. A program felhasználói felületének ismertetése

A program indítása után a 16. ábrán látható kezdő képernyő jelentkezik, amely a hagyományosan az iskolapadokban unalmas órákon "kockás" papíron, megszorító szabályok nélkül játszott játékteret modellezi.



16. ábra Kezdő képernyő

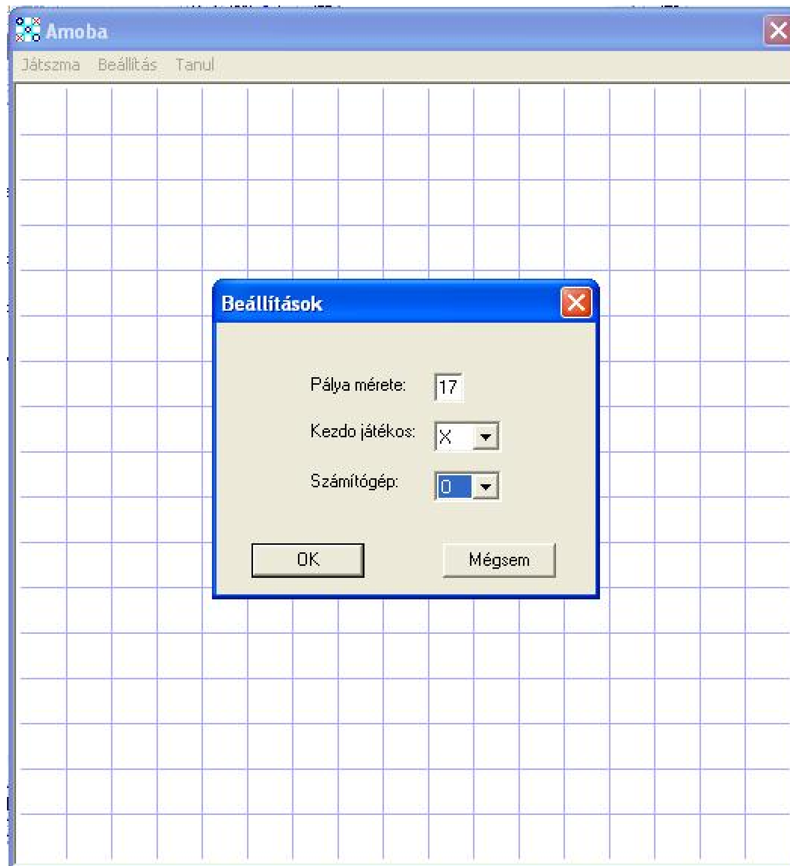
Az egérmutató helyén lévő négyzetbe bal klikkel letehetjük a kezdő jelet, így rögtön játszani kezdhetünk a gép ellen. Ekkor viszont még csak az előzetesen fixen bekódolt DNS hipotézis ellen játszunk, a tanulást külön dialógusablakban tudjuk elvégeztetni. A játékosok felváltva piros X, illetve kék O jelet helyezhetnek el a táblára, amennyiben valakinek összegyűlik 5 egy sorban, illetve nem marad több üres játékmező, a játék leáll, és figyelmeztető ablak értesít bennünket az eredményről, amelyet a 17. ábrán láthatunk.



17. ábra Nyertes játékállás figyelmeztető ablak

A menü "Játszma" pontja az "Új játék", valamint "Kilépés" almenüpontokat tartalmazza. Az "Új játék" almenüpontra kattintva a képernyő törlődik, és az aktuális játék befejezése nélkül, új játékba kezdhetünk. A "Kilépés" menüpont kiválasztása a programból való kilépést eredményezi.

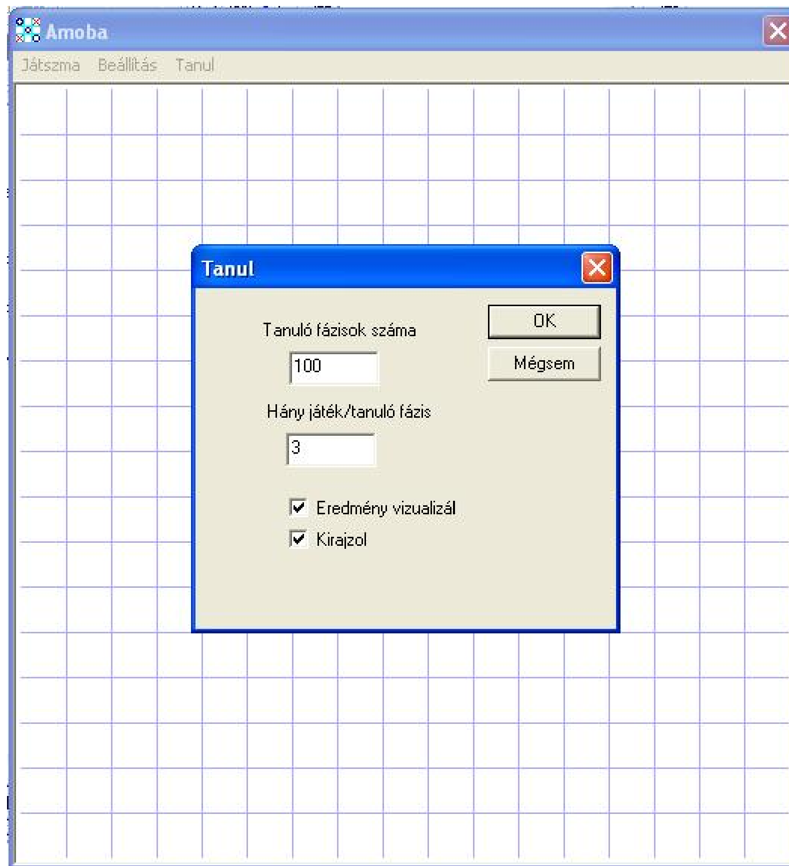
A "Beállítás" menüpont hatására a „Beállítások” dialógus ablakot aktiváljuk, amelyet a 18. ábra szemléltet. Ezen dialógusablakban megváltoztatható az alapértelmezésben 17 x 17-es játéktér mérete 5 és 25 értékhatárok között. Ugyancsak megváltoztatható a kezdő játékos, valamint a számítógép elleni játék esetében a számítógép jelölése. A kezdő játékos megváltoztatásával tesztelhető, hogy mennyivel erősebb az aktuális hipotézis, amennyiben a kezdő játékos előnye a számítógép mellett áll. Alapértelmezésben az emberi játékos kezd.



18. ábra Beállítások dialógus ablak

A "Tanul" menüpont segítségével érhetjük el a 19. ábrán bemutatott Tanul dialógusablakot, amelyet a "Learn.cpp" állományban implementáltam. Ebben a dialógusablakban meghatározhatom a tanuló ciklusok számát, amely azt jelenti hány új a bakteriális algoritmus által felállított hipotézist hasonlítson össze az előző ciklusban nyerő DNS-el. A ciklusban előre haladva a változtatás mértéke a szimulált lehűtéssel folyamatosan, a ciklus hosszának megfelelően lineárisan csökken.

A második paraméter megadásával meghatározhatom, hogy hány játszmáig tartson a két hipotézis mérkőzése. Alapértelmezésben ez a szám három, de a véletlen szerepe annál jobban csökkenthető, minél nagyobbra veszem. Ezzel párhozamosan, minél nagyobb ez a szám, annál hosszabb ideig tart ugyan annyi tanuló ciklus lefutása.



19. ábra Tanul dialógusablak

A dialógusablak alján két jelölőnégyzet bejelölésével kívánság szerint a tanulási folyamat során kialakuló játszmákat kirajzoltathatom a játéktérre, illetve az eredményt vizualizálhatom mérkőzésenként, amely megmutatja melyik hipotézis hány játszmát nyert mérkőzésenként, és lépésről lépésre figyelemmel követhetjük az aktuális legjobb DNS-ünk paramétereinek változását. Az eredmény vizualizálását a "LearnResults.cpp" állományban implementált "DNS" dialógusablak oldja meg. Ez a dialógusablak megmutatja a két egymás ellen versenyző DNS értékeit, a támadószorzót, valamint kettőjük mérkőzésének eredményeit, tehát hogy hány játszmát nyert DNS1, DNS2, illetve hány döntetlen született. Az "DNS" dialógusablakot a 15. ábra mutatja be.

5.4. Futási eredmények, összegzés

Legyen egy vektor koordináta minimális értéke 0, maximális értéke 1000. Az állapotterem tehát, amelyben a megoldást keresem 1000^5 méretű. A DNS első értékét véletlenszerűen határozom meg, de mint már említettem, egy kis heurisztikát is beleszövök az erőforrásokkal való takarékosság szempontjából. Ez azt jelenti, hogy úgy választom a kezdő DNS-t, valamint a folyamatos mutációk során a következőket, hogy mindig tartsam az egymás után következő vektorkoordinátákat szigorúan monoton emelkedő sorban.
Mivel 5 egymás melletti azonos bejegyzés a játék célfüggvényével egyenlő nyerő állás, ezt a paraméterértéket akár végtelennek is tekinthetném. Legyen ez a „végtelen" számosítva a numerikus műveletek elvégezhetősége érdekében, ezért válasszuk a lehetséges paraméterértéknél nagyságrendekkel nagyobb 100.000-et. Ezzel négyre redukáltuk a keresendő paraméterek számát.
Még egy paramétert kilőhetünk, ez pedig az 1-es esély. Ezt az értéket is nyugodt lélekkel lefixálhatjuk előre, mivel ez lesz a minimális érték, amelyhez képest a többit keressük. Legyen ez az érték az első futtatásnál 0, mivel valószínűleg semmi értékkel nem bír, ha a valódi játéktértől távol egymagában lerakunk egy jelet. Ennek értéke a csillag kiértékelés szerint 20-szorosa lenne az 1-es esély paraméterének, amelyet szinte minden üres játékmező minimálisan megkapna.
A két kilőtt paraméter miatt a keresési teret 1000^3 méretűre csökkentettem.
Egyszerre mindig csak egy osztódást és mutációt végzek. Három virtuális játszmát játszva a két hipotézissel egymás ellen, tehát a két különböző DNS-el, a megerősítés alapján, vagyis hogy melyik haszonfüggvény nyert több játszmát eldöntöm, hogy az eredeti hipotézisemet, vagy az újat tartom-e meg, aszerint hogy melyik képvisel nagyobb játékerőt.

A tanulási algoritmus ezek szerint való futtatásának eredményét a 7. táblázatban foglaltam össze. Az alábbi eredmények tehát 1000 tanulási ciklus alatt születtek, 3 játszma / mérkőzés, maximális értékként 1000 paraméterekkel.

DNS 1	0	0	0	0	0	
DNS 2	142	150	105	107	96	0,1758
DNS 3	502	481	359	313	334	0,5828
DNS 4	827	688	636	592	670	1,0000
DNS 5	100000	100000	100000	100000	100000	
Támadószorzó	1,38889	1,30435	1,38889	1,03448	1,282	

7. táblázat Futási eredmények 1

Egy tesztfuttatás futási ideje kb. 15 perc volt. A kapott hipotézisek jól látható módon közelítenek egy hipotetikus megoldáshoz. A későbbi eredményekkel való összehasonlíthatóság miatt az utolsó oszlopban a DNS 4. értékét egységnyinek tekintve indexeltem a keresett paramétereket.
A tesztjáték alkalmával a játék egyes helyzetekben jó választ ad, viszont kiismerhető módon, néhány helyzetre adott logikailag hibás válasz miatt minden alkalommal megverhető.

Most nézzük meg, hogyan változik a futási eredmény, amennyiben az egyes esély értékét 0 helyett 1-re változtatom. A többi beállítási paraméter változatlan.

DNS 1	1	1	1	1	1	
DNS 2	189	198	130	114	199	0,2387
DNS 3	420	462	450	326	405	0,5933
DNS 4	710	731	718	604	714	1,0000
DNS 5	100000	100000	100000	100000	100000	
Támadószorzó	1,07914	1,282	1,05741	1,17352	1,23512	

8. táblázat Futási eredmények 2

A kapott hipotézisek ebben az esetben is konvergálnak egy hipotetikus megoldáshoz, a tesztjáték során viszont ugyanaz tapasztalható, mint az előző esetben, vagyis néhány azonos típusú helyzetre az ágens válasza logikailag hibás. A hibás szituációkat, és az eredményként kapott DNS-eket összevetve a probléma az, hogy a DNS 4 értéke nem elegendő mértékben nagyobb, mint a DNS 3 értéke, és egyszerre több 3-as esély építésével, illetve ellenfél kettesének lezárásával az ágens kétoldalt nyitott négyes építését engedi meg számomra, mivel ennek hasznossága a paraméterek szerint alacsonyabb.

Az utolsó tesztelésnél a paraméterek keresésének terét kibővítem 1000-ről 3500-ra, hogy elkerüljük azt a problémát, hogy a kialakult arányok azért nem változtathatóak, mert túl szűk a keresési tartomány. Az eredményeket a 9. táblázat foglalja össze.

DNS 1	1	1	1	1	1	
DNS 2	416	422	448	487	429	0,1454
DNS 3	1665	1669	1257	2104	1532	0,5431
DNS 4	3068	3081	2474	3529	2995	1,0000
DNS 5	100000	100000	100000	100000	100000	
Támadószorzó	1,03448	1,30435	1,02041	1,15385	1,07914	

9. táblázat Futási eredmények 3

Az eredmények most is konvergálnak egy hipotetikus megoldáshoz, de mint a DNS-ek aránya előre vetíti az előző tesztelésekből, most sem képvisel túlságosan nagy játékerőt ágensünk.

Mivel a kialakított rendszer alkalmatlan arra, hogy a játék végén elemezze a játszmát, megállapítsa hogy mi vezetett nyeréshez, illetve vereséghez, ezért fordulhat elő az, hogy az implementált tanulási módszer nem a legoptimálisabb eredményt adja. A két egymás ellen játszó hipotézis mindig mohó politikát folytat, csak az azonos értékű lépések között választ véletlen szerűen, ezért ugyan úgy alakulhat egymás után több játék is, ugyan azt a hibát többször elkövetve. Így fordulhat elő, és a mérések bizonyítása szerint ez általában így is történik, hogy emberi mércével véve erősebb játékerőt képviselő paraméterekkel rendelkező értékelő függvény mégis néhány esetben, azonos típusú játékállások kialakulása esetén, alulmarad gyengébb társával szemben.

Leszűrhető tehát a mérési eredményekből valamint a fejlesztés során szerzett tapasztalatokból, hogy két egymás ellen játszó intelligens ágens segítségével, amelyek a folyamatosan véletlenszerűen változtatott hipotézisek szerint cselekszenek, a rendszer ugyan konvergál egy megoldáshoz, de a megfogalmazott cél, az emberi játékos ellen való eredményes fellépés szempontjából ez a megoldás nem mondható optimálisnak.

A program egyszerű átalakításával emberi tanító állhatna az egyik gépi játékos helyére, így sokkal jobb eredményt találhatnánk a keresett génkombináció helyére, kiküszöbölve a gépies, ugyan olyan kimenetelű játszmák kialakulását. Ebben az esetben csak akkor változtatnám az optimális génkombinációra tett hipotézisemet, amennyiben az emberi játékos nyer. De mivel így is csak véletlenszerűen járnám be a keresési teret, számos játszmára lenne szükség egy optimális eredményre akadáshoz, amely a számítógép számára talán kellemes időtöltés, viszont az emberi játékos bizonyára hamar elunná magát, ha nem pénzre megy a csata.

A tapasztalatok azt mutatják, hogy a minimax politika érvényesíthetősége korlátozott az amőba játék esetében. Természetesen amennyiben a teljes játékfa kiértékelésére lehetőség nyílna, abban esetben a minimax politika biztos nyerő stratégiát jelentene. A tapasztalatok viszont azt mutatják, hogy az általam felvett heurisztikus kiértékelő függvény szerkezet mellett, a játékos saját hasznát abban a mélységben szeretné maximalizálni, ameddig ellát. Ez ugyebár hasznos lehet, amennyiben az ágens nyerő játékállást talál ebben a mélységben, mivel ez azt jelenti, hogy ugyan a teljes játékfa nem áll rendelkezésre, de a minimax kiértékelést végrehajtva nyerünk. Így már nem is vagyunk kíváncsiak a fa többi részére. Ha viszont a játékfa levelei a heurisztikus kiértékelő függvényből adódó hasznosság értékeket tartalmazzák, amely alacsonyabb, mint a nyerő játékállás értéke, és ennek maximalizálására törekszünk, előfordul, hogy az ágens hasznának maximalizálása érdekében az aktuális lépésben direkt enged teret az ellenfél helyzetépítésének, hogy a haszon maximalizálásnál figyelembe vett játéksíkban lezárva ezt nagy hasznot kasszírozhasson be. A mellékletben található programverzió csak három mélységi szintet vizsgál, de a fejlesztés során kísérletet tettem 5 szintű minimax játékfa kiértékelésre is. A 3 mélységű kiértékelésnél még meg tudtam valósítani olyan feltételt, amely nem engedi tovább a kiértékelést mélyebb szintre, ha az első szinten saját, vagy ellenfele nyerő stratégiájának pontértékét jelzi a haszon kiértékelés. 5 szinten ugyanezt mát nem tudtam megoldani. Őszintén szólva valószínűleg a kiértékelő függvényem kezdeti koncepciója miatt nem alkalmazható eredményesen a magasabb mélységű keresés, mivel egy lépés mélységben valóban csak a következő lehetséges lépések környezetének közelében lévő hasznosság értékek számítanak, de mélységi keresésnél a teljes játékállásokat kellett volna kiértékelni. Erre jó példa lehet, hogy a sakkban könnyen leüthetem ellenfelem lovát a vizsgált horizontban, amennyiben előtte vezéremet ajánlottam fel leütésre. Amennyiben viszont a teljes játékállást értékelem, vagyis leszámolom saját és ellenfelem tisztjeit, gyalogjait, valamint ezek pozícióját stratégiai szempontból, azonnal más lépés indikálódik.

Összefoglalva az elért eredményeket azt kell mondanom, hogy egy órákon sokat unatkozó középiskolás diák könnyen megtalálhatja az amőba programom 1000 tanítási fázis után kialakult intelligens ágense elleni nyerő stratégiát, de ugyanakkor az alkalmazásból szigorúan játékbeli képességeit tekintve amerikai elnök könnyedén válhatna.

A program továbbfejlesztésének lehetősége, hogy a nyers játékállásokat használva bemenetként a lineáris kiértékelő függvény helyett neurális hálózatot alkalmazok, TD(λ) tanítási módszerrel, amely ugyan sokkal hosszabb futással, de sokkal jobb megoldást szolgáltatna a problémára.

Melléklet

1. melléklet Jelölések

t diszkrét idő

T az epizód utolsó periódusa

s_t állapot a *t*-edik időpillanatban

a_t akció a *t*-edik időpillanatban

r_t jutalom a *t*-edik időpillanatban *(s_t,a_{t-1},s_{t-1}* függvénye)

R_t a *t*-edik időpillanatbeli (az összegyűjtött diszkontált) hozam

$R_t^{(n)}$ *n*-lépés hozam

R_t^{λ} *λ*-hozam

π politika (a döntéseket meghatározó szabályok)

$\pi(s)$ akció választás az *s* állapot és determinisztikus π politika esetén

$\pi(s,a)$ az *a* akció választási va1őszinusege az *s* állapot és sztohasztikus π politika esetén

S a nem terminális állapotok halmaza

S^+ az összes állapot halmaza (a terminális állapotokkal együtt) $\mathrm{A}(s)$

az összes lehetséges *s* állapotbeli akciók halmaza

$P_{ss'}^{a}$ *s*' állapotba kerülés valószínűsége *s* állapot és *a* akció esetén

$R_{ss'}^{a}$ a varható jutalom *s*-ből *s'* állapotba jutáskor *a* akció mellett

$V^{\pi}(s)$ az *s* állapot értéke π politika esetén (varható hozam)

$V^{*}(s)$ az *s* állapot értéke optimális politika esetén

V, V_t V^{π} vagy V^{*} közelítése

$Q^{\pi}(s,a)$ az *a* akció értéke *s* állapot és π politika esetén

$Q^{*}(s,a)$ az *a* akció értéke *s* állapot és optimális politika esetén

Q, Q_t Q^{π} vagy Q^{*} közelítése

$\vec{\theta}_t$ V_t -hez vagy Q_t -hez tartozó paraméterverktor

$\vec{\phi}_s$ *s* állapotot reprezentáló vektor

δ_t	átmeneti-differencia hibaértéke a *t*-edik időpillanatban
$e_t(s)$	„emlékeztető nyom" (eligibility trace) *s* állapotban *t*-edik időpillanatban
$e_t(s,a)$	emlékeztető nyom az állapot-akció párra
γ	diszkontálási paraméter
ε	véletlen akció választási valószínűség ε-mohó politika esetén
α, β	lépésköz paraméter
λ	emlékeztető nyom paramétere

2. melléklet Hardware és Software-architektúra

A tesztfuttatásokat a következő hardwarekörnyezetben végeztem:
MSI K7 Mainboard, 400Mhz rendszerbusz, NVIDIA nForce 2 chipset
AMD Athlon 2800+ processor
768 MB DDR RAM

Operációs rendszer:
Windows XP Professional SP1

Fejlesztőkörnyezet:
A rendszer implementációját Microsoft Developer Studio 6.0 fejlesztőkörnyezettel C++ nyelven végeztem.

Software-architektúra:
Az implementáció során a modularitásra törekedtem, hogy a program lazán összefüggő modulok együttese legyen a könnyű módosíthatóság, karbantarthatóság érdekében. A software állományai, és ezek tartalma:

Amoba.h
Ez az alkalmazás fő header fájljai. Ez tartalmaz egyéb szükséges header fájlokat, valamint deklarálja a CAmobaApp osztályt.

Amoba.cpp
Ez a fő alkalmazás forrásfájlja, amely a CAmobaApp osztályt implementálja.

Amoba.rc
Ez a fájl tartalmazza az összes Microsoft Windows erőforrást amelyet a program használ. Tartalmazza az ikonokat, amelyet a RES alkönyvtárban helyeztünk el. Ezt az állományt közvetlenül lehet szerkeszteni a Microsoft Developer Studioval.

Amoba.clw
Ez a fájl a tartalmazza az információkat a ClassWizard számára, hogy a meglévő

osztályainkat szerkeszthessük, illetve új osztályokat hozhassunk létre. A ClassWizard ezt a fájlt használja arra is, hogy olyan információkat tároljon benne, amelyek dialógus ablakok létrehozásához, illetve szerkesztéséhez szükségesek.

AmobaFrm.h, AmobaFrm.cpp
Ezek a fájlok tartalmazzák a CAmobaFrame osztályt, amely a CFrameWnd osztályból származik, amely alkalmazásunk kezdő ablaka lesz. Ezek a fájlok tartalmazzák a perifériáról érkező akciók lekezelését, mint a menükiválasztás, egérkezelés.

amobaprg.h, amobaprg.cpp
Ezek a fájlok tartalmazzák a gépi gondolkodás, valamint a tanulás során használt genetikus algoritmus implementációját.

SetupDlg.h, SetupDlg.cpp
Ezek a fájlok a játék paraméterek beállítására használatos dialógusablakot implementálják, amelyben megadható a táblaméret, a kezdő játékos, valamint a számítógép jele.

Learn.h, Learn.cpp
Ezek a fájlok a tanulási paraméterek beállítására használatos dialógusablakot implementálják.

LearnResults.h, LearnResults.cpp
Ezek a fájlok a tanulási eredmények vizualizálását megvalósító dialógusablakot implementálják.

StdAfx.h, StdAfx.cpp
Ezek a fájlok arra szolgálnak, hogy előfordítási header fájlt (PCH) hozzanak létre Amoba.pch néven, valamint egy előfordított fájlt StdAfx.obj néven.

Resource.h
Ez egy standard header fájl, ami az erőforrás azonosítókat tartalmazza. A Microsoft Developer Studio ezt a fájlt folyamatosan frissíti a fejlesztés során.

Irodalomjegyzék

1., Stuart J. Russell-Peter Norving: Mesterséges Intelligencia modern megközelítésben, Panem-Prentice Hall, 2000

2., Futó Iván: Mesterséges Intelligencia, Aula Kiadó, 1999

3., Negnevitsky, M.: Artificial Intelligence: A Guide to Intelligent Systems, Addison Wesley, Pearson Education Limited, 2002

4., Retter Gyula: Fuzzy, Neurális, Genetikus, Kaotikus rendszerek – Bevezetés a „lágy számítás” módszereibe, Invest-Marketing Bt., 2003

5., Jenny Raggett-William Bains: Mesterséges intelligencia A-Z, Akadémiai Kiadó,1994

6., Mérő László: Észjárások – A racionális gondolkodás korlátai és a mesterséges intelligencia, Tericum Kiadó, 1997

7., Dr. Dudás László: Mesterséges intelligencia módszerek, ME, Gépészmérnöki kar, Informatikai Intézet oktatási segédlet, 1999

8., Fekete István, Gregorics Tibor, Nagy Sára: Bevezetés a mesterséges intelligenciába, LSI Oktatóközpont oktatási segédlet, 1999

9., R.S. Sutton and A.G. Barto. Reinforcement Learning. MIT Press, 1997.

10., Gerald Tesauro, Temporal Difference Learning and TD-Gammon, Communications of the ACM, 1995.

Summary

The Go-moku is one of the most popular and in its rules it is one of the simplest board game. Originally it was played on a 19x19 Go board but nowadays it's played on checked notes. The players put their marks after each other into one of the empty fields. The winner is who can collect five of his or her own marks in a horizontal, vertical or diagonal row. Although the game simple in its rules, to make an algorism on it could be much harder.

This paper provides examples of computer algorisms and practical difficulties on machine learning methods especially the reinforcement learning, and evolutional techniques by developing an adaptive logical computer game of Go-moku.

In the second chapter of this paper, we can find the main points of the theory of the machine learning. In the third chapter I am depicting the basics of methods and algorithms of the reinforcement learning. In the fourth chapter the game-three evaluation of the 'minimax' method is detailed in a two player full information game.

Based on this knowledge in the fifth chapter I am presenting a teaching method and the result of the software running on the Go-moku player problem.

The essence of the learning methods is that a static structure linear heuristically validating function's peeks are finding in a way when the two heuristic hypothesis of the process are playing against each other. By using bacterial method, I keep the one hypothesis, which won the more games, I copy it, and by using simulated cooling I mutate it to get the optimal approach.

Since the Go-moku game's rough states give far too wide state space by pre-filtering the game states I create a simpler state space. This state space can be more easily overviewed by the bacterial algorithms, and by finding the validating function's five parameters, a strong agent can be created against a human player.

To develop the software later I can use TD(λ) learning method on neural networks instead of the linear validating function, where without pre filtering the input could be the rough state of the board. In that case with a longer running time a much better solution of the problem could be found.