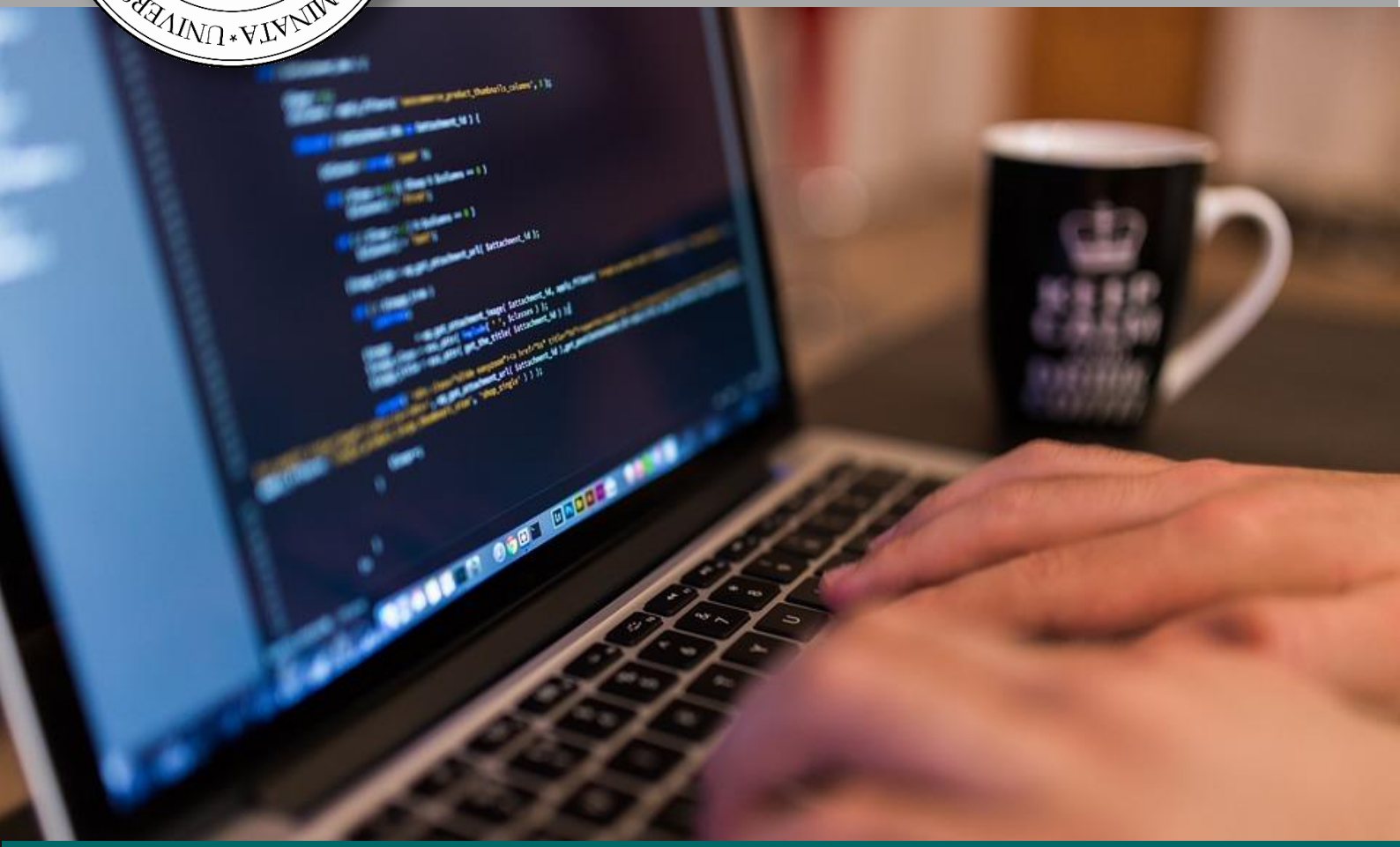
Belépő a tudás közösségébe

Informatika szakköri segédanyag



Algoritmizálás játékokban

Bende Imre

Szerkesztő: Zsakó László

A kiadvány „A felsőoktatásba bekerülést elősegítő készségfejlesztő és kommunikációs programok megvalósítása, valamint az MTMI szakok népszerűsítése a felsőoktatásban” (EFOP-3.4.4-16-2017-006) című pályázat keretében készült 2018-ban.

Eötvös Loránd Tudományegyetem
Informatikai Kar

SZÉCHENYI 2020
MAGYARORSZÁG KORMÁNYA
Európai Unió
Európai Szociális Alap
BEFEKTETÉS A JÖVŐBE

Ebben a szakköri anyagban számítógépen elérhető játékok algoritmussal való megoldását szeretném bemutatni. Minden feladathoz tartozik egy weboldal, ami a játék felületét adja, illetve annak szabályai, lépései vannak implementálva benne. A megoldáshoz a diákoknak különböző algoritmusokat, programokat kell írniuk, mely különböző témaköröket dolgoz fel a programozás oktatás egyes részeinek, majd helyezi el azokat egy reálisabb, könnyebben feldolgozható formában.

Bevezetés – ismerkedés a környezettel

A későbbiekben megjelenő játékok/feladatok valószínűleg már mindenki számára ismerősek, ezeket azonban többnyire lépésenként, fejbe oldották meg, játszották le a diákok. A megoldások létrehozásához mindenképpen szükség lesz arra, hogy ne csak egy adott állapotot lássunk át, hanem az egész folyamatot (a játék kezdetétől a végéig), majd az ezek után felismert sajátosságokat kell algoritmusba helyezni, amit végül pedig implementálni kell egy adott programozási nyelven (jelen esetben JavaScriptben). Érdemes felhasználni, megemlíteni az „oszd meg és uralkodj" elvet is, tehát kisebb részekre (lépésekre) szétbontani a folyamatot, majd ezáltal megoldani az egész problémát, játékot.

A feladatokhoz tartozó mellékletek egy-egy *index.html*-t tartalmaznak, ami az oldal felépítését tartalmazza, valamint egy *jatek.js* fájlt, ami pedig a játék szabályait írja le (igyekeztem minden szabályt, kivételt minél pontosabban implementálni benne). A diák feladata egy *jatekos.js* fájl létrehozása, amely két függvényt tartalmaz (tehát mindegyik esetben egy JavaScript fájlban kell a megoldást megvalósítani):

- *kezdes()*: ez tartalmazza azokat az adatokat, amikre már az elején szükségünk lehet. Ebben lehet inicializálni a megoldáshoz szükséges változókat, adatstruktúrákat.
- *lepes()*: itt megkapja a megoldó az egy adott állapothoz tartozó információkat, ez alapján kell meghatároznia, hogy mi lesz a következő lépése. Ez az egész magja, ez a függvény fog minden egyes gondolatmenetet tartalmazni. Addig hívodik ez a függvény, míg a játék véget nem ért (fontos, hogy egyes esetekben emiatt, akár végtelen ciklusba is kerülhetünk!).

Természetesen egyéb segédfüggvényeket is lehet írni, a feladat/weboldal csak ezt a kettőt fogja meghívni, de ha beágyazzuk azokba, akkor a hívásuk ugyancsak meg fog történni.

A játékok minden esetben tartalmaznak egy leírást, ami bemutatja a feladatot, milyen változók jöhetnek szóba, illetve a *kezdes()*, *lepes()* függvények milyen paramétereket várnak be, illetve miket kell, hogy visszaadjanak. Majd bemutatom a felületet, hogy hogyan is jelenik meg vizuálisan a feladat világa a mellékletek közt található weboldal segítségével. Lényeges pontnak a megoldási stratégiákat tartom, mivel itt részletezem, hogy milyen ötletek érdemes felhasználni a megoldás implementálásakor, ezeket részletezem is. Fontos, hogy minden feladatnál (ahogyan az általában egy programozási problémánál elő szokott fordulni) nem csak egy megoldás lehetséges, lehet ötleteket felhasználni, lehet azokat ötvözni, lehet teljesen új, akár önálló gondolatokat is beépíteni. Nyilván egyesesetekben azért vannak „hatékonyabb" megoldások, de ezeket is érdemes felülvizsgálni. Értem itt ezalatt, hogy gyorsabb az algoritmusunk, de milyen áron (mondjuk több memóriát használ fel, lehet a gép ennyivel nem is feltétlenül rendelkezik)? Egyes esetekben a megoldásokat is bemutatom, részletezve, hogy mit, miért és hogyan próbáltam felhasználni annak létrehozásakor. Végül pár a honlap, felület elkészítésével kapcsolatos információt is leírok, amellyel a célom nem csak az oldal környezetének jobb megismerése, hanem hogy ezáltal az olvasó, diák is egyfajta irányzatot, segítséget kapjon későbbi hasonló felületek készítéséhez.

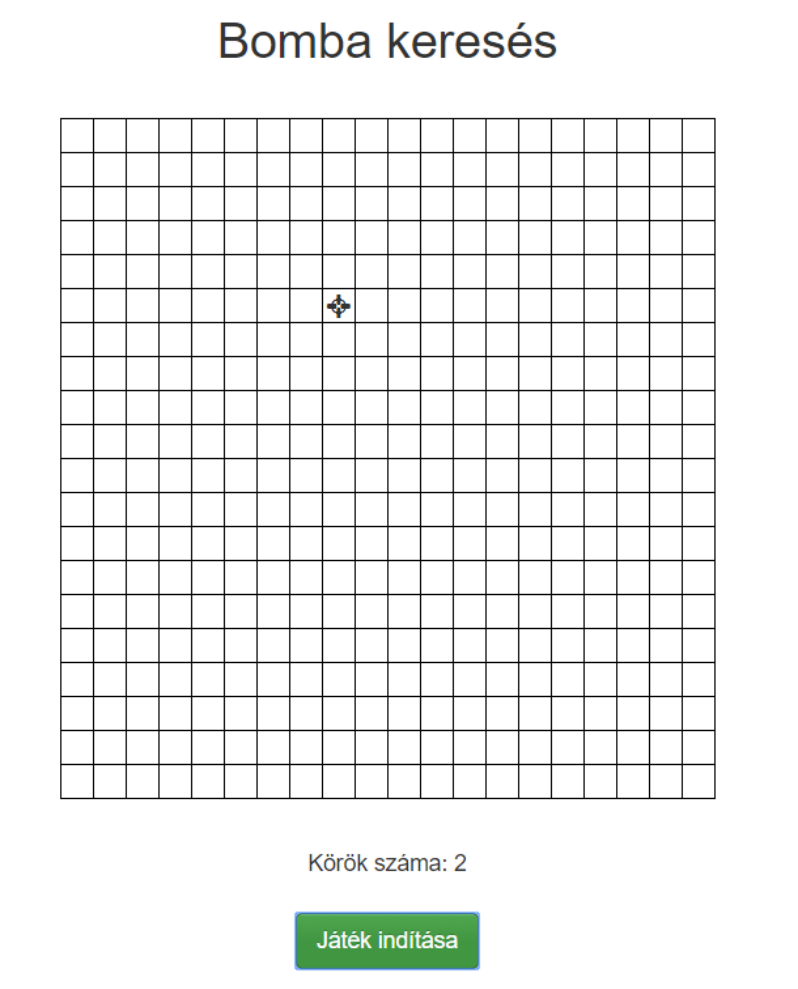
Bombakeresés

A feladat

Egy N emeletes ház minden emeletén M lakás egyvonalban helyezkedik el. A házban valaki elrejtett egy bombát egy aktatáskában, így a feladatunk, hogy minél előbb megtaláljuk azt, mielőtt felrobbanna! Főhősünk képes ablakok közt ugrálni és megvizsgálni, hogy tőle merre található a bomba (F=fel, L=le, J=jobbra, B=balra, illetve ezek mentén kombinációk is lehetnek: FJ, FB, LJ, illetve LB is).

A *kezdes(N, M)*-ben megkapjuk, hogy hány emelet (N), illetve emeletenkénti lakás (M) van. A *lepes(poz, irany)* függvény bemenő paramétere a főhős pozíciója (x, y koordinátával, pl.: {x: 0, y: 0}), valamint szöveges formában, hogy tőle merre helyezkedik el a bomba (pl.: „FJ"), ennek visszatérési értéke pedig a következő lépés pozíciója kell, hogy legyen (pl.: {x: 0, y: 0}).

A felület



1. ábra Bomba kereső játék felülete

Játék indítása gombra indul a játékos *lepes()* függvényének hívása másfél másodpercenként. Célzókereszten látszik az aktuálisan vizsgált hely, míg egy táskával van jelölve a négyzet, ha sikerült megtalálni a bombát. Valamint azt is láthatjuk, hogy hányadik vizsgálatnál tartunk, hányadikra találtuk meg a bombát.

Megoldási stratégiák

Kézenfekvő megoldás lehet első lépésként egy mátrixon való lineáris keresés, azonban ez nagyon lassú adott esetben N*M-szer fut le, ennyi idő alatt a bomba fel is robbanhat, így tehát próbáljuk felhasználni azt, hogy ismerjük tőlünk mely irányba helyezkedik el a bomba. Érdemes a legjobban leszűkíteni a lehetséges helyek körét, egy olyan módszerrel mellyel folyamatosan tudjuk felezni (vagy mivel síkról beszélünk így negyedelni) a tartományt. Ezt a logaritmikus kereséssel fogjuk megvalósítani, valamint annak kétszeri felhasználásával a sík két tengelyén. Logaritmikus keresés algoritmusa pszeudokóddal:

```
U := 1;
V := N;
L := Hamis;
Ciklus amíg L Hamis ÉS U<=V
  i := (U+V)/2;
  HA A[i]=X AKKOR
    L := Igaz;
  HA A[i]<X AKKOR
    U := i+1;
  HA A[i]>X AKKOR
    V := i-1;
Ciklus vége.
```

Megoldás

Tehát a megoldás során mindkét iránynak (fel-le, jobbra-balra) megfelelően kiválasztom a következő vizsgálati helyet felezési módszerrel, majd ennek a pontnak a koordinátáit adom vissza egy modellben:

```
if (irany.includes("F")) {
  y2 = y - 1;
} else if (irany.includes("L")) {
  y1 = y + 1;
}

if (irany.includes("B")) {
  x2 = x - 1;
} else if (irany.includes("J")) {
  x1 = x + 1;
}

x = Math.floor(x1 + (x2 - x1) / 2);
y = Math.floor(y1 + (y2 - y1) / 2);

return {
  x: x,
  y: y
}
```

Síkban tehát az iránynak megfelelően egy téglalappal jelölhetjük a bomba lehetséges helyzetét, amit minden egyes lépéssel negyedelünk (ha egy koordinátája stimmel, akkor már csak egy egyenesen felezgetjük az értéket, míg, ha mindkettő jó, akkor pedig ugyebár már megtaláltuk a bombát).

A honlapról

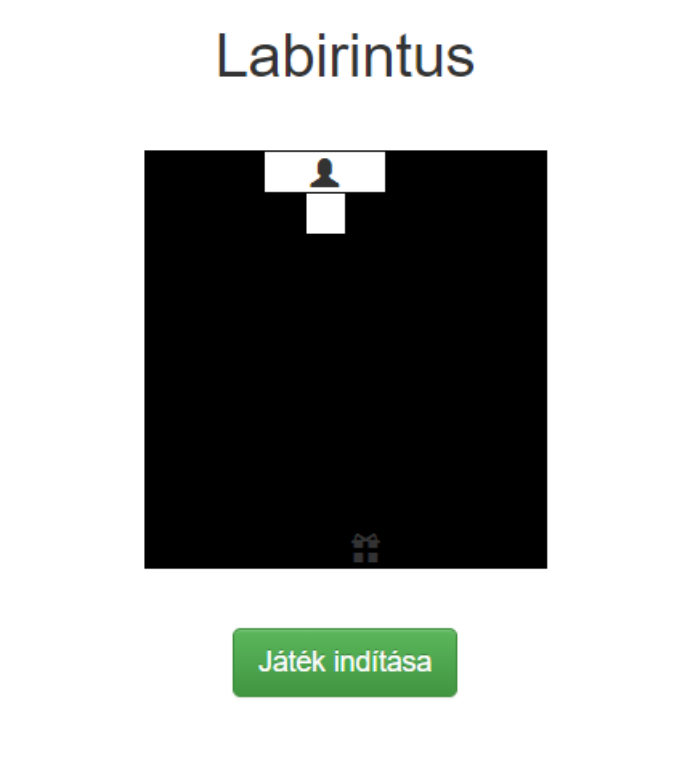
- A bomba helyzetét minden játék elején határozom meg, úgy, hogy egy-egy random számot választok a két koordinátájára.
- A játék magja, másfélpercenként hívja meg a *lepes()* függvényt, ha az egy helyes helyet határoz meg, akkor odalépteti a célzókeresztet, növeli a körök számát, majd vizsgálja, hogy véget ért-e a játék, tehát a játékos és a bomba pozíciója megegyezik-e.

Labirintus

A labirintus egy N*M-es táblán helyezkedik el. Cél a kezdőpontból kiindulva megtalálni a labirintusba elrejtett kincset.

A *kezdes(n, m, poz)*-ben megkapjuk a tábla méretét (N, M) és a kezdőpozíciót. A *lepes(poz, iranyok)*-ben pedig paraméterként megkapjuk, hogy mely pozíción tartózkodunk éppen, valamint, hogy mely irányokba lehet tovább haladni (É= észak, D=dél, K=kelet, NY=nyugat) egy tömbben, visszatérésként pedig a választott irányt kell meghatározni (pl.: „NY”).

A felület



2. ábra Labirintus játék felülete

A felületen megjelenik a labirintus, fehérrel látszódnak azok a mezők, amelyek szomszédjában már jártunk (így a falak is látszódnak), feketével pedig, ahol még nem voltunk. A „Játék indítása” gombra kattintva másfél másodpercenként meghívja a *lepes()* függvényt, amellyel a játékost mozgatja. Egy ajándék ikonnal van jelezve a kincs, amit meg kell találni.

Megoldási stratégiák

A megoldáshoz szükséges pár témakör ismerete: rekurzió, gráfábrázolás, illetve egy kis alapötlet is kell hozzá.

Alapötlet: Érdemes kiválasztani egy elsődleges irányt és ha lehetséges az arra való tovább haladás, akkor arra menni tovább, ha nem akkor az órajárásnak megfelelően egy másik irányt kéne kiválasztani. Ha zsákutcába kerültünk, akkor visszalépünk (rekurzió, backtrack[1]) és az előző szabály (1. balra, 2. egyenesen, 3. jobbra) szerint egy másik még lehetséges irányt keresünk, ha ez nem lehetséges akkor addig lépünk vissza (4. hátra), míg nincs egy még be nem járt útvonal. Ezzel a stratégiával (feltéve, ha létezik) biztosan találunk egy útvonalat az indulóhely és a cél között. Nyilván nem az optimális legrövidebb útvonalat fogjuk megkapni, ahhoz előzetesen ismernünk kéne az egész labirintust, vagy bejárni az egészet és az alapján kijelölni, kiválasztani azt (érdemes lehet egy másik feladatként azt is végig gondolni, majd megcsinálni, akár ezt a játékot is fel lehet használni a célra).

Megoldás

A labirintusban való keresés során kétféle adathalmazt használok fel, valamint módosítgatok. Az egyik egy tömb (*lepesek*), amely tárolja az eddigi egymás utáni lépéseimet, ez alapján vissza tudok lépni, ha zsákutcába kerülnek (tulajdonképpen egyfajta veremként funkcionál). A másik egy mátrix (*felfedezettLabirintus*), amely a jelen esetben 10x10-es táblánál tárolja, hogy az egyes mezőknél mely irányba mentem már, így minden esetben tudni fogom, hogy ha egy adott irány nincs benne, akkor arra még próbálkozhatok, valamint, ha minden lehetséges irány szerepel benne, akkor az nem helyes út/zsákutca, így vissza kell lépnem.

Így a következőkben nézzük részletesebben az algoritmus egyes lépéseit. Elsőként megvizsgálom, hogy az aktuális mezőből van-e még lehetséges további útvonal (nem zsákutca, vagy van olyan irány amerre még nem voltam), ha nincs akkor visszalépek.

```
if (felfedezettLabirintus[jatekos.x][jatekos.y].length == lehet-
segesIranyok.length) {
  return lepesek[--lepesekSzama];
}
```

A lehetséges irányok közül kiválasztok egyet, amerre még nem jártam (itt akár egy stratégia mentén is mehetünk, de alapjaiba véve a labirintus előismerete nélkül mindkettő csak találgatás, így nem tartottam fontosnak egy plusz logika beépítését):

```
var i = 0;
while(felfedezettLabirintus[jatekos.x][jatekos.y].includes(lehet-
segesIranyok[i])) {
  i++;
}
felfedezettLabirintus[jatekos.x][jatekos.y].push(lehetsegesIra-
nyok[i]);
```

A kiválasztott iránynak megfelelően eltárolom egy változóba a játékos új pozícióját:

[1] Részletesebben: http://progalap.elte.hu/downloads/seged/eTananyag/lecke35_lap1.html

```
var ujPoz, ellentetIrany;
if (lehetsegesIranyok[i] == "NY") {
  ellentetIrany = "K";
  ujPoz = {
    x: jatekos.x,
    y: jatekos.y - 1
  };
} else if (lehetsegesIranyok[i] == "É") {
  ellentetIrany = "D";
  ujPoz = {
    x: jatekos.x - 1,
    y: jatekos.y
  };
} else if (lehetsegesIranyok[i] == "K") {
  ellentetIrany = "NY";
  ujPoz = {
    x: jatekos.x,
    y: jatekos.y + 1
  };
} else if (lehetsegesIranyok[i] == "D") {
  ellentetIrany = "É";
  ujPoz = {
    x: jatekos.x + 1,
    y: jatekos.y
  };
}
lepesek[++lepesekSzama] = ujPoz;
```

Végül pedig az ellentétes irányt is eltárolom a mátrixba, hogy a következő választásnál azt már ne válassza, hiszen az egy felesleges visszalépésnek számítana. Ezek után a függvény visszatér a játékos új poziciójával:

```
felfedezettLabirintus[ujPoz.x][ujPoz.y].push(ellentetIrany);
return ujPoz;
```

A honlapról

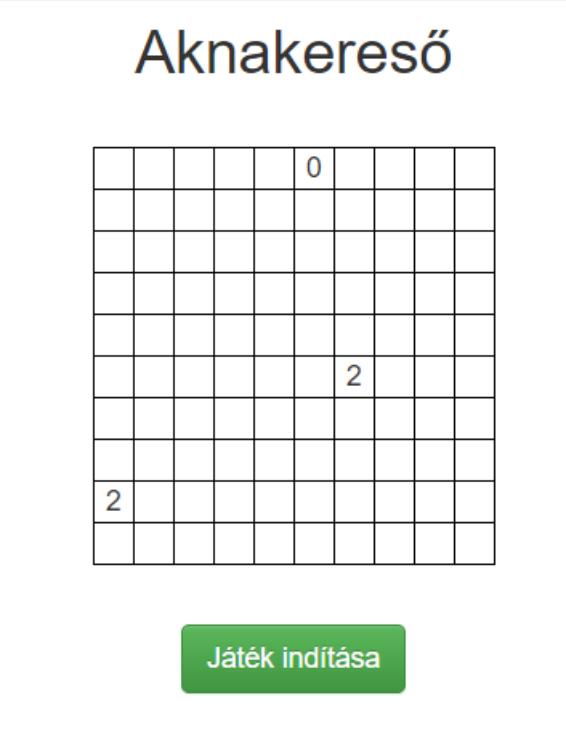
- A labirintus egésze már előre le van generálva, tehát minden elindításkor ugyanott lesznek a falak és a kincs. Így egyszerűbbé, kiszámíthatóbbá válhat a megoldás, lehetőség van ugyanazon labirintusra ellenőrizni, hogy hol is tartunk, mi okozhatja az esetleges hibákat. Természetesen ezt adott esetben átírhatjuk, fontos ilyenkor, hogy mindenképpen legyen út a játékos kezdőhelyétől a célig.
- A labirintus gráfját egy NxM-es mátrixban tárolom, amely minden egyes eleme egy pozíciót reprezentál, ehhez pedig értékként a lehetséges irányok kezdőbetűi vannak eltárolva egy tömbben.
- A lépések hívásakor folyamatosan figyelem, hogy megengedhető-e a játékos lépése. Ha a játékos és a kincs pozíciója megegyezik, akkor vége a játéknak.
- Egy külön mátrixban tárolom, hogy a játékos hol járt már, így ennek megfelelően lehetnek fehérek a már bejárt területek szomszédjai, valamint fekete a többi cella.

Aknakereső

Az aknakereső játék lényege, hogy az NxM méretű táblán meg kell találni K db aknát, mindezt úgy, hogy ha kiválasztunk egy helyet és az nem akna (az első semmiképpen sem az), akkor ott megjelenik, hogy hány akna van a közvetlen közelében. A cél az összes olyan mező megtalálása, ahol nincs akna.

A *kezdes(n, m)*-ben megkapjuk a tábla méretét (N, M). A *lepes(matrix)*-ben pedig megkapjuk az eddig felfedezett terület mátrixát, ahol már voltunk ott a szomszédos aknát számával, ahol még nem ott üres mező szerepel.

A felület



3. ábra Aknakereső játék felülete

A játék elindítása gomb hatására minden kiválasztott mezőre megjelenik, hogy azon mező szomszédságában hány akna van. Ha egy olyan mezőre lépünk, melyen akna van, akkor a játék befejeződik.

Megoldási stratégiák

Előre le kell szögezni, hogy vannak szituációk a játékban, amikor „tippelnünk” kell, tehát nem biztos, hogy tudjuk mely helyen van akna vagy sem, így ezáltal teljes, egész, mindent megoldó algoritmust sem tudunk írni. Azonban mindenképpen érdemes sémákat felismerni a játék során, majd azokat egymás után felhasználni. Feltehetőleg hasonló módon tanultunk meg mi is végig vinni egy aknakeresőt pályát, majd a végén reménykedtünk, hogy nem bukkan fel az a szituáció, amikor a szerencsén múlott a pálya megoldása.

Az alap stratégia, ami egyszerűbbeknél végig működhet, hogy ha egy négyzet mellett pont annyi szabad hely van, ahányat mutat, akkor mellette biztos aknák vannak, majd a többi négyzetből ezáltal következtethetünk, hogy mellettük szerepel-e még akna.

Ezen kívül még gyakran találkozhatunk a következőkkel:

- Egy oldalon nyitott 1-2-1 felbontás, ekkor biztosan az egyesek felett van a két akna.
- Egy oldalon nyitott 1-2-2-1 felbontás, ekkor biztosan a két kettes felett van a két akna.

Ezekkel többségében már meg lehet oldani az egyszerűbb feladatokat (mint amilyen a weboldal is). Viszont nagyobb táblán, több aknával nem feltétlenül jelentenek biztos sikert ezek az egyszerűbb stratégiák, összetettebb dolgokat is lehet, hogy fel lehet használni (érdemes a megoldás után változtatni a tábla méretét/aknák számát, így nehezíteni akár feladatot). Ha valakit érdekelnek ezek, akkor mindenképpen érdemes minél többet felismerni, implementálni a megoldásunkban (ennek megfelelően az oldal aknalerakó algoritmusát is lehet módosítása nagyobb pályamérettel/több aknával). Ehhez segítséget nyújthat számunkra a következő weboldal: http://www.minesweeper.info/wiki/Strategy.

A honlapról

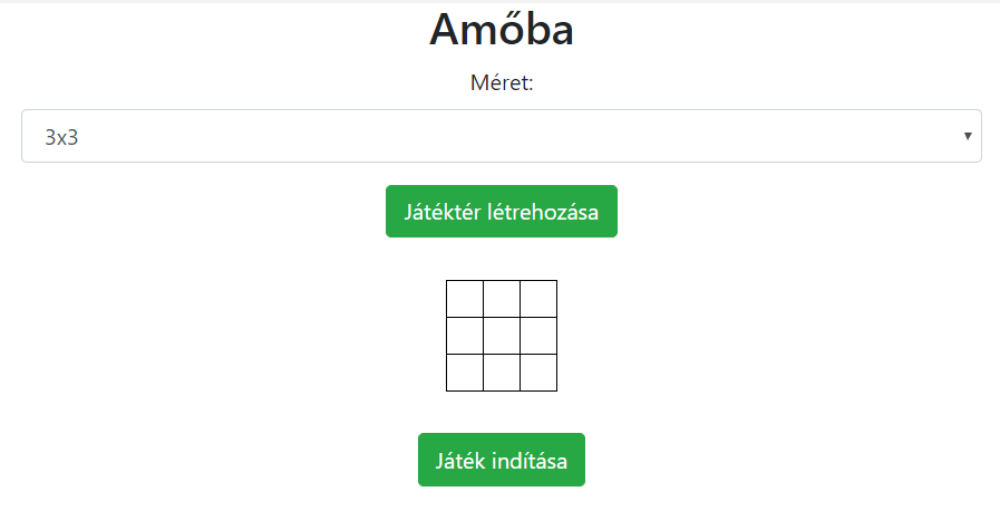
- Mindenképpen oda kell figyelni, hogy az első lépés kiválasztásánál nem lehet bomba, így a bombák pozícióját érdemes ezután megállapítani.
- Már az elején legeneráltam az összes mező értékét, így nem kell minden egyes lépés után kiszámítani, hány akna is van a szomszédjában.

Amőba

A játék talán mindenki számára ismerős: két játékos egy négyzetrácsos lapon (jelen esetben egy táblázatban) felváltva tesznek „X”-et, illetve „O”-t. Az nyer, akinek először összejön függőlegesen, vízszintesen, vagy átlóban az öt figura. Ennek egy rövidebb fajtája a Tic-Tac-Toe, mikor ezt csak egy 3x3-as táblázatban játszák és ott csak 3-at kell hasonló módon összegyűjteni.

Jelen esetben a diáknak a feladat egy „játékos” (mesterséges intelligencia) létrehozása, amely maga választja ki a következő lépés helyét. Tehát a *lepes(matrix, kovetkezoJatekos)* függvény megkapja a tábla eddigi állapotát, illetve, hogy mely jelzés is az övé (a következő játékosé), ezt követően pedig a következő lépés pozícióját kell visszaadni (az első játékos függvénye lepes1, míg a másodiké lepes2 kell, hogy legyen, mivel egyéb esetben a függvények nevei ütköznének).

A felület



4. ábra Amőba játék felülete

Először választhatunk, hogy 3x3-as (3 azonos szimbólum kell a győzelemhez), vagy 10x10-es (5 azonos szimbólum kell a győzelemhez) táblát szeretnénk. Majd a játék indítása gombra kattintva a két játékos *lepes* függvényeit hívogatja sorba, míg valaki nem nyer.

Megoldási stratégiák

Az első talán „emberközelibb" irány az lehet, ha átgondjuk, hogy mi magunk is milyen stratégiával játszunk, választjuk a következő lépést és annak megfelelő algoritmust implementálunk. Itt a következő gondolatok kerülhetnek szóba:

- Ha egy játékos közel van a győzelemhez, akkor megpróbálom kivédeni azt. Ez akkor fordulhat elő, ha a lépésünk előtt:
    - Van egy nyitott hármas (egyik szélén sincs általunk lerakott jel)
    - Egy lépéssel a másik játékos két egymásba helyezett nyitott hármast szerez
- Az előzőt meg is lehet fordítani, tehát megpróbálunk nyitott hármasokat, akár egymásba helyezni a győzelemhez
- Ha nincs olyan lépés, amivel ilyet tudnánk létrehozni, akkor megpróbálunk úgy rakni, hogy közel kerüljünk hozzá (egyszerűbb esetben első lépésként egy random szomszédos négyzetet is választhatunk, majd ezt később finomítjuk tovább)
- 3x3-as esetén akár egy konkrét lépés sorozatot is lehet definiálni, nyilván ez a 10x10-es verziónál már nagyon időigényes lenne, és nem is igazán tud jó megoldás lenni

Érdemes minél többet játszani, próbálkozni, hogy újabb stratégiákat, lépés sorozatokat ismerjünk fel.

A másik megközelítés ennél sokkal összetettebb csak igazán jó játékosok, vagy egy gyors számítógép tudja kivitelezni. Ez arról szólna, hogy az összes létező jövőbeni lépést legeneráljuk egy gráfba, majd minden szintjén kiválasztjuk azokat a lépéseket, amik az éppen adott játékosnak legnagyobb eséllyel hoz győzelmet (ez a minimax algoritmus). Ahogyan utaltam rá, ez nagyon időigényes és a memóriát sem kíméli, gondoljunk csak bele, hogy minél több szint van, aszerint hatványozodik a fa leveleinek száma.

```
Eljárás minimax(csúcs, mélység)
  Ha a csúcs levél, or mélység = 0
    minimax := a csúcs heurisztikus értéke
  különben
    α := -∞
    Ciklus gyerekére a csúcsnak
      α := max(α, -minimax(gyerek, mélység-1))
    minimax := α
}
```

Az első sakk mesterséges intelligenciák is hasonló elven működtek, ezért is kellett hozzájuk már akkor is egy-egy erősebb, gyorsabb számítógép. A sakknál persze többféle lépés lehetséges, mint egy amőbában. Érdekesség: 1997-ben létrehozott Deep Blue direkt erre kialakított számítógép annak idején 3,5-2,5 arányban legyőzte Kaszparovot (akkori világbajnok), amely 40 lépéspár mélységig tudta vizsgálni a játékot (csak hasonlításképpen Kaszparov becsült keresési mélysége 10-12 pár volt).

Mindenképpen a 3x3-assal lenne érdemes elkezdeni foglalkozni, hiszen ott hamarabb össze lehet rakni egy működő algoritmust, hamarabb rá lehet jönni apróbb trükkökre, stratégiákra, aztán ezekből kiindulva lehet aztán a rendes amőbába is bele kezdeni.

A honlapról

- Több kódolást igényelt, hogy minden irányban vizsgálva legyen, hogy nyert-e már valaki (plusz a vizsgálatnál érdemes figyelni a határokra, nehogy egy olyan elemre próbáljunk hivatkozni, ami nem is létezik).
- A játéktér hasonlóan az előzökhöz egy mátrixban tárolódik, ezt kapják meg a játékosok a *lepes* függvényeikben.

Felhasználás

Érdemes 1-1 feladatra 1-2 szakköri órát rászánni. Nem csak a feladatok elolvasása és stratégiák értelmezése fontos, hanem a kettő közötti közös/egyénenkénti ötletelés, hogy miket lenne érdemes felhasználni, milyen témakörök jöhetnek szóba, amikkel már régebben találkoztunk. Bizonyos feladatoknál az otthoni tökéletesítésre is érdemes lehetőséget adni, főleg az MI (mesterséges intelligencia) közeli témáknál sok lehetőség nyílik erre.

További lehetőségek

Érdemes a fentebb említett témákat, feladatokat tovább gondolni és az eddig létrehozott feladat környezeteket átfejlesztve újabb játékokat tervezni, kreálni. Lehetnek ezek konkrét problémák megjelenítése vizuális környezetben, vagy akár egy-kétszemélyes játékok, amelyekben feladat egy „mesterséges intelligencia” elkészítése.