Budapesti Corvinus Egyetem

Operációkutatás

Mérgezett csoki játék (Chomp game) fejlesztése és betanítása python program nyelven

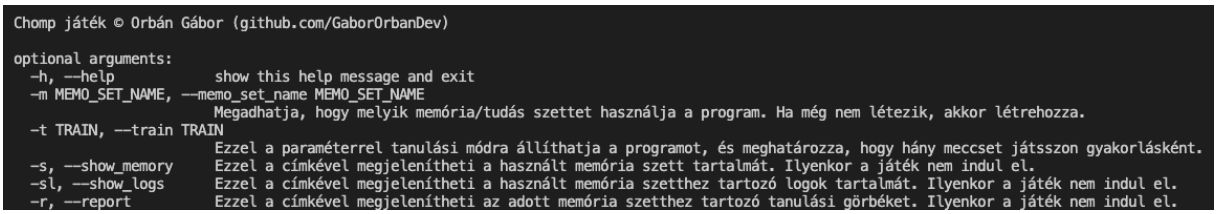
- szorgalmi feladat -

Orbán Gábor
Neptun-kód: Q4GPVC

Budapest, 2023

Program működésének a bemutatása

A fejlesztett program egy parancssoros felhasználói felülettel rendelkező alkalmazás, amely alkalmas arra, hogy a felhasználó különböző tapasztalati szinttel rendelkező tudásszetteket (a program kódban memória szettnek nevezem) hozzon létre, és ezek ellen játszhassa a mérgezett csoki játékot. A betanítás és a játék mellett a felhasználó lekérdezheti a tudás szetthez tartozó paraméterek értékét, a tanulási folyamat log állományát, illetve az ahhoz tartozó tanulási görbéket, azaz az egyes helyzetekhez tartozó választási lehetőségek valószínűségének a változását a tanulás során.



A használható paraméterek listája (képernyőkép a konzolról)

A tanulási folyamat működése

A tanulási folyamat implementálásakor az emberi tanulás folyamatát próbáltam utánozni. A folyamat elvi lényege, hogy a számítógép a játék során eseményekkel találkozik, ezekhez az eseményekhez a játék végén tapasztalatokat társít, a tapasztalatokat pedig emlékek formájában eltárolja.

A program szempontjából az esemény az, amikor saját maga, illetve a játékos letör egy kockát a tábla csokoládéból. Az eseménnyel kapcsolatban 3 adatot rögzít a rendszer:

1. ki lépte az adott lépést: számítógép/játékos
2. milyen választásai lehetőségei voltak: a letörhető kockák koordinátái
3. melyik lépést választotta: a letört kocka koordinátája.

A játék végén a program kiértékeli az eseményeket. Az egyes eseményekkel kapcsolatban 3 dolgot vizsgál az algoritmus, hogy meghatározza, hogy az adott eseményhez negatív vagy pozitív tapasztalatot (visszacsatolást) társítson:

I) van-e emléke arról, hogy hasonló tábla helyzettel találkozott-e már
II) az adott lépés a győzelméhez vagy a vereségéhez segítette hozzá
III) az vizsgált esemény magához vagy az ellenfélhez kötődik-e.

Ha a program még nem találkozott az adott helyzettel, akkor a tapasztalat függvényében létrehoz egy új emléket, ha azonban már van emléke az adott helyezettel kapcsolatban, akkor a tapasztalat függvényében szerkeszti azt. Minden egyes emlék 3 párhuzamos listából áll, az adott lista indexeken ugyanarról a lépésről tárol információkat. Az első lista az adott helyzetben választható lépéseket, a második lista az adott lépéshez tartozó súlyt, a harmadik lista pedig az adott lépéshez tartozó nettósított számlálót tartalmazza.

Mikor a számítógép először találkozik az adott helyzettel, akkor az egyes választási lehetőségek közül egyenlő valószínűséggel választ. A súlyok feladata - miután megszületik az első tapasztalat az adott helyzettel kapcsolatban és emlék lesz belőle - hogy az egyes lépések bekövetkezési valószínűségét a tapasztalatok alapján növelje vagy csökkentse. A súlyok értékének kiszámításához a lépésekhez tartozó nettósított számlálókat használja az algoritmus.

A számláló értéke 1-gyel növekszik, ha az adott lépés győzelemhez vezette a játékost vagy a számítógépet, és 1-gyel csökken, ha veszteséghez; majd ezután egy hasznossági függvény segítségével, ami jelen esetben egy transzformált szigma függvény, a program kiszámítja a lépéshez tartozó súlyozást, ami meghatározza, hogy a későbbiekben mennyire tekinti az adott lépést előnyösnek sajátmagára nézve a számítógép. A transzformált szigma függvény képlete, ahol az x egyenlő a nettósított számlálóval és w egyenlő a számlálóhoz tartozó súllyal:

$$w(x) = \frac{2}{1 + e^{\frac{-x}{3,5}}}$$

Az adott hasznossági függvényről részletesebben a következő fejezetben fogok írni.

Az emlékeket a program futáson kívül egy bináris file-ban tárolja. Az emlékek kezelésével kapcsolatban fontos fejlesztési irányelv volt, hogy nem a tanulási ciklus végén, hanem minden egyes játék (iteráció) után az új és módosított emlékeket az alkalmazás mentse az adott file-ba, azért, mert ha a futás közben hiba történne, és leállna a program, maximum az utoljára játszott játék tapasztalata veszne el akkor. Ez magas iterációs számú betanítás mellett kritikus lehet.

A verhetetlenség kérdése – tanítási tapasztalatok

Az alkalmazás fejlesztése során többféle módon próbálkoztam betanítani a programot a játékra. A tanítás során 3 fő problémával/kérdéssel szembesültem:

a) mi legyen a program hasznossági függvénye
b) hogyan kapjon a számítógép megoldandó helyzeteket
c) hány játékot játsszon a gép, hogy képes legyen a saját hibáit korrigálni és emellett haladó szinten játszani.

A) Hasznossági függvény meghatározása

A hasznossági függvény a tanulási algoritmus lelke, hiszen ez határozza meg, hogy az egyes lépésekhez milyen súlyt (hasznosságot) társít a program.

Először a súlyokat egy meghatározott aránnyal (továbbiakban: tanulási ráta) történő szorzásával próbálkoztam, így pozitív esetben a súlyokat a tanulási ráta arányával felszázalékoltam, negatív esetben pedig leszázalékoltam volna. Ennek a módszernek az előnye az, hogy a használt függvény alulról korlátos, valamint több egymás utáni javítás vagy rontás esetén a súlyokat nem lineárisan növeli/csökkenti. Mellékhatása azonban a viszonylag gyorsan bekövetkező túltanulás, azaz, hogy a játékszám növelése mellett a játékban mutatott képessége a számítógépnek egy ponton túl visszaesik. Ennek azaz oka, hogy a függvény felülről nem korlátos, egyes lépések súlyát túlértékelheti az alternatívákkal szemben, így kevésbé lesz „kreatív” az adott esetekben. A túltanulás különösen kritikus, ha egy stratégiailag rossz lépést jónak ítélt a rendszer, és eszerint sajátította el, mivel akár több ezerszer kell hibáznia vele, hogy megfelelően leértékelje.

A túltanulást, hogy lecsökkentsem kísérleteztem felső korlát bevezetésével, illetve korlátolt lineáris függvény használatával. A lineáris függvénnyel kapcsolatban azt találtam problémának, hogy nem veszi figyelembe azt, hogy máshogy kell súlyozni egy új esetet, mint egy relatíve régi, jól betanult esetet.

Végül a tárgy előadójával való konzultálás után úgy döntöttem, hogy bevezetek egy számlálót az egyes lépésekre, és a számláló értékét egy függvény segítségével kivetítem egy korlátos

intervallumba (ez a függvény értékkészlete). Erre a célra a szigma függvényt választottam ki, azonban a transzformációk nélküli függvény túl hamar megközelíti az alsó- illetve felsőkorlátját. Ezért igyekeztem úgy a függvény képét horizontálisan és vertikálisan széjjelebb húzni, hogy közben lokálisan ne váljon lineárissá. Végül a fenti képletben alkalmazott paraméterekkel történő transzformációt hajtottam rajta végre. A szigma függvénynek köszönhetően a programra sokkal kevésbé lett jellemző a túltanulás. Emellett sokkal kreatívabban játék élményt tapasztaltam vele, és a hibásan betanult lépéseket viszonylag gyorsan újra súlyozta.

B) Helyzetek generálása

Ezzel a problémával kapcsolatban 2 megoldást láttam: az egyik, hogy az emberi játékos lépésit stratégia nélkül, véletlenszerűen generálja a program magának, a másik pedig, hogy az emberi játkos automatizmusa a számítógép tudás szettjét használja, így a lépései mögött bizonyos szintű „tudatosság" feltételezhető.

Az első módszer előnye, hogy megfelelő mennyiségű játék mellett, akár az összes lépéssorozat reprodukálható. Hátránya azonban, hogy a lépések mögött nincs semmilyen „céltudatosság", minden lépést ugyanakkora valószínűséggel választ, így bizonyos helyzetekben olyan lépést is hasznosnak ítélhet meg, ami egy értelemmel rendelkező játékos szemben a vereségét okozná.

A második módszer elméletben pont ennek a tanulási hibának a kiszűrését szolgálná, azonban itt felmerülhet, hogy mivel mindkét játékos ugyanazzal a súlyozással dönt az esetekben, a tanulási folyamat során sokkal kevesebb különböző játék lehetőséggel találkozik, illetve a félreértet helyzetek öngerjesztő módon berögződhetnek.

A két módszer tesztelése után úgy találtam, hogy a véletlenszerű helyzet generálás jobb tudás szettet eredményez. Így ennél a módszernél maradtam.

C) Ideális játékszám

Az ideális játékszám keresésekor arra figyeltem, hogy a gépet ne jellemezze a túltanulás, tehát kreatívan játsszon és a hibásan megtanult helyzeteket gyorsan és jól korrigálni tudja.

A szigma függvényt használó tanulási algoritmus és a véletlenszerűen generált helyzetek esetén úgy tapasztaltam, hogy körülbelül 5000-10000 játék után megközelítőleg verhetetlen az alkalmazás, veresége esetén körülbelül 10-20 irányított játék után érezhetően megtanulja kezelni a hibásan elsajátított helyzetet.

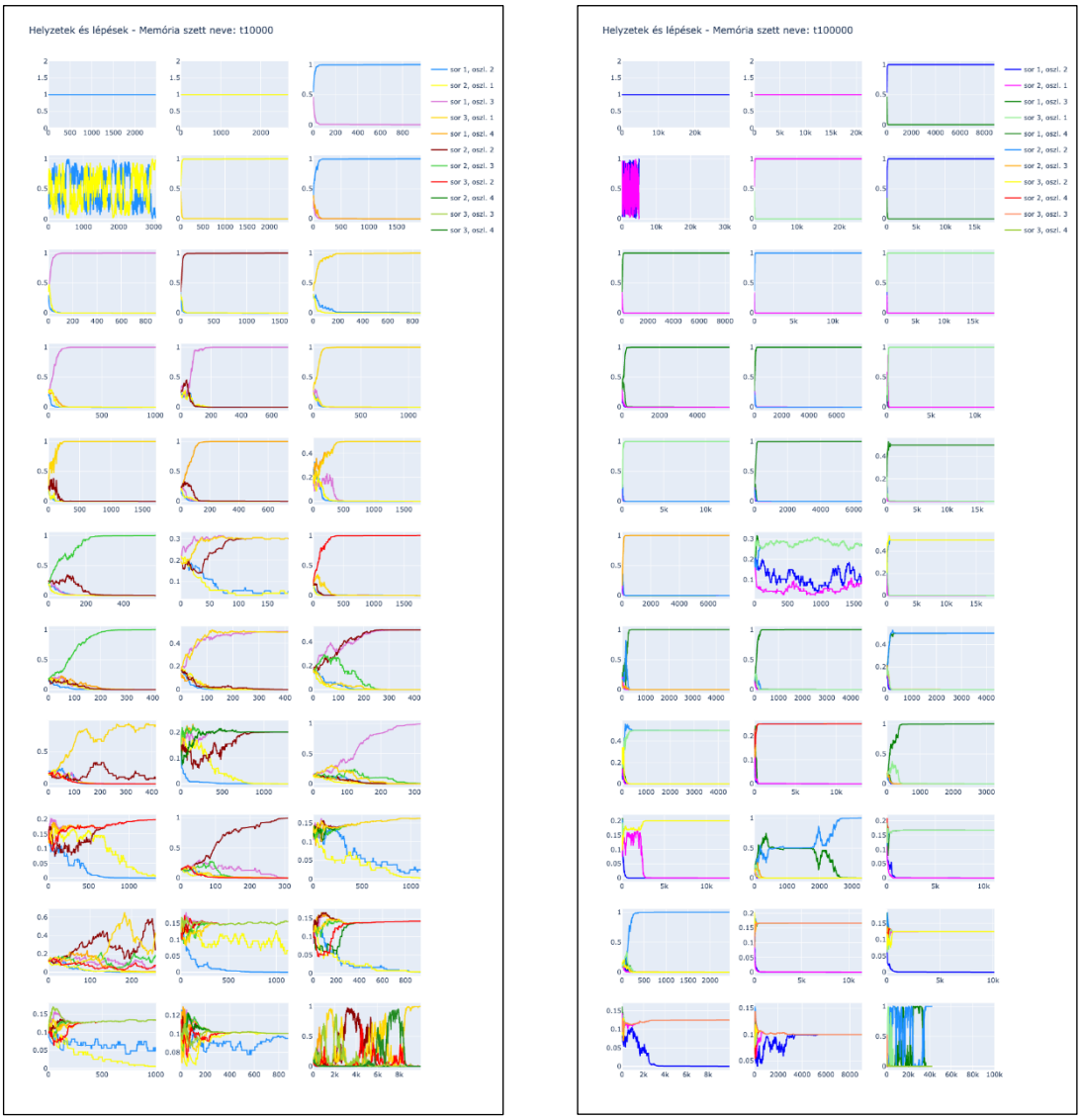
Az optimális lépések – tanulási görbék vizsgálata

A „python chomp_game.py -m <<tudás szett neve>> -r" command line utasításra az alkalmazás legenerálja az adott tudásszetthez tartozó helyzetek és lépések tanulási görbéjét. Ezek a görbék helyzetenként bemutatják, hogy az egyes lépéseknek a bekövetkezési valószínűsége hogyan változott a tanulás előre haladtával (x-tengely jelentése: hányszor találkozott a tanulás során az adott esettel a program).

Feltételezve, hogy a tudásszettek a tapasztalat növekedésével az optimális megoldások irányába tartó valószínűségekkel rendelkeznek, a következő kísérletet végeztem: generáltam egy 10.000 és egy 100.000 játékszámú tapasztalattal rendelkező memória szettet; ezután pedig megvizsgáltam a tudásgörbék jellemző képét a két különböző memória szett esetén.

A vizsgálatok után az alábbi következtetésekre jutottam (az adott helyzetekre itt grafikon képük alapján hivatkozok):

- A játékban 1 darab komplementer helyzet van (ez konkrétan a mérgezett csoki 2 szélső szomszédja), ahol a játék kimenetelén már nem változtat, hogy melyik lépést válassza a program. Látható, hogy itt rendre megcserélik egymást a görbék. Tehát ebben az esetben nincs optimális megoldás.
- Azokban a helyzetekben egyértelműen van optimális lépés, méghozzá egyetlenegy, ahol az egyik lépés valószínűsége 1-hez tart, míg a többié 0-hoz.
- Több optimális megoldás is lehetséges ott, ahol a lépések egyrésze 0-hoz tart, míg másrésze egy ugyanazon nem nulla értékhez.
- Nem feltétlen van optimális megoldás, lehet, hogy mindegyik lépés rossz, ahol az egyes lehetőségek valószínűsége eleinte szétválik, de a végén ugyanoda tartanak.
- A jobb alsó sarok helyzete speciális, mivel „egymást váltják az optimális megoldások”, én ez alapján arra jutottam, hogy a legelején nincs optimális megoldás, de vannak kifejezetten rossz megoldások.



Tanulási görbék