Amikor a nagyházid elver amőbában

Móna Dániel

Amikor majdnem egy évvel ezelőtt el kellett dönteni, hogy milyen Prog1 házit szeretnék csinálni, nagyjából egyből elhatároztam, hogy azért egy telefonkönyvnél valami jobbat szeretnék írni. Voltak is a listán például játékok, amik közül az amőba úgy szimpatikusnak tűnt. A specifikáció beadása előtt kicsit hezitáltam, mert az InfoC-s amőbajáték kiírásban ott volt a gépi ellenfél, és ha már ilyesmit elvállalok, szerettem volna *rendesen* megcsinálni. Végül csak nekivágtam, mert YouTube-videókból, meg innen-onnan volt sejtésem, hogy hogyan is lehetne nekifogni ennek.

Ezen „csináljuk meg akkor már rendesen” mentalitás persze a program minden részére átcsapott, és emiatt kissé túl is lőttem a kötelező követelményeket. Sikerült a Prog1 tananyag teljes egészét felhasználni, a `void*`-tól az állapotgépen át a függvénypointerig. Természetesen pótleadás lett belőle.

Amit példának kiemelnék, az az a kis „gomb library”, amit a menü gombjaihoz összehoztam. A gomboknak volt:

- Callback függvénye
- Predikátumfüggvénye láthatóságra
- Predikátumfüggvénye lenyomott állapotra

3-féle gomb volt: sima, kapcsoló (ilyenek vannak a Beállítások menüben), illetve átlátszó. Ez utóbbit ráadásul csak a játék pause menüjének gombjához használtam. Ezenfelül lehetett még őket gombcsoportokba rendezni, illetve gombokat és gombcsoportokat gomblistákba pakolni. Mindezek után a `gomb_uj` függvény így nézett ki:

```
Gomb *gomb_uj(GombTipus típus, char *szoveg, void (*fuggveny)(int),
     int fv_param, bool (*lathato_pred)(int,int), int lth_p1, int lth_p2,
     bool (*lenyomva_pred)(int,int), int lny_p1, int lny_p2);
```

Mondhatom, lenyűgöző.

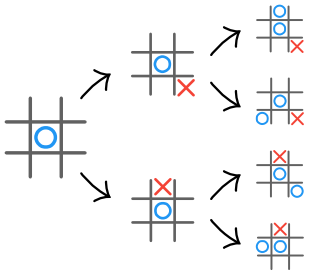
A gomboknál viszont sokkal izgalmasabb a kérdés: hogyan tud ez amőbázni? Mostanság rengeteget hallani mindenféle tanuló algoritmusról, meg neurális hálóról, és megijedhetünk, hogy akkor most ezt kellene megérteni, hogy gép ellenfelet tudjunk írni? Bár valószínűleg ilyen megoldásokkal is lehetne olyan MI-ket csinálni,

amik nemhogy elvernek, de *totálisan megsemmisítenek* amőbában, de szerencsére, amint az számomra is kiderült, az ilyen szintű problémához nem kell ekkora nagyágyúkat használnunk. De hogy fussunk akkor neki?

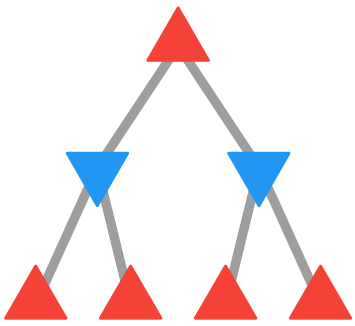
Ami az embernek gólyafejjel elsőre eszébe juthat arról, hogy hogyan írhatna amőbázó gépet az az, hogy pakoljunk bele jó sok `if else`-t: `if` ez, csináld ezt; `else if` az, csináld azt; `else if` amaz, csináld amazt; és így tovább. Ezt a gondolatot aztán kb. 3 másodperc megfontolás után mindenki kihajítja az ablakon, ugyanis rögtön látszik, hogy:

- kilométeres, átláthatatlan spagettikódot eredményezne
- *minden* helyzetre külön készülni kellene, ami lehetetlen
- közel sem lenne kellemes bővíteni rajta valamit

Felmerülhet viszont az az ötlet is, hogy jól járhatnánk azzal, ha előre kiszámítjuk a lehetséges játékállásokat. Ezt az amőbához hasonló körökre osztott játékoknál megoldható, és ha megcsináljuk, már csak valahogyan ki kell választani az éppen legjobbnak tűnő állást.



Erre pedig pontosan alkalmas az úgynevezett min-max (avagy minimax) algoritmus. A min-max abból indul ki, hogy van egy játékállásokból épített fánk, amiben az olyan játékállásokhoz, amiknek már nincsenek alesetei a fában, valamilyen pontszám van rendelve. A fa egyes szintjein felváltva következik a két játékos, melyek közül az egyik minél magasabb, a másik minél alacsonyabb pontú állások felé próbálja vinni a játékot. Az olyan csomópont értéke a fában, ahol a maximalizáló játékos jön, az adott csomópont gyerekei közül a legnagyobb pontszámú, a minimalizáló játékos esetén pedig nyilván a legkisebb.



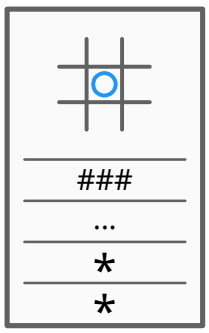
A linkelt írásban látszik, hogy ha lehetőségünk van az állásokat addig előreszámítani, hogy mindig eljussunk az egyik játékos győzelméhez, akkor semmi plusz okosság nem szükséges, a min-max megadja a tökéletes lépést. Ugyanis ebben az esetben elég úgy pontozni, hogy +∞ (vagy csak egy *jó nagy* szám), ha a maximalizáló játékos nyer, és -∞ (vagy csak ugyanaz a *jó nagy* szám, csak most mínuszban), ha a minimalizáló.

Ez szép és jó a cikkbeli 3×3-as „tic-tac-toe” amőbához, azonban az általam írt n×m-es amőbánál és még sok más játéknál egyszerűen túl sok a lehetőség, túl sok számítást igényelne mindig eddig előre gondolkodni, azaz valahány kör után félbe kell hagynunk a játékállások számítását. Ennek megfelelően a fa levelei nagyrészt olyan játékállások lesznek, amiknél még nem nyert senki – azaz kell valami okos pontozás, ami egy be nem fejezett játékállásról is el tudja mondani, hogy mennyire kedvező az egyes játékosoknak.

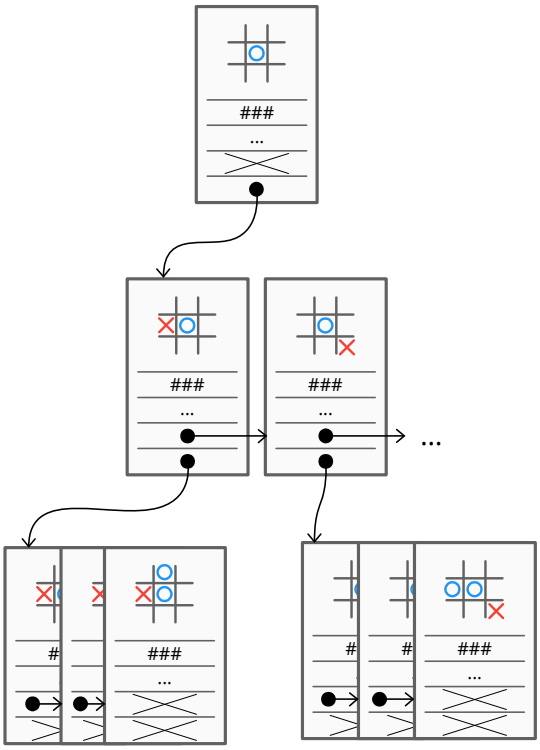
Már valószínűleg ebből a sok fecsegésből is kikövetkeztethetted, de le is írom (még több fecsegéssel), hogy a min-max *rekurzív*: meghívjuk a fa gyökerére, és odaadjuk neki, hogy itt mondjuk a maximalizáló játékos jön. Akkor most a maximalizáló játékos gondolkodásával meg akarja keresni, hogy a fa gyökerét jelentő állás alesetei közül melyik a maximális pontszámú – de ezeknek még nincs pontszáma! Nosza rajta, meghívja önmagát, most a minimalizáló játékos szerepében egyenként az alesetekre… Ez így megy a szerepeket váltogatva végig le a fán, amíg el nem érünk egy levélhez, egy aleset nélküli játékálláshoz: ekkor odaadja ezt a játékállást a pontozófüggvénynek, ami visszaadja a pontszámot, amit utána vissza lehet adni a hívónak, aki vagy a maximalizáló, vagy a minimalizáló játékos szerepében most már válogathat az immár pontozott állások közt, visszaadhatja a legnagyobb/legkisebb pontszámút… És így tovább fel a fán, míg végül a fa gyökerén meghívott min-max kiválaszthatja a most már pontot kapott alesetei közül a legmegfelelőbbnek tűnőt.

Hogyan van mindez megvalósítva az én házimban? Nézzük először a játékállásokat. Egy játékállás-struktúrában el van tárolva:

- a pálya
- a legutóbbi lépés helye
- melyik játékos van éppen soron
- a játékálláshoz rendelt pontszám



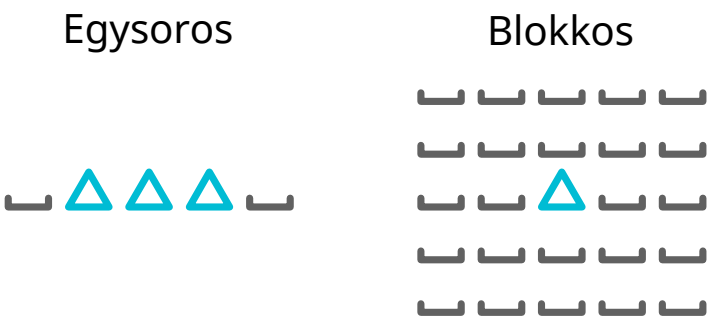
A struktúra tartalmaz még két pointert: az egyik mutat egy, az alesetekből álló láncolt lista elejére, a másik pointer pont ezekben a láncolt listákban mutatja a következő elemet. A lehetséges játékállások felderítése során felépülő adatstruktúra így lényegében egy jó nagy fésűs lista, de kicsit jobban felfogható, mit szeretne elérni, ha így rendezzük ábrára az elemeit:



Erre ezután már meghívhatjuk a min-max-ot, ami jelen esetben úgy van megírva, hogy a legjobb alesetet tartalmazó játékállásra mutató pointert adjon vissza (levél esetén pontoz, majd visszaad egy pointert önmagára a pontozott levélre). Ez a fa gyökerén meghívott min-max esetén lehetővé teszi, hogy rögtön visszaadhassuk a gép lépésének helyét: csak ki kell olvasnunk a visszakapott játékállásból.

Végül már csak azt kell kitalálni, hogy milyen okossággal pontozzunk. Ebben sikerült arra rájönnöm, hogy az amőba igazából *mintákról* szól. Amikor amőbázunk, nekünk kedvező mintákat akarunk összehozni a pályán, miközben ebben az ellenfelet gátoljuk.

Kellenek tehát először is mintákat tároló struktúrák. Kétfélét csináltam: egysoros és blokkos mintát. (Ebből a pontozásra végül csak az egysoros lett használva, de nyugodtan hozzá lehetne adni valami blokkos mintát is, a kód támogatja.)



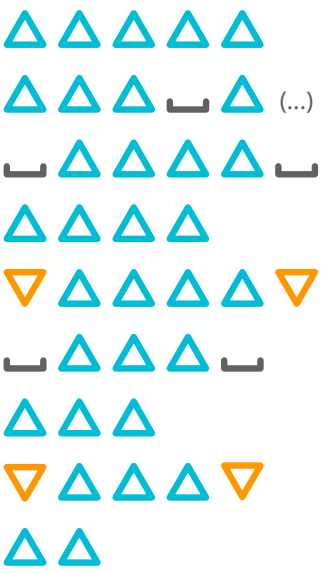
A mintákban a következő mintajelek lehetségesek:

A kör és az iksz lényegében csak azért van ott, hogy ugyanazon az `enum` értéken ugyanaz a jel legyen, mint a pályacella `enum`-ban, ahol csak üres, kör és iksz van, mintákban nincs sok értelme használni őket.

Ami már sokkal hasznosabb, az a változó mintajel. Ugyanis így nem kell ugyanazt a mintát kétszer tárolni, és lehet úgy megírni a mintakereső függvényt, hogy odaadjuk neki a változó jeles mintát, illetve hogy mit helyettesítsen a változóba, kört vagy ikszet (lehetne mást is, de megint csak nem sok értelme van). A negált változóba pedig ennek megfelelően a behelyettesített mintajel ellentettje kerül behelyettesítésre.

Az érdektelen mintajel ott jöhet jól, ha egy pozíció tartalma a mintában, nos – érdektelen, azaz teljesen mindegy, hogy ott mi van, mindig illeszkedni fog rá. Meg lehet jegyezni, hogy az üres nem ugyanaz, mint az érdektelen: ha a mintában valahol üres van, akkor ott *üres helynek kell lenni*, a minta csak úgy illeszkedik.

A pontozás a következő mintákat használja (abban az esetben, ha 5 jel kell sorban a győzelemhez) →



A pontozófüggvény végighalad ezen a mintalistán, és mindegyiket először az egyik, majd a másik jelből többféle módon (balról jobbra, jobbra lefelé átlósan, lefelé, balra lefelé átlósan – máshogy már nem kell) megpróbálja ráilleszteni a pályára minden pontból. Minden mintához hozzá van rendelve egy pontszám, ha a gép jeléből találja meg, akkor hozzáadja ezt a pontszámot a játékállás pontjaihoz, ha az ellenfél jeléből, akkor levonja, illetve ezen felül levon 1 pontot.

Ez azért jó, mert így a gép még 1 mély játékállás-fa esetén is jól játszik, mindig „észreveszi”, ha védekeznie kell.

Így tehát a pontozás is megvan, készen vagyunk. Esetleg még az lehet a gond, hogy túl lassan fut az egész. A következő ötletekkel némileg növelni lehet a lépésszámítás sebességét:

- Nyilván jó ötlet, ha optimalizáltatjuk a programot a fordítóval. Már ezzel rengeteget gyorsul.
- Minél jobban pontozunk, annál kevesebb körrel muszáj előre gondolkodni. Ha csak a nyertes és az ellenfél nyertes mintájára adnánk pontot, végig kellene ásni az egészet. A használt mennyiségű mintával és a hozzájuk rendelt pontértékek többé-kevésbé finomhangolásával viszont sikerült olyan jó pontozást összehoznom, hogy lényegi min-max nélkül is viszonylag jól játszik a gép: könnyű nehézségen a fa 1 mély, azaz a gyökér aleseteinek rögtön nincsenek alesetei, közvetlenül pontozva vannak. Persze valószínűleg ezt is túl lehet lőni, azaz netán előfordulhat, hogy annyi számításba kerül a pontozás, hogy jobban megérné egy egyszerűbb pontozással mélyebbre menni.
- Ne építsünk mindig új játékállás-fát. Ezt úgy lehet összehozni, hogy minden lépés után a keletkezett játékállást megpróbáljuk megtalálni a meglévő fában. Ha megvan, az lesz az új gyökér, a többit eldobjuk. Ezután a kapott fát továbbszámoljuk egy körrel és megvan az új fánk. Ha nincs meg, nyilván új fát építünk, de ez ritkábban fordul elő, esetleg soha, a játékállásokat felderítő függvénytől függően.
- Alfa-béta vágás. Ez egy afféle felokosított min-max. A lényege az, hogy hogyha tudjuk, hogy egy bizonyos játékállásba a fán az ellenfél nem fogja hagyni, hogy eljussunk, mert van egy jobb opciója, akkor azzal a játékállással és az aleseteivel sem nem kell foglalkoznunk. Így a játékállás-fa egy viszonylag jelentős részét nem kell kiértékelnünk, amivel gyorsabb futást kapunk, és/vagy lehetőségünk van mélyebb játékállás-fát használni.

Amit végezetül ismét hangsúlyoznék, hogy ez nem csak amőbához jó. A min-max működik bármilyen körökre osztott játékra, ahol egy játékálláshoz pontszám rendelhető. Összesítve megint:

1. Felderítjük a játékállásokat valahány körrel előre
2. Pontozzuk azokat játékállásokat, amiknek már nem számítottunk alesetet
3. Min-max

Már ez az egyszerű algoritmus, plusz egy megfelelő pontozás is (számomra legalábbis) meglepően „emberi" játékot eredményez. A gép tényleg jókat lép, és a min-max működéséből fakadóan látszólag „azon gondolkozik", mit lépnél te, hiszen a minimalizáló játékos körében lényegében a te helyzetedből keresi a jó lépést.

Azt kell mondjam, hogy 2-4 órával legalább csúszott a házim elkészülése amiatt, mert egy kis részlet tesztelése átcsapott abba, hogy elkezdtem vele játszani... és amikor elsőre sikerült levadászni minden ehhez kapcsolódó bugot, és a gép mintegy varázsütésre elkezdett értelmeseket lépni – az egy remek érzés volt. Meg amikor valamivel később először sikerült nyernem a saját nagyházim ellen amőbában. Mert előtte kikaptam. Elég sokszor.

Nos, remélem, hogy sikerült néhány ötletet és némi motivációt átadnom. Sok sikert kívánok a nagyházidhoz / egyéb projektedhez, bízom benne, hogy sikerül neked is valami tényleg szórakoztatót összehoznod!