

Ocvist

dokumentáció főbb működési elvekről

BÁNOVICS-ANGYAL GÁBOR



Program célja

Ez a program tisztán random forest tanítóalgoritmust valósít meg. Az amőba játék értékeléséhez nincs heurisztikus függvény írva és a program nem számolja előre a lépésvariációkat, kizárolag a jelenlegi állást értékeli az addig véletlenszerűen játszott partijaiból levont következtetések és mintázatfelismerés (lásd később) által.

Megoldandó problémák

1. Véletlenszerűen szétszórja a jeleket

Amőba játékaim tapasztalata alapján a jeleket egy csoportba érdemes rakni, mert akár egy jel túl messze tétele is akkora előnyt jelenthet a másiknak, hogy győzelemre vezeti. Ezért a program sem az egész táblán rak véletlenszerűen, hanem csak a már kialakult alakzat bizonyos sugarán belülre.

2. Túl sok olyan eset, amikor a játék már értelmét vesztette

Tekintsünk egy egyszerű példát: körnek összejött 4 jel egymás mellett, vagyis egy lépéstre került a győzelemtől. Ebben az esetben, ha a program véletlenszerűen lép, akkor elég kicsi az esélye, hogy pont jó helyre fog lépni, ami nagyon elnyújtja ennek az állásnak a játékát, holott már értelmét vesztette, mert a győzelem feltételezhetően sokszorosan el lett kerülve. Így egy-egy állásban túl sok lehetőség van, ami miatt emberi időn belül nem lehet elég szimulációt futtatni, hogy értelmes lépéseket tanulhasson meg a gép, ezzel a módszerrel. Ennek érdekében az egyértelmű állásokra mintázatfelismerés lett beépítve, hogy ezzel is csökkentve legyen az eseménytér és a program lépéseinél az elfogadható irányba tereljém.

3. Egyes pozíciók elmentése

Az egyes pozíciókat egy stringként mentem el. Ehhez az egyes cellákat bizonyos irányba haladva megszámozom (lásd később), majd a stringbe sorba feljegyzem, hogy a körök és x-ek melyik számú mezőn állnak.

4. Egyforma pozíciók kezelése

Tegyük fel, hogy a program eljutott már az alábbi helyzetek:

×	×	O				
	×	O				
		O	O			
			X			

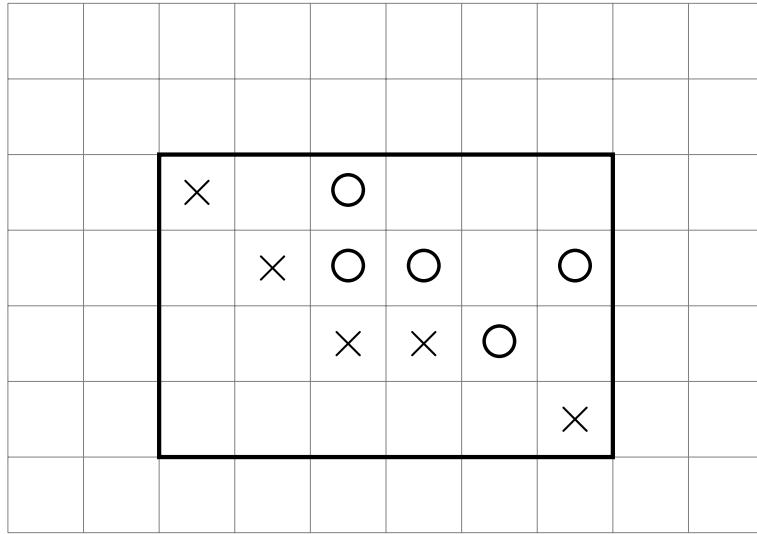
		O	X	X		
		O	X			
		O	O			
			X			

Könnyen észrevehetjük, hogy a két állás megegyezik, csak egymás tükröképeik. A programnak fel kell ismernie, hogy ezek egyformák és úgy is kell kezelje őket, különben egy álláshoz több különböző redundáns adathalmaz fog keletkezni.

Lehetséges transzformációk:

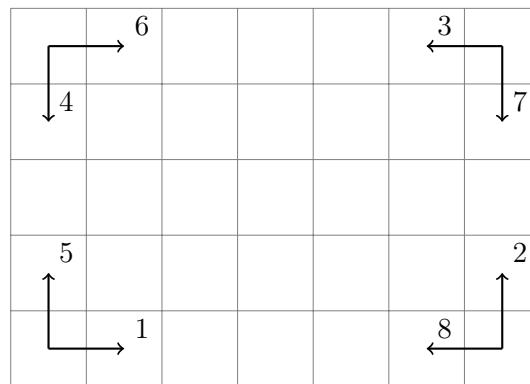
- Eltolás megoldása

A kialakult alakzat köré egy legkisebb befoglaló téglalapot (későbbiekben: köré írható téglalap) írunk és csak ezen belül figyeljük a jelek elhelyezkedését, nem pedig az egész táblán. Így kiküszöböltük, hogy pl. ugyanaz az állás egy sorral odébb más adatként szerepeljen.



1. ábra. Példa egy (képzeletbeli) téglalap meghatározására.

- Forgatás, tükrözés megoldása



Példa: 2-es nyíl iránya:

35	30	25	20	15	10	5
34	29	24	19	14	9	4
33	28	23	18	13	8	3
32	27	22	17	12	7	2
31	26	21	16	11	6	1

Például az 1. ábrán látható állásból az alábbi pozícióleírások adódnak:

pos1: 0500x0201x0301x0401o0102x0202o0302o0502o0003x0203o
pos2: 0000x0200o0101o0102x0202o0103x0203o0303o0204x0305x
pos3: 0300o0500x0001o0201o0301o0401x0102o0202x0302x0003x
pos4: 0000x0101x0002o0102o0202x0103o0203x0204o0105o0305x
pos5: 0300x0201x0102x0202o0302o0103x0203o0104o0005x0205o
pos6: 0000x0101o0201x0301x0002o0202o0302o0402x0303o0503x
pos7: 0100o0300x0201o0102o0202x0003o0103o0203x0104x0005x
pos8: 0000x0200o0101x0201o0301o0501o0202x0302x0402o0503x

5. Szimmetrikus állások kezelése

Az alábbi ábrán egy szimmetrikus állás látható:

Vagyis ebben az esetben mindegy, hogy az alakzat melyik felére kerül a következő jel. Viszont a programnak nem szabad külön esetként tekintenie ezekre, mert akkor szintén redundáns adatok keletkeznek.

Megoldás: Az előbb leírt 8 stringet kell vizsgálni: ha vannak közöttük olyanok, amik ugyanazt a stringet eredményezték, akkor az állásnak van valamilyen szimmetriája, amit már ezután lehet kezelni.

Mintázatfelismerés

Mintázatfelismerésre – ami valójában azt jelenti, hogy előre megmondom a gépnek, hogy bizonyos esetekben mi a jó lépés – azért van szükség, mert enélkül akár az egylépéses nyerő állások is nagyon sokáig nyújtanák az amúgy is hosszú tanítást, mire rájönne az ilyen esetekben a triviális lépésre. Enélküл feltételezhetően semmi értékes nem történne hosszú tanítás után sem. Vagyis ezzel az egyértelműen értelmetlen lépéseket szűrjük ki már játék közben, ami még így is a legtöbb esetben több lehetséges lépésre szűkít csak.

A mintázatfelismerésnek hierarchikus sorrendje van, ami azt jelenti, hogy először az egylépéses, majd kétlépéses, és így tovább kombinációkat vizsgálja meg, és amelyik szintnek megfelelő mintázatot ismeri fel, arra fog csak reagálni. Fontos megjegyzés, hogy a program először a saját esetét majd az ellenfélét figyeli (vagyis először támad, utána védekezik), majd lép a következő szintre.

Például: először megvizsgálja, hogy van-e egylépéses nyerése, ha nincs, akkor megnézi, hogy van-e az ellenfelének, ha nincs, akkor megnézi, hogy tud-e kialakítani védhetetlen nyerésre vezető állást, majd hogy az ellenfele tud-e és így tovább.

Mintázatfelismerés lépései (zárójelben a programban megírt függvények nevei), a táblázatokban ahol "." van ott a cella értéke 0 vagyis a program oda nem sorsol lépést, csak oda ahol 1-esek vannak:

1. Triviális eset (*trivial*)

Ezen a szinten a program felismeri az egylépéses nyeréseket (1-est ír a megfelelő cellákba).

Példák:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	
2	2	
3	3	
4	4	
5	5	
6	6	
7	7	
8	1	o	o	o	o	x	8	
9	9	
10	10	
11	11	
12	12	
13	13	
14	14	
15	15	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	
2	2	
3	3	
4	4	
5	5	
6	6	
7	7	
8	o	o	1	o	o	x	8	
9	9	
10	10	
11	11	
12	12	
13	13	
14	14	
15	15	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	

2. Védhetetlen egylépéses nyerésre vezető állás (*trivial2nd*)

Olyan helyzet kialakítását keresi, ahol legalább két olyan lépés is van, amelyik egy lépésben nyer.

Példák:

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

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

3. Következő szinten a cél két különböző irányból egymásba érő a nyeréstől két két lépéstre lévő alakzat vagy egy két lépére lévő és egy egy lépére lévő, de védhető alakzat kialakítása egy lépéssel

Felhasznált függvények:

- *open3*: nyílt hármas alakzatnak a koordinátáival tér vissza.

Példa:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	
2	2	
3	3	
4	4	
5	5	
6	.	.	.	x	.	o	o	o	.	.	x	.	.	.	6	
7	7	
8	8	
9	9	
10	10	
11	11	
12	12	
13	13	
14	14	
15	15	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	

Visszatérés: `[(7, 5), (5, 5), (6, 5)]`

- *open3_incompl*: olyan hármas alakzatnak a koordinátáival tér vissza, mely között van egy üres hely.

Példa:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	
2	2	
3	3	
4	4	
5	5	
6	.	.	.	x	.	o	o	o	.	x	6	
7	7	
8	8	
9	9	
10	10	
11	11	
12	12	
13	13	
14	14	
15	15	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	

Visszatérés: `[(8, 5), (5, 5), (6, 5)]`

- *half_closed4*: egyik végén zárt négyesnek a koordinátáival tér vissza.

Példa:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	
2	2	
3	3	
4	4	
5	5	
6	.	.	.	x	o	o	o	o	6	
7	7	
8	8	
9	9	
10	10	
11	11	
12	12	
13	13	
14	14	
15	15	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	

Visszatérés: `[{(4, 5), (7, 5), (5, 5), (6, 5)}]`

- *closed4_incompl*: minden két végén zárt olyan négyes alakzatnak a koordinátáival tér vissza, mely között van egy üres hely.

Példa:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	
2	2	
3	3	
4	4	
5	5	
6	.	.	.	x	o	o	o	.	o	x	6	
7	7	
8	8	
9	9	
10	10	
11	11	
12	12	
13	13	
14	14	
15	15	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	

Visszatérés: `[{(4, 5), (8, 5), (5, 5), (6, 5)}]`

- Speciális eset: *double4_inline*: megnézi, hogy ki tud-e alakítani egy olyan alakzatot, ami minden két irányából nézve egy-egy hiányos négyesnek fogható fel. Vagyis ilyen alakzatra való kipötlés lehetőségét keresi:

x o . o o o . o x

Példa:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	
2	2	
3	3	
4	4	
5	5	
6	.	.	.	x	o	.	.	o	o	.	o	x	.	.	6	
7	7	
8	8	
9	9	
10	10	
11	11	
12	12	
13	13	
14	14	
15	15	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	

Visszatérés: {(6, 5)}

Fő függvény, amely az előbbieket használja: **force** illetve **force_new** (*force* javított változata, de még nem töröltem az előzőt, mert fejlesztés alatt áll a program). Ebben a függvényben a program azokra a helyekre, ahol 1-esek szerepelnek virtuálisan beír egy jelet és megnézi, hogy a fent ismertetett alakzatkereső függvények által visszaadott listáknak van-e metszete, ha igen, akkor az azt jelenti, hogy arra a helyre téve egy jelet az adott játékosnak védhetetlen nyerése lehet. Egyelőre még csak feltételezés, mert előfordulhat, hogy az ellenfél tud annyira agresszív lenni, hogy még mindig ő jön ki győztesen (ezt valósítja meg a *mini-force* függvény)

Megjegyzés: bizonyos függvényeknél nem mindegy, hogy az keresi az adott alakzatot, aki támad vagyis saját jelét vizsgálja vagy az ellenfelét, mert egy-egy ilyen alakzat ellen többféleképpen lehet védekezni, mint támadásra kihasználni.

Játék és tanulás

A program maga ellen játszik minden egyes lépésnél kitölti a táblát 1-esekkel, a kialakult alakzat körül, majd megnézi, hogy a mintázatfelismerés révén tudja-e szűkíteni ezeket, majd a megmaradt lehetséges helyek közül véletlenszerűen választ egyet. Így megy a játék végéig.

Ezután a lejátszott partit elmenti. A legutolsó lépést egyértelműen tiltólistára helyezi, a közbülső lépéseket a nyerő félnek +1-gyes súlytal, a vesztes félnek -1-gyes súlytal értékeli.

Sok játék lejátszása után középjátékba tartozó lépéseket is tiltólistára helyezhet, ha annak az addigi lejátszott partik alapján nagyon rossz statisztikája van (*conclusion* függvény). Illetve azokat az állásokat, amik csak egy partiban fordultak elő törli (*cleardat* függvény), hogy memóriát szabadítson fel.

A program *multiprocessing*-et is használ, de az egyes ágak által generált tanulságok összeolvasztásában még hiba van.