



Eötvös Loránd Tudományegyetem

Informatikai Kar

Algoritmusok és alkalmazásaik Tanszék

---

# A mesterséges intelligencia módszerei a HEX kétszemélyes táblajátékban

Témavezető

Szerző

Dr. Szabó László

Ószi Krisztián

Egyetemi docens, habilitált doktor

Programtervező Informatikus MSc

Budapest 2021

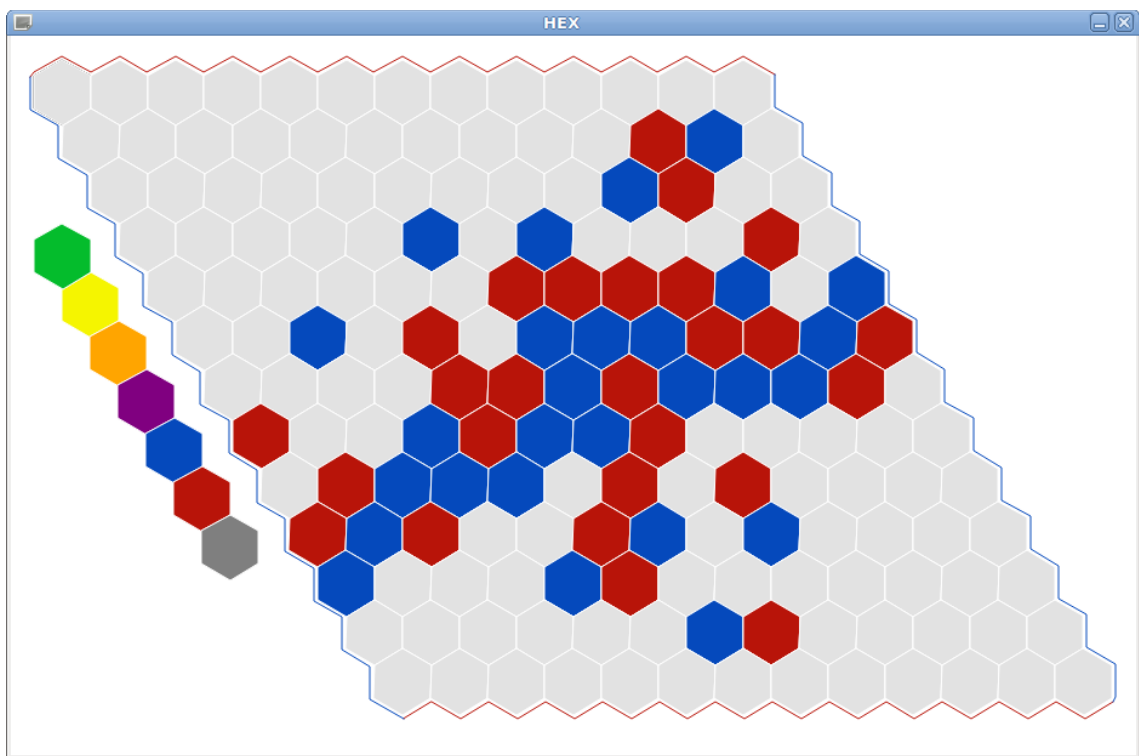
# Tartalom

1	Bevezetés.....	- 3 -
1.1	John Nash & Piet Hein.....	- 5 -
1.2	Versenyek.....	- 5 -
2	Kapcsolódó munkák.....	- 6 -
2.1	Computer HEX.....	- 6 -
2.2	Neurohex.....	- 7 -
2.3	OHex.....	- 8 -
2.4	Six.....	- 9 -
3	A HEX játék.....	- 10 -
3.1	Nyerő stratégia.....	- 10 -
3.2	Zérusösszegű játék.....	- 10 -
3.3	A játék komplexitása.....	- 11 -
3.4	A játék stratégiai és mintái.....	- 11 -
4	Mesterségei intelligencia módszerei.....	- 17 -
4.1	Heurisztikák.....	- 18 -
4.2	Gráfkeresési stratégiák.....	- 21 -
4.3	Min-max.....	- 22 -
4.3.1	Átlagolás.....	- 23 -
4.3.2	Változó mélység.....	- 23 -
4.4	Alfa-béta algoritmus.....	- 24 -
4.5	Monte Carlo algoritmus.....	- 25 -
4.5.1	UCB.....	- 27 -
4.6	Adatbázisok.....	- 32 -
5	Mesterséges neuronhálók.....	- 33 -
5.1	Gépi tanulás (Machine Learning).....	- 33 -
5.2	Mesterséges neurális hálózatok áttekintése.....	- 34 -
5.2.1	Biológiai neuronok.....	- 36 -

5.2.2	Mesterséges neuron.....	- 37 -
5.2.3	Neuronok hálózata.....	- 39 -
5.2.4	Előrecsatolt hálózatok.....	- 40 -
5.2.5	Aktivációs függvények.....	- 41 -
5.2.6	Optimalizáció és veszteségfüggvény .....	- 43 -
5.3	Tanulási adatbázis.....	- 45 -
5.4	Adat reprezentáció.....	- 46 -
5.5	Neurális hálózat tanítása .....	- 48 -
5.6	Rétegek.....	- 50 -
5.7	Predikció .....	- 51 -
6	Programozási áttekintés.....	- 52 -
6.1	Felület rajzolása .....	- 52 -
6.2	Többszálás programozás .....	- 55 -
6.3	Virtuális környezet.....	- 58 -
6.4	Python virtuális környezet.....	- 61 -
7	Felhasználói kézikönyv .....	- 62 -
7.1	Követelmények.....	- 63 -
8	Üzembe helyezés .....	- 67 -
8.1	Neurális hálózat elkészítése .....	- 67 -
8.1.1	Játékadatbázis előkészítése.....	- 67 -
8.1.2	Játékadatbázis előkészítése.....	- 67 -
8.1.3	Tanítás.....	- 68 -
8.1.4	Ellenőrzés.....	- 68 -
8.1.5	Interfész tesztelése .....	- 68 -
8.2	Applikáció indítása.....	- 68 -
9	Összefoglalás .....	- 69 -
9.1	Eredmények.....	- 69 -
9.2	Köszönetnyilvánítás .....	- 70 -
10	Referenciák.....	- 71 -

# 1 Bevezetés

A HEX [1] egy kétszemélyes táblajáték, melynek táblája rombusz alakú és hatszög alakú mezőkből áll. A játékosok felváltva helyeznek el egyet-egyét a korongjaik közül valamelyik mezőn (a játékosok korongjai különböző színűek), céljuk, hogy utat (hidat) építsenek a rombusz két-két szemközti oldalai között. Az a győztes, akinek ez előbb sikerül. A játékot az 1940-es években találta ki két matematikus, Piet Hein és John Nash, elsősorban a saját szórakoztatásukra, de kiderült, hogy mély matematikai vonatkozásai is vannak. Érdekes, hogy a készítőik egymástól teljesen függetlenül fedezték fel a játékot, 6 év eltéréssel.



1. ábra:  
*HEX demo program*

1942-ben a dán Politiken folyóiratban jelent meg először 11x11-es méretben és CON-TAC-TIX néven. A HEX nevet később kapta, amelynek amerikai vonatkozása van. Az alábbi neveken is ismerhetjük: Nash-Jhon, Banoochee, Hexy, Hexomania, Queenbee vagy Polygon.

A játék népszerűségének több oka is van. Egyszerű szabályokhoz kötött, nincsenek nyelvi korlátjai, mindenki megtanulhatja, könnyen lehet benne fejlődni. Habár egyszerű, igazából egy stratégiai játék.

A klasszikus táblaméret 11x11 hatszöget tartalmaz, játszáskisebb, illetve nagyobb táblával is. A versenytábla jellemző mérete 13x13, ezért ezt lett a program alapértelmezett táblamérete is.

A HEX táblajáték nem olyan egyszerű, mint első ránézésre tűnik. Többfajta játszható stratégia van a játékban, ezért is érdekes kutatási téma a mesterséges intelligenciában. A tábla méretével a lehetséges játékmenetek száma hatványozottan megnő. A teljes játékfa kiértékelése még a klasszikus, kisebb méreteken is egy önálló projekt lehet. A tény, hogy egy mezőnek 4 helyett 6 szomszédja van, két nem szomszédos mező összekötése többféleképpen lehetséges, illetve, hogy az ellenfélnek más blokkolási stratégiára van szüksége, egy más jellegű játékot ad. A HEX érdekes, így esett a témaválasztás erre a táblajátékra, látszólagos egyszerűsége melletti komplexitása kihívássá teszi.

**A dolgozat fő célkitűzése az lett, hogy a játékstratégiák mély ismerete nélkül (de azokat nem figyelmen kívül hagyva), hogyan lehetséges, illetve tudunk-e, egyszerűen átlátható módszerekkel és neurális hálókkal, más táblajátékokra könnyen átvihető megoldást adni a mesterséges intelligencia jelenlegi módszereivel a HEX játék megnyerésére.**

A dolgozatban megvizsgáljuk a játék stratégiáit, hogyan alkalmazhatjuk rá a mesterséges intelligencia különböző módszereit és a modern mesterséges neuronháló adta lehetőségeket, illetve azok kombináltjait. Fontos szempont, hogy a megoldás ne legyen túl specializált, átvihető legyen más játékokra.

Sajnos két akadályozó tényezőt lépett fel. Egyik, hogy a nagy teljesítményű eszköz elérésének reménye teljes mértékben szertefoszlott. A másik, hogy az eredetileg a neurális hálózat tanításához szánt könyvtárakhoz és azok további függőségeihez nem áll rendelkezésre fordításhoz és futtatáshoz szükséges elégséges információ vagy bináris.

## **1.1 John Nash & Piet Hein**

John Nash [2] amerikai matematikus volt. A matematikai játékelméletben elért eredményeiért osztott formában részesült közgazdasági Nobel-díjban. Szülei tanárok voltak, az iskolában jeleskedett matematikában és kémiában, majd tanulmányait az amerikai Princeton egyetemen folytatta, ahol a doktori disszertációját a „non-cooperative games” témában írta, mely később a Nobel-díjhoz vezette őt. A Nobel díjat végül 1994-ben megosztva kapta meg Reinhard Selten-nel és John Harsányi-val.

Piet Hein [3] dán polihisztor a matematikában, a dán építészetben és az irodalom terén is maradandót alkotott. A Koppenhágai Metropolitan Egyetemen több szakot is elkezdett, melyeket végül nem fejezett be közben festészetet is tanult. Ezután publikálta a HEX játék első változatát egy dán napilapban. Később az általa javasolt szuper-ellipszis használata az építészetben a dán építészet védjegyévé vált. A Yale és Odense Egyetem díszdoktorának választották.

## **1.2 Versenyek**

Noha a HEX nem olyan közismert, mint a Sakk vagy a Go, de HEX versenyeket is rendeznek világszerte, embereknek is és gépeknek is. Ilyen a 2000-től a gépek számára rendezett Computer Olimpiad is. A 2003, 2004 és 2006 években megrendezett Computer Olimpiad-on az első helyet a magyar fejlesztésű Six (Melis Gábor) nyerte, megverve a később élre törő Mohex/Walve elődjét, a Mongoose-ot.

## 2 Kapcsolódó munkák

A következő fejezetben a diplomatémához kapcsolódó HEX tanulmányokat és eredményeket tekintjük át röviden.

### 2.1 Computer HEX

A Computer Hex-et [4] a 2008-as megjelenése óta a legerősebb Hex algoritmusként tartják számon. Számos módosítás, kiegészítés érkezett hozzá az évek során, több mint 20 ember aktívan fejlesztette és javította. 2008-2015 között, a Mohex, vagy elődje, a Wolve nyert a ICGA Olympiad Hex bajnokságon.

A **Wolve** a tradicionális alfa/béta játédfa kiértékelést használja és kombinálja mintakereséssel. Az algoritmus mustplay és notplay mintákat keres és követ.

A **MoHex** a Monte Carlo algoritmust használja. Tehát hasonló elven működik, mint a Wolve, de csak egy mintát használ (veszélyes hidak blokkolása). A Monte Carlo keresési fa implementációját oly módon kombinálja játédfa és alfa-béta vágások használatával, hogy az egyértelműen veszélyes (tehát a másik játékos számára előnyös) helyzetek, azaz a dupla hidak kialakulását blokkolja.

A **Solver** a Wolve elődje. 9x9-es táblaméretig ismeri a nyerő stratégiákat. Nagyobb táblaméret esetén dekompozíciót használ, mely során a tábla egy 9x9-es részét és a fennmaradó területet külön próbálja megoldani.

A Mongoose és a Queenbee korábbi verziók. Sajnálatos módon a projekt már pár éve nem aktív, és mivel a szerzők a fordításhoz szükséges verziókat és a

lefordított binárisokat nem teszik elérhetővé, a munka a könyvtárral nehezített. Így már a függőségek fordítása is nehézséget jelentett a maga több száz hibájával. A sikeres fordítás után a mellékelt egység és regressziós tesztek ugyan sikeresen lefutottak, de a programkód további hiányosságokat tartalmazott. A programkód javítása egyrészt online fórumokról összegyűjtött korábbi hozzászólások alapján, git fork-ok alapján és saját intuíció alapján történtek. A program ennek ellenére gyakran elszáll futási idejű hibával (segmentation fault), így nem megfelelő.

## 2.2 Neurohex

A Neurohex [5] Deep Q-learning-et és konvolúciós neuronhálót használ, hogy megtanulja játszani a HEX-et. A Computer Hex projekt, azon belül is a MoHex algoritmusát futtatták saját maga ellen 13x13-as táblán, hogy előállítson teszt adatokat. Az előállított kb. 11 000 játszámát és 551 000 játékállapotot felhasználva készítettek neurális hálózatot. A hálózatot 2 hétig tréningezték saját maga ellen. A megoldás első lépésből ~20%-ban, második lépésből ~1%-ban volt képes nyerni a MoHex, a korábbi ICGA Olympiad Hex bajok algoritmussal szemben – amiből a diplomamunka neurális hálózata is készült.

A Q-tanulás (Q-learning) egy cselekvés-érték (action-value) függvényt használ. Egy aktuális állapotból egy másik állapotba való jutáshoz értéket rendel, amit maximalizálni próbál.

A github oldalon GNU GPL liszenszelés alatt elérhetővé tették az általunk használt adatbázist. A neurális hálózat tanítását az általuk biztosított adatbázis alapján végeztem el. Ezt ezúton is köszönöm.

A Neurohex szintén egy pár éves projekt, ami részben a Computer Hex fork-ja, de ugyanazok a fordítási problémák jelentkeztek, mint az eredeti forráskódban.



## 2.3 OHex

Az OHex [6] egy adatbázis, ami 4x4-től 11x11-es táblaméretig tartalmaz összegyűjtött játszmákat és állapotokat, legtöbbször 10x10 méretben. Az OHex lehetőséget ad, hogy adatbázis alapján játszunk.

táblaméret	játékok	állapotok
4x4	365	1661
5x5	164	1298
6x6	288	3417
7x7	1963	33954
8x8	137	3268
9x9	2054	54628
10x10	798395	24507299
11x11	109025	3972692

1. táblázat

*Ohex adatbázis*

Az adatbázis a valószínűsíthetően nyertes és valószínűsíthetően vesztes pozíciókat tartalmazza. Tehát azt tárolja, hogy egy adott pozícióból a játékosok nyertek vagy veszítettek. Az adatbázis különböző forrásokból lett összegyűjtve, többnyire emberi játékokból. A készítő felkereste a HEX játékot szolgáltató weboldalakat, és elkérte a lejátszott játékok adatbázisait. A gyűjtést követően azt egységes formára hozta és azokat online módon böngészhetővé tette. Ez nagyon sok tárolt adat, de még így is messze van a teljes állapotgráf.

Eredetileg ennek az adatbázisnak a 11x11-es változatát akartam felhasználni, de a készítő ezt nem teszi elérhetővé.

## 2.4 Six

A 2003, 2004, 2006 években megrendezett Computer Olimpiad-on az első helyet a magyar fejlesztésű Six [7] (Melis Gábor) nyerte. Sajnos a készítő, illetve a projekt weblapja már nem elérhető, csak annyi tudható biztosan, hogy alfa-béta algoritmust használt [8].

## 3 A HEX játék

### 3.1 Nyerő stratégia

Eddig 9x9-es táblaméretig sikerült kigenerálni a teljes játékfát. Ezen méretig ismert az első játékos számára az összes nyerő stratégia [6].

A játék nem végződhet döntetlennel, vagyis egy korongokkal telített táblán az egyik játékosnak van útja a saját két oldala között. Az egyetlen módja annak, hogy feltartóztassuk az ellenfelet a saját útvonalának építésében, ha mi építjük azt fel. Ez persze nagyon intuitív bizonyítás. John Nash elmondása szerint ezt formálisan is sikerült bizonyítani, de ezt sohasem publikálta. Végül David Gale bizonyította a Brouwer-féle fixpont tétel alkalmazásával [9].

### 3.2 Zérusösszegű játék

A táblán két játékos játszik egymás ellen. Az egyik játékos lépése szükségszerűen hatással van a másik lépésére. A kétszemélyes táblajátékok esetén az egyik játékos vagy nyer (+1) vagy veszít (-1). A matematikai játékelmélet (game theory) ezt zérus összegű játéknak nevezi. Mivel a játszma során mindkét játékos teljes információval rendelkezik a játék állapotteréről, teljes információjú játékról is beszélünk.

### **3.3 A játék komplexitása**

A játék bonyolultság szempontjából PSPACE osztályba tartozik [10]. Tehát determinisztikus Turing-géppel polinomiális szalagigénnyel kiszámítható.

### **3.4 A játék stratégiái és mintái**

A HEX egy kétszemélyes táblajáték, melyben a játékosok különböző színű korongokat helyeznek fel a táblára. A játék megnyerése érdekében a játékosok stratégiákat és mintákat követnek [11]. A meghatározott taktikával jobb pozícióba kell a játékosnak kerülnie, ami a játszma megnyeréséhez vezet. A jó stratégia figyelembe vesz több lehetséges lépéssorozatot, és nem csak a tábla egy lokális részére koncentrál, hanem figyelembe veszi a globális játékeret.

#### **3.4.1 Kapcsolt mezők (connection)**

Amikor két szomszédos mezőn azonos színű korongok foglalnak helyet, a legerősebb linket (kapcsolatot) kapjuk meg a játékban. A korongok egymás szomszédságában láncot alkotnak.

#### **3.4.2 Egy-híd (one-bridge) és két-híd (two-bridge)**

Egy-hídnak nevezzük azt a mintát, amikor két mező egy kapcsoló korong segítségével láncú alakítható egy lépéssel. Ez a kapcsolat nem erős, hiszen mindösszesen egy korong letételével blokkolható.

Két-Hídnak nevezzük azt a mintát, amikor két mező kétféleképpen is kapcsolható mindössze egy kapcsoló koronggal. Ez a típusú link a legerősebb a szomszédos korongok után.



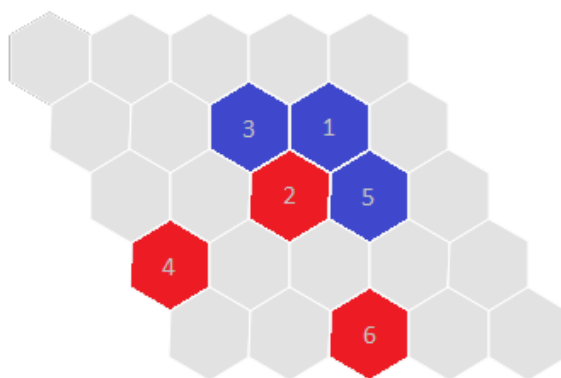
2. ábra:  
kapcsolat, egy-híd és két-híd

A hidak kialakítását elvágó módszerekkel (blokádkkal) akadályozhatjuk meg.

### 3.4.3 Elvágás (V-blokád)

Az ellenfél láncot épít, amit szeretnék elvágni (vagyis blokkolni). Ezt a lánc és a tábla széle között tesszük meg. Erre jó megoldás az elvágó minta. A megfelelő irányban kicsit távolabb kell felépíteni. Eközben valószínűleg sikerül létrehozni két-híd mintát is, amit később felhasználhatunk a saját láncunk építéséhez, de a blokkolt játékos is építhet közben láncot, hidakat.

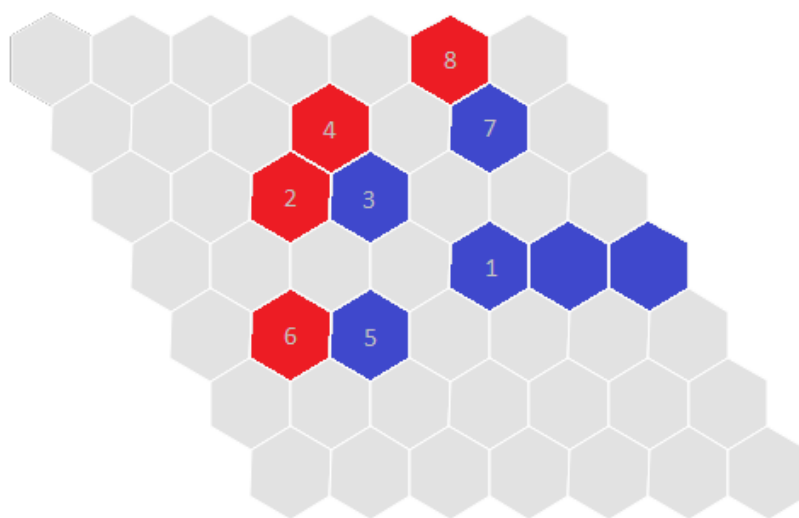
Amikor a piros játékos letette a 2. sorszámmal jelzett korongját, abban a pillanatban a 2-4-6 sorszámú mezők közötti háromszög alakú terület a kék számára érdektelenné vált.



3. ábra:  
elvágás

#### 3.4.4 Elvágás (U blokad)

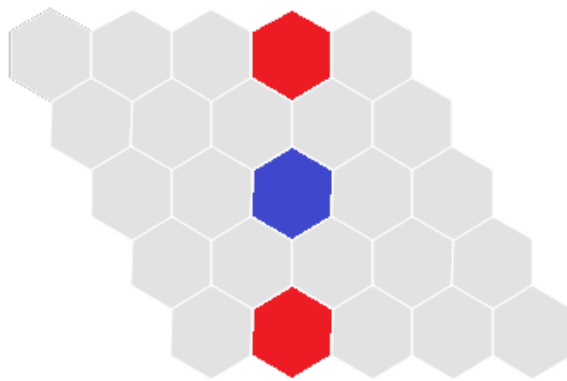
Az U blokad vizuálisan pont a fordítottja az előzőnek. A blokádot nem szabad túl közel építeni, mert átmehet rajta az ellenfél. Ennek folyamán szintén több két-híd alakul ki.



4. ábra:  
U blokad

### 3.4.5 Hidak felügyelete

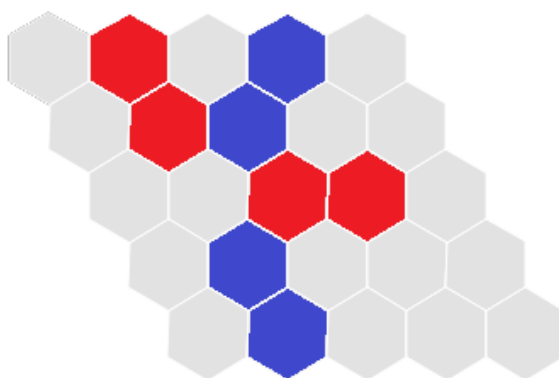
Ebben a mintában szükséges felismerni, hogyan alakulnak ki potenciálisan két-hidak. A stratégia szerint meg kell akadályozni az ellenséges két-hidak kialakulását. Ha két távolabb lévő korong közé lerakunk egy harmadikat, két egymást követő két-híd alakul ki. Ez egy nagyon erős minta, hiszen könnyen 5 hosszú lánc építhető belőle.



5. ábra:  
*hidak felügyelete*

### 3.4.6 Pivot pontok

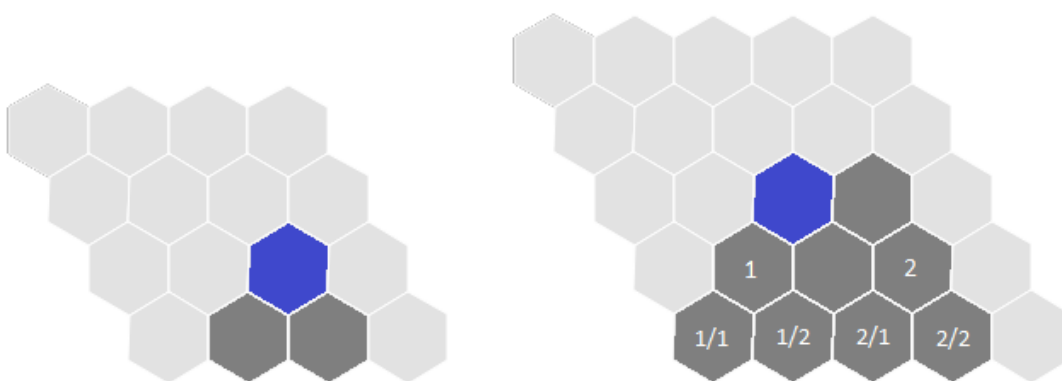
Kialakulhatnak stratégiai pontok, amelyek egyszerre láncot alkotnak és blokkolnak. Előre látni ezeket nagyon nehéz feladat. Hasonlóan pivot pont a 4 sarok és a szomszédos mezők. Néhány HEX variánsban lehet a második lépésben cserélni (swap, avagy csereszabály), így a kezdő játékos előnye csökkenthető, de ezzel most csak említés szintjén foglalkozunk.



6. ábra:  
pivot pontok

### 3.4.7 Él minták

Amikor a játék a tábla széléhez közelít, képbe kerülnek az él sablonok (edge templates). Az él sablon azt írja le, hogy egy adott pontból elindulva hogyan építkezzünk a tábla széléig, hogy azt biztosan elérjük, az ellenfél bármely lépése esetén. Tehát nyerő stratégia a tábla széléig. A módszer akkor működik, ha a korongunk és a táblaszél által bezárt háromszög alakú terület üres. Az 1 és 2 méretű él sablon könnyen megérthető, gyakorlatilag egy vagy két darab két-híd.

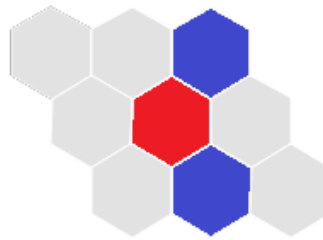


7. ábra:  
Él minták



### 3.4.8 Kényszerítés

Az ellenfél két-hídját blokkolni gyakran fölösleges, mert ha egyik mezőre korongot helyezünk, az ellenfelünk a másikra helyezi és felépítette a kapcsolatát. Előfordulhat, hogy ezt mégis érdemes meglépni, mert az így elfoglalt mező később értékes lehet számunkra.



8. ábra:  
*kényszerítés*

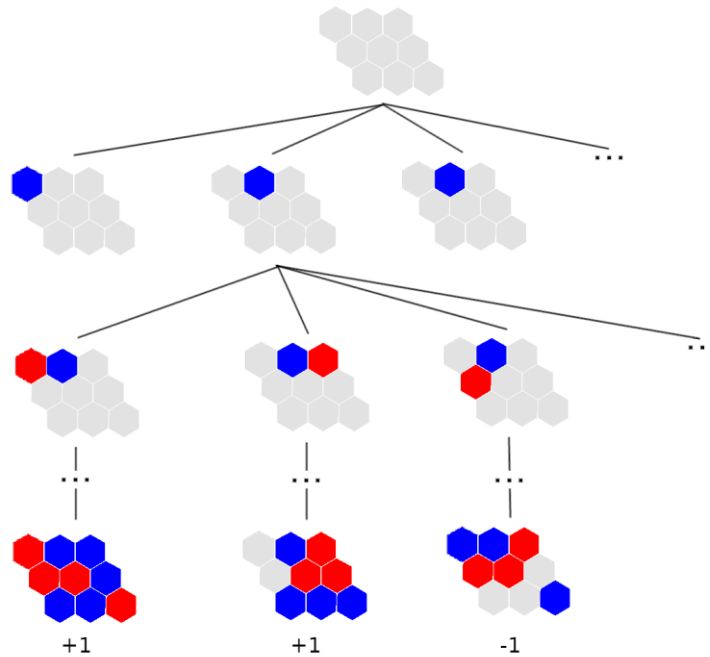
## 4 Mesterségei intelligencia módszerei

A kétszemélyes játékokban mindkét játékosnak számolnia kell az ellenfél lépéseivel, illetve, hogy a saját lépései hogyan hatnak a másik lépéseire. Az ellenfél előre nem kiszámolható gondolkodása miatt számos lehetőséget meg kell vizsgálni.

Ahhoz, hogy megfelelő döntéseket hozzunk (jó lépéseket lépjünk), a játékot egy gráfként definiáljuk. Ezután gráfkeresési problémaként tekintünk rá.

- A kiinduló csúcs lesz a kezdőállapot, illetve definiálja azt is, hogy melyik a következő játékos.
- Az állapot átmenetek lesznek a gráf élei, tehát, hogy egy bizonyos állapotból milyen lehetséges lépések léteznek és a lépés hatására melyik állapotba jutunk.
- A terminálási teszt eldönti, hogy a játék véget ért-e. Azt az állapotot, ahol a játék befejeződött, terminálási állapotnak nevezzük.
- Továbbá létezik egy hasznosságfüggvény, ami a végállapothoz egy értéket rendel. A HEX esetében +1 és -1 értékeket rendelhetünk a végállapothoz, tehát nyert és veszített.

A fent említett módszer segítségével egy játékfának nevezett gráfot építünk fel, amit kis tábla méret esetén teljes mélységében ki tudunk értékelni.



9. ábra:  
játékfa

Az útkeresési probléma egy olyan út megtalálása a gráfban, ami nyertes állapotba vezet. Ez a játékfa, még az egyszerűbb játékok esetében is túl nagynak bizonyul.

## 4.1 Heurisztikák

Ahogy az előző pontban láttuk, a teljes játékfa kiértékelése a processzor és memória kapacitás korlátaiba ütközik. A gráf mérete miatt csak egy kis részét tudjuk átvizsgálni. Tehát a gráf lokális vizsgálatával kell következtetnünk a végállapot hasznosságára. Ennek eszköze a heurisztikus függvény, ami minden állapothoz egy heurisztikus értéket rendel. A megfelelő heurisztikus függvény problémánkét teljesen más lehet.

A HEX esetében most a heurisztikus függvény előállításához először egy újabb függvényre lesz szükségünk, ami megállapítja, hogy egy adott állapotban a kék,

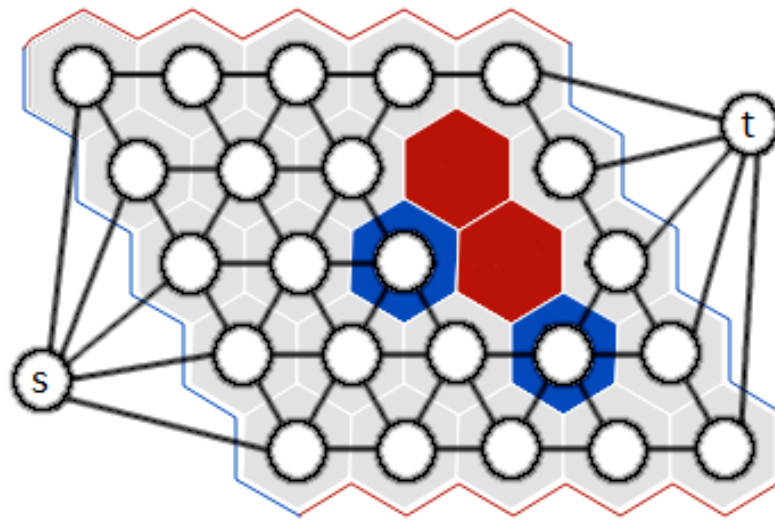
illetve piros játékosnak a legideálisabb esetben még hány lépést kell megtennie, hogy nyerni tudjon. Ehhez első lépésben az állapottérből gráfot fogunk készíteni, mindkét játékos számára egyet-egyet. (Hasonlóan a David Gale cikkben használthoz [9].)

- Definiáljuk a játékos színét
- A táblán lévő mezők legyenek a gráf csúcsai
- A szomszédos mezők által reprezentált csúcsok között éleket helyezünk el az alábbi feltételek szerint:
  - Két üres mező között irányított éleket definiálunk, súlyuk pedig mindkét irányban legyen 1.
  - Két ellentétes színű mező között nem definiálunk éleket.
  - Egy saját és egy ellentétes színű mező között nem definiálunk éleket.
  - Egy üres mező és egy saját színű mező között éleket definiálunk. Az üres mezőbe vezető él súlya 1, míg a színezettbe vezető él súlya 0.
  - Ha mégis definiálunk élt a nem értelmezett helyeken, akkor ott a súly legyen  $+\infty$ .
- Adjunk hozzá egy s és egy t csúcsot a gráfhoz (a definiált szín szerint).
  - Az s csúcsból vezessenek kezdetben 1 súlyú élek az összes szélső mezőkbe: a kék játékos esetén a tábla bal oldalán található mezőkbe, a piros játékos esetén a tábla tetején lévő mezőkbe.
  - A szemközti oldalon lévő csúcsokból vezessenek 0 súlyú élek a t csúcsba: a kék játékos esetén a tábla jobb oldalából indulnak az élek, a piros játékos esetén a tábla aljából indulnak az élek.

Az így definiált súlyozott, irányított, negatív élsúly mentes gráfon egy útkeresési problémát fogunk megoldani. Ahogy a következő ábrán is látni, egy állapotból előállítottuk az állapottér gráfot.

Figyeljük meg, hogy s-ből t-be a legrövidebb út a Dijkstra algoritmussal meghatározható. Az s-ből t-be vezető legrövidebb út hossza leírja, hogy a saját színű

játékos hány lépésre van a híd felépítésétől. Ha s-ből t-be nem vezet út, annak költsége  $+\infty$ , tehát veszített. Ha s-ből t-be a legrövidebb út hossza 0, akkor nyert.



10. ábra:

*útkeresési probléma a heurisztika értékének megállapításához, a kék játékos következik*

Nem szabad megfeledkezni azonban a másik játékos lépéseiről sem. A másik játékos számára is elő kell állítanunk a gráfot. A fenti példában a kék játékos következik. A kék játékos legrövidebb útja 4, a piros játékos legrövidebb útja 3. Ebben a helyzetben kell megtalálnunk azt a lépést, ami a lehető legjobbnak tűnik számunkra. Ha a kék a 4. sor 3. mezőjére lép, azzal saját maga számára csökkenti s-ből t-be vezető legrövidebb út hosszát 3-ra, de emellett a piros játékos legrövidebb útját növelte 4-re. Tehát ezzel figyelembe tudjuk venni azt is, hogy a másik játékosnak keresztbe tegyünk.

A heurisztika értéke egy előállított állapotra legyen:

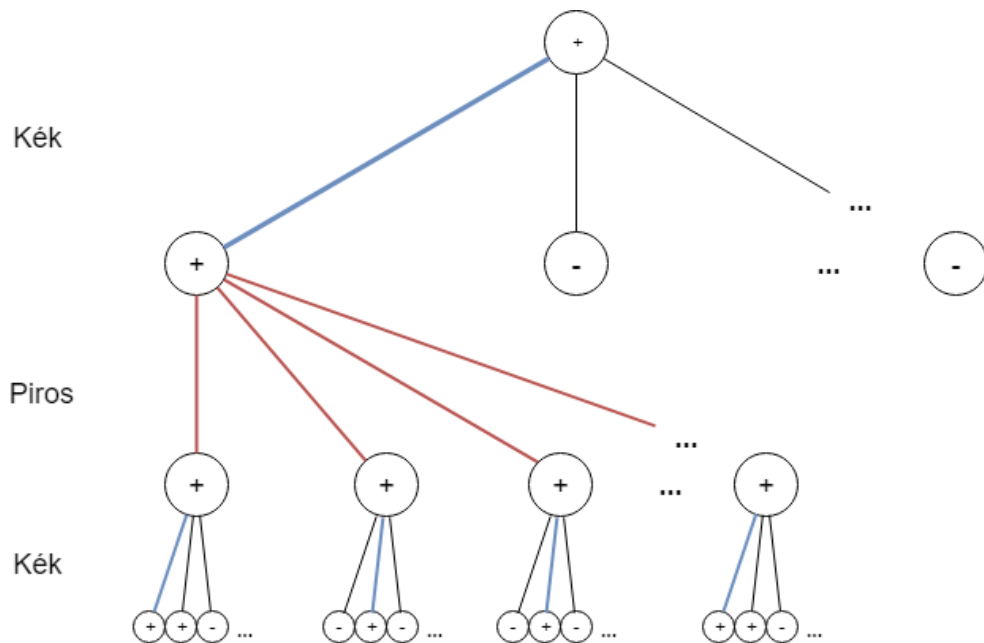
$$h(x) = \text{Legrövidebbút}(\text{játékos}) - \text{Legrövidebbút}(\text{ellenfél})$$

Minél kisebb a h függvény értéke, annál kedvezőbb helyzetbe jutunk.

## 4.2 Gráfkeresési stratégiák

Hogyan tud a játékos biztosan nyerni? Arra van szüksége, hogy minden lépésben tudjon olyan válaszlépést adni, amivel végül győzni tud. Az ellenfélnek is van beleszólása a játékmenetbe, így nem elég egy olyan utat találni a gráfban, aminek nyertes a kimenetele. Az ellenfél minden lépésével számolni kell, nem tudjuk megjósolni, hogy mit fog lépni. Ennek sok oka van, lehet, hogy más stratégiát követ, más heurisztikát használ vagy egyszerűen hibázik.

A játékos részére ez „és/vagy” fával ábrázolható. A gyökérnél, azaz a 0. szinten a játékos lehetséges döntései (állapot átmenetei) találhatóak, a páratlan szinteken az ellenfél lehetséges döntései szerepelnek. A páratlan szinteken „vagy” kapcsolat van, ezeken a szinteken elég a számunkra legmegfelelőbbet számításba venni, hiszen ezen a szinten a döntés nálunk van. A páratlan szinteken „és” kapcsolat van, ezeken a szinteken az összes lehetőséget számításba kell venni, hiszen nem tudhatjuk előre az ellenfél lépését. A nyerő stratégia az és/vagy fából állapítható meg. A nyerő stratégia egy olyan hiperút a fában, melynek minden levele nyertes pozíció. Más megfogalmazásban, a fából az ellenfél szintjén meg hagyjuk az összes élt, de a mi szintünkön töröljük a kedvezőtlen éleket, azzal a megszorítással, hogy minimum egyet hagynunk kell. Ha a levelek szintjén csak nyerő állapotok maradnak, találtunk nyerő stratégiát [12].



11. ábra:  
és/vagy fa

## 4.3 Min-max

Az algoritmus alapján szükségünk van a levelekhez egy hasznosságfüggvényre. A levelekhez a hasznosságfüggvény értékeit rendeljük. Az ellenfél szintjén minimumot veszünk a csúcs gyerekei közül, a saját szintünkön pedig maximumot veszünk a csúcs gyerekei közül, ezzel felfuttatva egy értéket a gyöker csúcsra.

Korábban már megemlítettük, hogy nagy táblaméret mellett a játédfa felépítése praktikusán nem megoldható. Ehelyett a játédfa részleges felépítését választjuk. Tehát a játékfát megválasztott  $L$  korlátig építjük fel, szükségünk van egy heurisztikus függvényre. A heurisztikus függvény legyen a következő, ahol  $-\infty$  a vereséget,  $+\infty$  pedig a nyertes játszmát reprezentálja.

$f: \text{állapot} \rightarrow [-\infty, +\infty]$  függvény.

A korábban bemutatott heurisztika a következő volt (emlékeztető):

$$h(x) = \text{Legrövidebbút(játékos)} - \text{Legrövidebbút(ellenfél)}$$

Kiértékeljük az összes levél heurisztikus értékét, majd a min-max algoritmussal felfuttatjuk az értéket a gyökérig. A következő lépést pedig annak a gyerekcsúcsnak az irányába tesszük meg, amelynek a legnagyobb a heurisztikus értéke. Ez a módszer teljesen a heurisztikus értékre támaszkodik. Tehát, ha a heurisztika téved, akkor az algoritmus is téved.

A min-max egyik változata a nega-max algoritmus, ami lényegében ugyanazt a logikát követi, mint a min-max. A különbség, hogy az ellenfél szintjén negáljuk az értékeket és mindegyik szinten maximalizálunk. Az eredmény ugyanaz, az implementáció viszont egyszerűbb lehet.

### 4.3.1 Átlagolás

Az átlagoló kiértékelés célja, hogy a tévedéseket kisimítsa. Maximalizálás esetén az  $m$  darab legnagyobb elem értékének átlagát vesszük, minimalizálás esetén az  $n$  darab legkisebb elem átlagát. Az  $m$  és  $n$  paraméterek megválasztása intuitív.

### 4.3.2 Változó mélység

A változó mélységű kiértékelés célja szintén a tévedések eltávolítása. Abban az esetben, ha a szülő és a gyerek heurisztikája közötti differencia meghalad egy  $L$  küszöbértéket, feltételezhetjük, hogy szükség van további mélység vizsgálatára. Az érték kiszámításához szükséges egy nyugalmi feltétel magadása. Használatával a kiértékelési mélység csökkenhet is és nőhet is, így, ha nyugalomba került a heurisztika, akkor megszakítjuk a rekurziót.

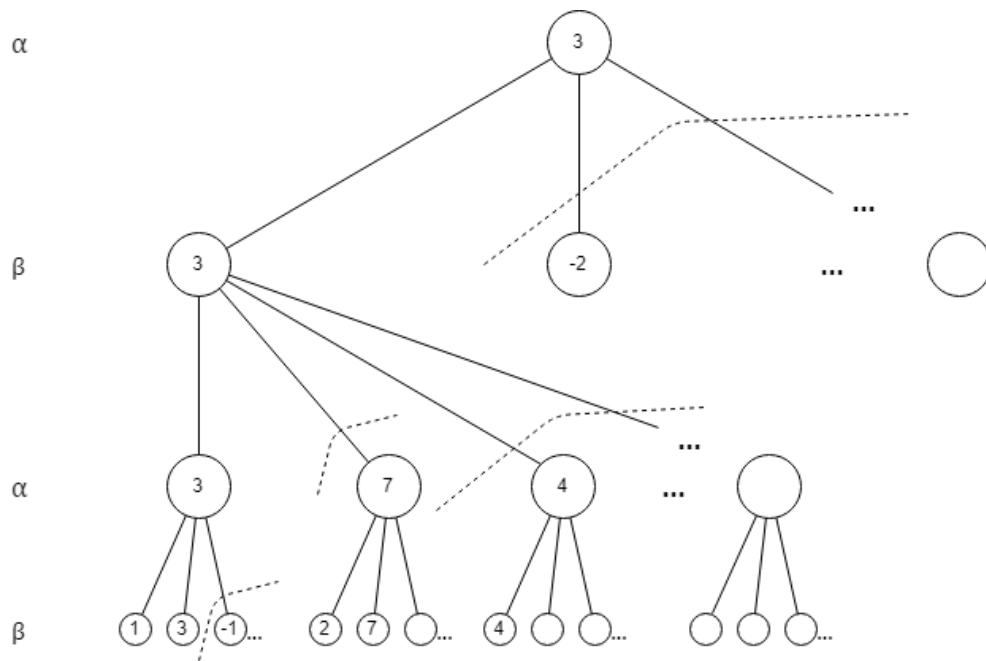
A módszerrel csökkenthető az erőforrásigény. Itt is fontos figyelembe venni, hogy a módszer a heurisztikus függvényre és a nyugalmi feltételre épül.



## 4.4 Alfa-béta algoritmus

Már említettük, hogy a játékfa mérete a mélység függvényében exponenciálisan nő. Azonban lehetséges a megvizsgált lehetőségek számát csökkenteni, ha a fát mélységi bejárás segítségével járjuk be. Ez esetben egyszerre csak egy úttal kell foglalkoznunk. Emellett megkísérlünk minden olyan részfat elhagyni, ami számunkra kedvezőtlen lehet. A módszer értelmezhetjük úgy is, mint a min-max algoritmus egy relaxációját. Az alfa-béta algoritmussal lehetőség van a min-max érték előállítására, anélkül, hogy az összes részfat végig járnánk. Ehhez feltételezzük, hogy ha egy bizonyos lépés számunkra nem előnyös, azt nem fogjuk megjátszani, ezzel párhuzamosan azt is feltételezzük, hogy az ellenfél a számára legelőnyösebb, vagy számunkra legelőnytelenebb pozíciót próbálja megjátszani. A szinteknek megfelelően ezeket az ágakat levágjuk a játékfából. Innen jön az alfa-béta vágás neve.

Az alfa-béta algoritmus alapján a játékos szintjén alfa, az ellenfél szintjén béta értékeket állítunk elő. Az alfa szinten a levelek heurisztikus értékének maximumát vesszük, így, ha a feldolgozás során találunk egy levelet, aminek biztosan kisebb lesz a heurisztikája a jelenlegi legjobb leveleknél, azt levágjuk, nem vizsgáljuk tovább. Ha találunk egy nagyobb heurisztikájú levelet, a korábbi elhagyjuk. Az béta szinten a levelek heurisztikus értékének minimumát vesszük, így, ha a feldolgozás során találunk egy levelet, aminek biztosan nagyobb lesz a heurisztikája jelenlegi legkisebb heurisztikus értékkel rendelkező levélnél, azt levágjuk, nem vizsgáljuk tovább. Ha találunk egy kisebb heurisztikájú levelet, a korábbi elhagyjuk. Az így kapott értékeket felfuttatjuk a gyökér felé.



12. ábra:  
alfa-béta vágások

Tehát az alfa-béta algoritmus esetében ugyan aztaz értéket kapjuk a gyökérben, mint a min-max algoritmussal, de a memóriaigénye alacsony, hiszen mindig csak egy utat tárol. A számítási igénye a vágások függvényében drasztikusan kevesebb lehet. Jó eredmény a részfa rendezésével érhető el.

## 4.5 Monte Carlo algoritmus

A Monte Carlo keresés, avagy Monte Carlo Tree Search (MCTS) újabb módszer a részleges játékfában történő döntés meghozatalára. Ez az algoritmus statisztikai módszerre épül és véletlenszerű (randomizált).

Az alap MCTS algoritmus jól követhető. Keresési fát építünk a játékfa csúcsaihoz, ezekből a legígéretesebbet kiterjesztjük. A kiterjesztett csúcsokra szimulációt futtatunk, ami teljes egészében véletlenszerűen előállított lépések sorozata, egészen a terminálási állapotig. A kapott értéket (végeredményt) felfelé

propagáljuk a keresési fában. Az algoritmust futtathatjuk egy mennyiségi korlátig vagy időkorlátig, eközben végig az algoritmus 4 lépését ismételjük:

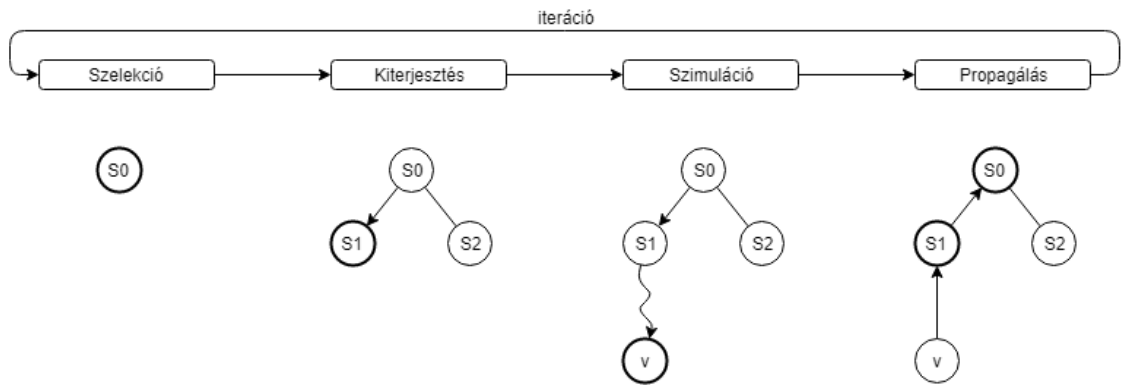
- Szelekció  
A gyökér csúcsból kiindulva kiválasztjuk a legígéretesebb gyermeket vagy gyermekeket. A mező ígéretességét egy  $V$  értékfüggvény segítségével számítjuk ki.
- Kiterjesztés  
Amennyiben a gyerek nem levél, azaz nem terminálási állapot, a kiválasztott gyereket kiterjesztjük és a gyermekeiből kiválasztunk egyet.
- Szimuláció  
Futtatjuk a szimulációt a kiválasztott csúcsra a terminálási állapotig, véletlenszerű lépésekkel.
- Propagálás  
Frissítjük a fát a kapott végeredményt felfelé propagálva.

Ha a terminálási állapot hasznosságfüggvénye nyert/vesztett, azaz 1/0 értékeket képez, akkor ezeket az értékeket propagáljuk felfelé.

Ekkor minden csúcs értéke az alattuk található nyertes állapotok és a vesztes állapotok hányadosa lesz. Tehát, ez az értéke a győzelmek aránya, másképpen fogalmazva a hasznosságok átlaga az  $i$ . csúcsra:

$$V_i = \frac{won_i}{n_i}$$

Kezdetben a kiterjesztetlen mezők értéke 0/0, amelyekre úgy tekintünk, hogy  $+\infty$ . Így a gyerekek mindenképpen kiterjesztésre kerülnek.



13. ábra:

Monte Carlo Tree Search első iteráció

Ha a terminálási állapotok hasznosságfüggvénye valamilyen intervallumra képez, például  $[0, 10]$ , akkor ezt az értéket vesszük és propagáljuk a gyöker csúcs (a kiinduló állapot) felé.

Ha a terminálási állapot hasznosságfüggvénye nyert/vesztett, azaz 1/0 értékeket képez, dönthetünk úgy, hogy többször futtatjuk a szimulációt, és visszaadhatjuk a nyertes szimulációk számát. Így 10x futtatott véletlenszerű szimulációval elérjük, hogy az egyszerű nyert/vesztett értékekkel inkább egy  $[0, 10]$  intervallumra képezzünk.

#### 4.5.1 UCB

Az UCB (Upper Confidence Bounds applied to Trees) a csúcsok értékeinek kiszámítására szolgál. Akkor használható, ha a terminálási állapotokon a hasznosság függvény egy intervallumra képez értékeket, pl.  $[-100, +100]$ . A formula kisimítja a véletlenszerűen generált értékek miatti tévedést, mint a túl sok nyertes vagy túl sok vesztes állapot:

$$UCB = \bar{V}_i + C \sqrt{\frac{\ln N}{n_i}}$$

ahol

- $\bar{V}_i$  a mező értéke, tehát a hasznosságok átlaga az  $i$ . csúcs gyerekeire.
- $C$  konstans érték. Alapértéke 2, de megválasztható.
- $N$  összes lefuttatott szimulációk száma, megegyezik az  $S_0$  csúcshoz tartozó  $n_0$  értékkel.
- $n_i$  a csúcson és alatta lefuttatott összes szimuláció száma.

Szükség van az első szintű csúcsok előkészítésére. Ezután az algoritmus a következőképpen néz ki:

---

**Algorithm 1:** MCTS iteration

---

**Result:** MAX epirical value from level 1 child nodes

add all level 1 child nodes;

**while** not limit **do**

    cursor = root;

**while** cursor is not leaf **do**

        | cursor = MAX UCB value child node;

**end**

**if** cursor is not expanded yet **then**

        | Add all possible states as child nodes;

        | cursor = first child node;

**end**

**if** cursor is a terminate state **then**

        | value = cursor value;

**else**

        | value = simulate from cursor;

**end**

    propagate value to root;

**end**

---

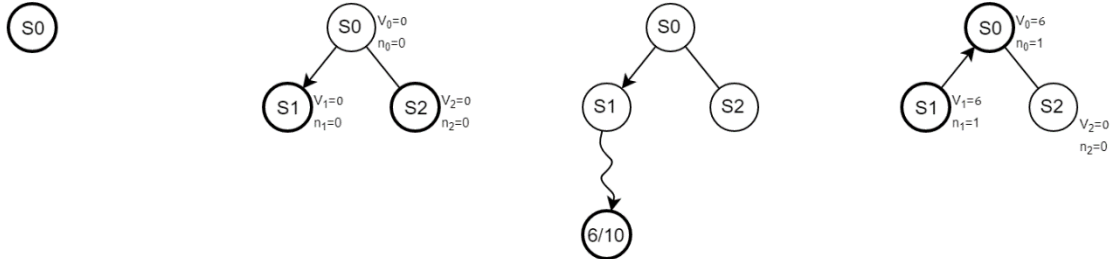
*1. algoritmus:*

*Monte Carlo Tree Search iteráció*

Egy gyakorlati példával szemléltetjük az algoritmus működését. Első lépésben az  $S_0$ , azaz a gyökér csúcshoz elő lettek készítve a lehetséges lépések, azaz a gyerekei. A gyökér csúcs előkészítését az algoritmus alapján automatikusan elvégezzük. Kezdetben kiterjesztetlen mezők értéke legyen  $+\infty$ . Így a gyerekek mindenképpen kiterjesztésre kerülnek.

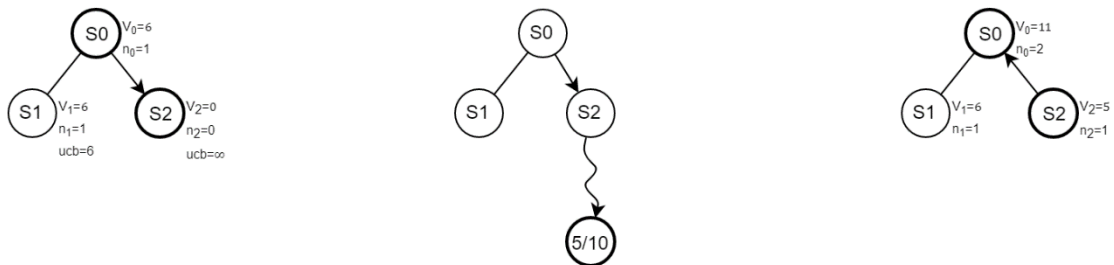
A gyökértől haladva lefelé kiválasztjuk az egyik gyereket. Mivel még egyik sem volt kiterjesztve, az algoritmus nem tud dönteni, az elsőt fogja választani. Mivel  $S_1$

még nem volt kiterjesztve (és így szimulálva sem), S1 szimulálása következik. A mi példánkban 10-ből 6 szimuláció esetén sikerült nyerni. Ezt az értéket propagáljuk a gyökér felé a V értékben, illetve propagáljuk a szimulációk számát az n értékben.



14. ábra:  
MCTS és UCB

A második iterációban indulunk a gyökér szintről a levelek felé. Az első szinten S1 és S2 közül kell választani. S1-hez már van UCB értékünk, de a kiterjesztetlen csúcs értéke  $+\infty$ , így a korábban leírtak alapján S1-et választjuk, és mivel még nem volt kiterjesztve, szimulációt futtatunk rajta. A példában 10-ből 5 alkalommal sikerült nyerni. Az értékeket propagáljuk felfelé.



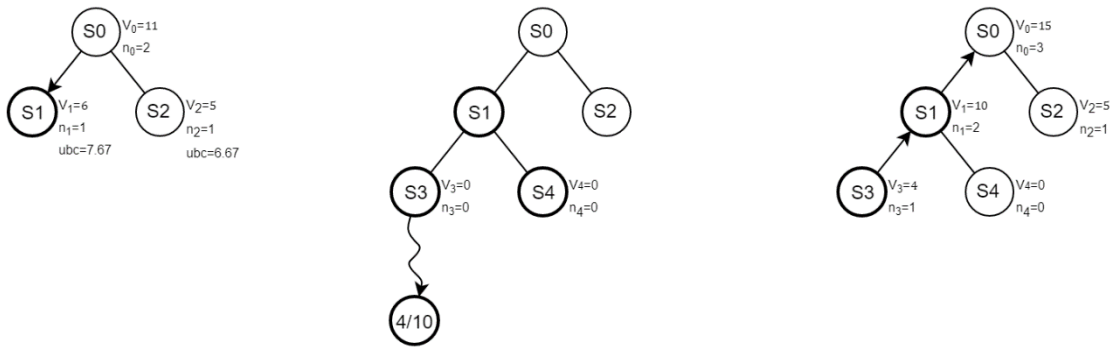
15. ábra:  
MCTS és UCB

A harmadik iterációban már S1 és S2 is rendelkezik V értékekkel, így egészen biztos ki kell számolnunk a UCB értéket.

$$UCB(S1) = \bar{V}_l + 2 * \sqrt{\frac{\ln(N)}{n_i}} = 6 + 2 * \sqrt{\frac{\ln(2)}{1}} = 7.67$$

$$UCB(S2) = \bar{V}_l + 2 * \sqrt{\frac{\ln(N)}{n_i}} = 5 + 2 * \sqrt{\frac{\ln(2)}{1}} = 6.67$$

Az S1 kerül kiterjesztésre a magasabb V érték miatt. Analóg módon folytatjuk az algoritmust. Mivel S1 még nem volt kiterjesztve, legeneráljuk a gyerekeit, majd az első csúcsra szimulációt futtatunk. S3 csúcsra a példában 10-ből 4 sikeres szimuláció volt. Az értékeket propagáljuk felfelé.



16. ábra:  
MCTS és UCB

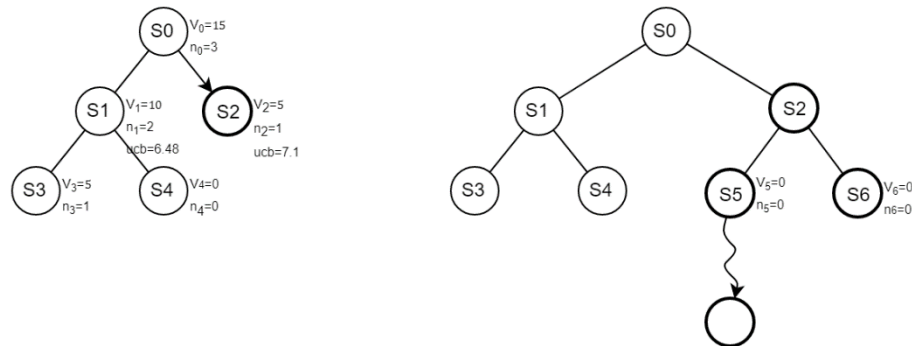
A következő iterációban ismét ki kell számítani a fában lefelé haladáshoz a gyerekek UCB értékeit.

$$UCB(S1) = \bar{V}_l + 2 * \sqrt{\frac{\ln(N)}{n_i}} = \frac{10}{2} + 2 * \sqrt{\frac{\ln(3)}{2}} = 6.48$$

$$UCB(S2) = \bar{V}_l + 2 * \sqrt{\frac{\ln(N)}{n_i}} = 5 + 2 * \sqrt{\frac{\ln(3)}{1}} = 7.1$$

A S2 kerül kiterjesztésre a magasabb V érték miatt. Analóg módon folytatjuk az algoritmust. Mivel S2 még nem volt kiterjesztve, legeneráljuk a gyerekeit, majd az

első csúcsra, azaz az S5 csúcsra szimulációt futtatunk. Az algoritmust limitáció eléréséig ismétljük.



17. ábra:  
MCTS és UCB

Tekintsük át az algoritmus tulajdonságait. Az algoritmus közelíthet a min-max algoritmus értékeihez és az alfa-béta algoritmushoz hasonlóan csökkenti a keresőfa méretét, de nem szükséges heurisztikus függvény használata. Nem szükséges továbbá semmilyen más algoritmus implementálása. Nagy mélységű kiértékelés esetén megnő a memóriahasználat. Előfordulhat, hogy a fa kiegyensúlyozatlan lesz, mert csak az ígéretes ágakra koncentrál. Előnye, hogy a futtatása könnyen megszakítható időkorláttal vagy más limitációval is, és a rendelkezésre álló adatok közül a legjobb kiválasztható. Ha egy ágot nem értékel ki, akkor a nyerő stratégia rejtve maradhat, annak ellenére, hogy egy jó humán játékos intuíció alapján megláthatja.

**Az algoritmus azon adaptációja, mely a hasznosságfüggvény helyett  $n$  szimulációt futtat és ez alapján viszi át az értéket a hasznosságfüggvényre, a dolgozat saját ötlete, illetve eredménye.**



## 4.6 Adatbázisok

Több játékhoz léteznek előre legenerált adatbázisok. A HEX-hez Computer HEX [4] projekt keretében 9x9-es táblaméretig legenerálták az összes lehetséges állapotot és nyerő stratégiát. Tehát 9x9-es méretig a kezdő játékosnak rendelkezésére áll az összes nyerő stratégia. Ha rendelkezünk ilyen adatbázissal, nincs szükségünk egyéb logikára, mint kikeresni az adatbázisból az aktuális állapotot és kiolvasni, hogy melyik mezőre helyezzük a következő korongunkat.

A 10x10-es és 11x11-es HEX tábla esetében nem áll rendelkezésünkre a nyerő stratégiák adatbázisa. Az OHex [6] projekt keretében a készítőik összegyűjtöttek lejátszott mérkőzéseket, különböző online adatbázisokat felhasználásával, melyek főleg emberi játékokból állnak (tehát nem gép volt a játékos ellenfele). Ebben az esetben is játszhatunk adatbázisból, de a végkifejlet nem garantált. Az adatbázis alapján megállapíthatjuk, hogy egy mező megjátszásával milyen százalékban nyertek a korábbi játékosok. Tehát kiolvashatjuk, hogy adott állapotból valószínűsíthetően nyeri fogunk-e vagy sem, de pontosabban fogalmazva, azt tudjuk meg, hogy a korábbi játékosok nyertek-e vagy sem.

# 5 Mesterséges neuronhálók

## 5.1 Gépi tanulás (Machine Learning)

Léteznek problémák, amit meg tudunk oldani, főleg a gyakorlatunkra és intuícióinkra támaszkodva, viszont nem tudunk rájuk explicit algoritmust adni. Ilyen többek között korunk egyik slágertémája, az önvezető autók. A részfeladatok ugyan egyszerűek, de az összetett egész rendkívül nehezen implementálható a klasszikus algoritmikus szemlélettel. Az explicit módszereknél hatékonyabb lehet a gépi tanulás. Egy kezdő működésből kiindulva próbálkozunk feladatot megoldani, ha elérünk valami, ami működik, vagy éppen, hogy nem működik, az eredmény fényében tanulunk, módosítjuk a saját működésünket. Tehát gyakorlunk. [13]

A gépi tanulást több kategóriába sorolhatjuk:

- felügyeletlen tanulás (unsupervised learning)
- felügyelt tanítás (supervised learning)
- megerősítéses tanulás (reinforcement learning)

A felügyelet nélküli tanulás semmilyen eszközt nem ad a kimenet ellenőrzés alatt tartásához. A kimenetek a bemenő adatóktól függenek. Felügyelet nélküli tanulás megoldás lehet klaszterezési problémák megoldására.

A felügyelt tanulásban adatokkal töltjük fel a neurális hálónkat és meg is mondjuk neki, hogy mi az elvárt kimenet, azaz, mi a predikció. A lehetséges kimenetek halmazát címkéknek (labels) nevezzük. Ezen belül megkülönböztetünk klasszifikációt és regressziót. Előbbi esetben egy fix étékkészletből kerül ki a kiment

érték, második esetben ez nem igaz, a kimenet egy intervallum egy pontja. **A dolgozatban bemutatott módszer is felügyelt tanítás lesz.**

A megerősítéssel tanulás sokkal több kontrollt ad a modellünk felett a felügyelt tanulással összehasonlítva. Egy rendkívül egyszerű pszichológia módszert használ fel, a jutalmazást. Fő részei a környezetnek az ágens, az akciók és az értékelés. Az ágens akciókat hajt végre, amit értékelés (jutalmazás) formájában megerősítünk. Az ágens saját hibáiból tanulhat, miközben megpróbálja maximalizálni a jutalmat.

A mesterséges neuronháló tanítása során elkövethetünk két alapvető hibát. Az egyik az alultanítás, a másik pedig a túltanítás. Előbbi túl kevés minta, utóbbi túl sok minta esetén történik meg. Ha nem mutatunk megfelelő mennyiségű vagy minőségű mintát, a hálózatunk kevéssé lesz jó predikciókat adni. Ha túl sok adatot adunk, felesleges paramétereket, amik nem relevánsak vagy nem fontosak, esetleg redundánsak a predikció szempontjából, szintén rossz predikciókat eredményezhetnek.

## 5.2 Mesterséges neurális hálózatok áttekintése

A mesterséges neuronok és az azokból épített neurális hálózatok már az 1950-es évek óta léteznek, de a számítástechnika csak 2010 környékétől képes azt a számítási teljesítményt nyújtani, amire a modelleknek szüksége van. Az új videokártyák az algebrai számítások támogatásával szintén nagy gyorsulást eredményeznek.

A gépek utasításokat követnek sorról sorra, különböző struktúrákban (pl. elágazás, ciklus), ezt imperatív programozásnak nevezzük. A programozás során ismerjük a probléma megoldását és ezt le is tudjuk írni sorról sorra. A problémák egy része nem ilyen. Lehet, hogy nem ismerjük a megoldást, vagy ismerjük, de nem tudjuk explicit leírni a megoldás menetét. Ez több okból lehetséges, pl. túl komplex a probléma, túl sok a bemenet, túl sok a kimenet. Az ilyen környezetekben válhat

jobb választássá a mesterséges neurális hálózat alkalmazása, amit az explicit működés ismerete nélkül konfigurálunk.

A Tensorflow [14] a Google mesterséges intelligencia csomagja. A Tensorflow kész megoldásokat tartalmaz mesterséges neurális hálózatok készítésére és tanítására. Csakhogy néhányat említsük, mély neurális hálózatok és konvolúciós neurális hálózatok is készíthetőek, tartalmazza a Sigmoid, ReLU aktivációs függvényeket, többfajta optimalizációs algoritmust, mint a gradiens eresztéses módszert és az ADAM optimalizációt. A Tensorflow és a Keras [15] gyakran párban jár, a Keras ugyanis egy Tensorflow magas szintű API. A Keras API elérhető C++ és Python nyelvekhez.

Érdekes a tenzor jelentésével kezdeni. A Wikipedia leírása alapján: „A tenzor egy matematikai objektum, amely a skalár és vektor fogalom általánosítása.” [16] Eredetileg a fizikában alkalmazták, deformálható anyagokban fellépő feszültségek és nyomások, azaz „tenziók” leírására.

A Tensorflow segítségével a gráfként felépített neurális hálózatunkban a gráfunk csúcspontjai a tenzoroknak felelnek meg, az élei pedig tenzióknak. (Ezt a gráfot nevezzük modellnek is.) A transzformációs gráfon tehát a tenzorok folynak végig, innen jön a tenzor folyam, a Tensorflow elnevezés.

A grafikus kártyák hardveresen támogatott matematika számításai felhasználhatóak (GPGPU) a tenzor transzformációk elvégzésére. Egy mai grafikus kártyában több ezer CUDA mag található, melyek munkába állítása nagyságrendi sebesség növekedést eredményez.

A bemenet egy 1 dimenziós tenzor, azaz vektor típusú, hasonlóan a kimenethez. A bemenet kiértékelése egy igen gyakran használt művelet. Ennek során a mesterséges neuron bemeneti értékeit beszorozzuk a súlyértékkel és ez pont az a mátrix szorzás, amit a GPU hardveresen támogat. A GPU sok párhuzamos mátrix szorzást tud elvégezni hatékonyan. Ezért lehetséges, hogy nem a CPU, hanem a GPU teljesítménye határozza meg a számítási képességeinket mesterséges

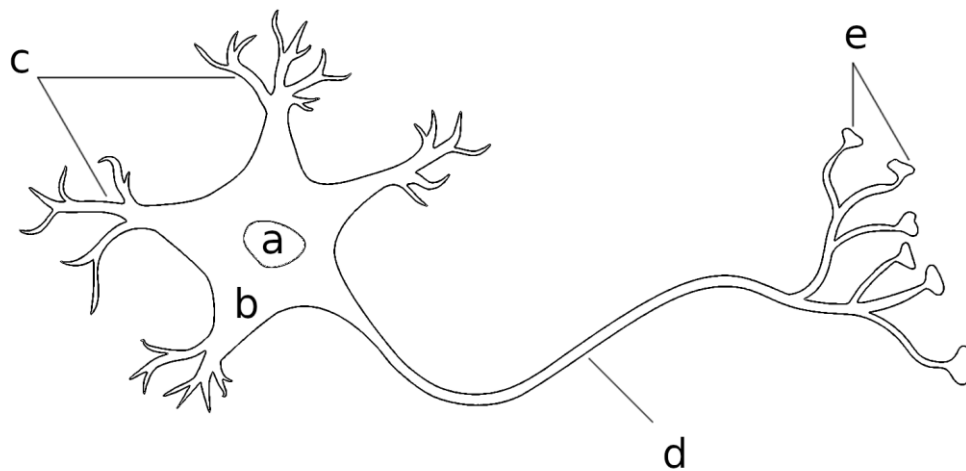
neurális hálózatok esetén. A kapott vektorra ezután alkalmazni kell az aktivációs függvényeket és folytatni a műveleteket a kimeneti rétegig.

A neurális hálózat tanítása alatt a súlyértékek beállítását értjük. Erre ma már több tanító algoritmus létezik (optimizer), így ezeket a súlyértékeket nem nekünk kell beállítani, de ez nem volt mindig így. Az első módszer, ami adott minta bemenetek és minta kimentek segítségével képes volt súlyozni a rejtett rétegek éleit is, a gradiens eresztéses módszer volt. Ennek segítségével az elvárt és a kapott kimenet összehasonlításából kapott gradienssel képes az algoritmus beállítani a súlyokat az eltérések visszaterjesztésével (backpropagation).

### **5.2.1 Biológiai neuronok**

A biológia neuron mintájára készül mesterséges neuron is. A neuronnak nevezzük az idegsejtet és nyúlványainak összességét. Habár teljes működésük még nem teljesen tisztázott, de a lényege nagyjából ismert. A mesterséges intelligencia szemszögéből nem fontos, hogy a mesterséges neuronok ugyanúgy működjenek, mint a biológiai neuronok, amennyiben feladatunkat, azaz az információtovábbítást (az elektromos impulzusok továbbítását) megfelelően ellátják.

A lenti ábrán egy neuron egyszerűsített rajza található [17]:



18. ábra:  
biológiai sejt

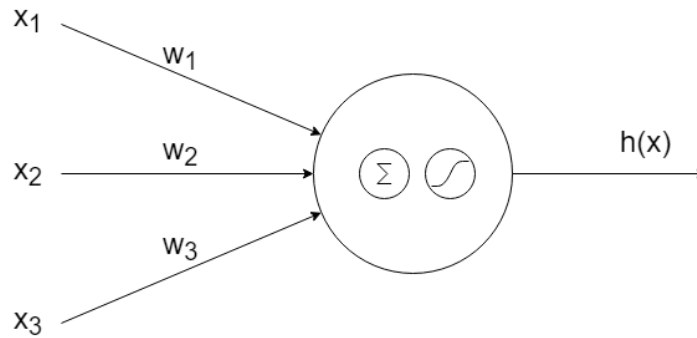
- a) Sejtmag
- b) Sejt
- c) Dendrit
- d) Axon nyúlvány
- e) Végfa

### 5.2.2 Mesterséges neuron

A mesterséges neuron leképezése egy valódinak. A dendrit az input oldalnak felelnek meg, míg az axon nyúlvány és a végfa a kimenteknek (terminal branches). A végfa nyúlványai dendritekhez kapcsolódnak, így alkotva kapcsolatot a neuronok között. Ezeket a kapcsolatokat szinapszisznak nevezzük.

A mesterséges neuron először megvizsgálja a bemeneteket. A bemeneti függvény számításokat végez, ez alapján az aktivációs függvény meghatározza az értéket a végfa irányába. Az aktivációs függvény lehet küszöbérték függvény, ami egy érték elérése esetén 1-et, alatta pedig 0-át ad vissza. Ilyen például az

előjelfüggvény is. Az aktivációs függvény ezen kívül lehet logisztikus függvény is. Ilyen például a Sigmoid függvény, ami monoton növekvő, folytonos, differenciálható és konvergens (ez később fontos szempont lesz). A kimeneti érték több másik bemenetre is csatlakozhat. Az egyenletben a szabadsági fokot a  $\bar{w}$  súlyvektor adja. Tehát a neuron tanításán a  $w_i$  súlyértékek beállítását értjük.



19. ábra  
mesterséges neuron

A bemeneti adatokat  $\bar{x}$ -szel jelöljük:

$$\bar{x} = (x_1, x_2, x_3, \dots, x_n)$$

A bemenetekhez tartozó súlyokat  $\bar{w}$ -vel jelöljük:

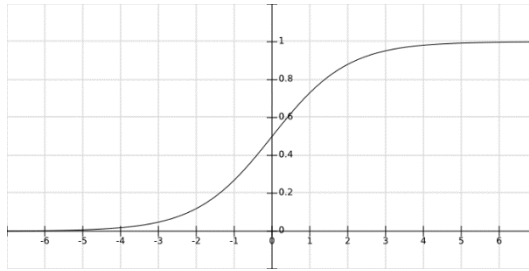
$$\bar{w} = (w_1, w_2, w_3, \dots, w_n)$$

A bemeneti függvény a következő:

$$f_w(x) = \sum_{i=1}^n x_i w_i$$

Aktivációs függvénynek válasszuk a Sigmoid függvényt [18]:

$$g(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{1 + e^x}$$



20. ábra

*Sigmoid függvény*

A  $h(x)$  függvény pedig hipotézisfüggvény. Megadja, hogy egy adott bemenetre a mesterséges neuron mit ad vissza:

$$h(x) = g(f_w(x))$$

A logisztikai függvények általában 0 és 1 közé képeznek. Célszerű megadni, hogy milyen érték felett és alatt kerekítjük 0-ra és 1-re a kiszámított értéket.

$x_1$ -hez tartozó súlyértéket,  $w_1$ -et inicializáljuk és rögzítjük -1 értékre. Később megmagyarázzuk, ennek mi a jelentősége.

### 5.2.3 Neuronok hálózata

Az előző fejezetben képet kaptunk nagy vonalakban a mesterséges neuronok működéséről. Ha a neuronokat összekötjük a szinapszisokkal, akár kis, akár nagy mennyiségben, akkor neurális hálózatról beszélünk. A neuronok hálózatát irányított gráfként reprezentálhatjuk. A neurális hálózatoknak két fő csoportja van:

- Előrecsatolt
- Visszacsatolt

Az előrecsatolt hálók esetében nem találunk hurkokat a gráfban. Kimenete közvetlenül a bemeneti értékektől függ, nincs belső állapota, csak a súlyok. Ellentétben a visszacsatolt hálózatok esetében hurok találhatóak a gráfban, tehát egy neuron kimenete egy korábbira kapcsolódik. Így a hálózat dinamikusan változik,

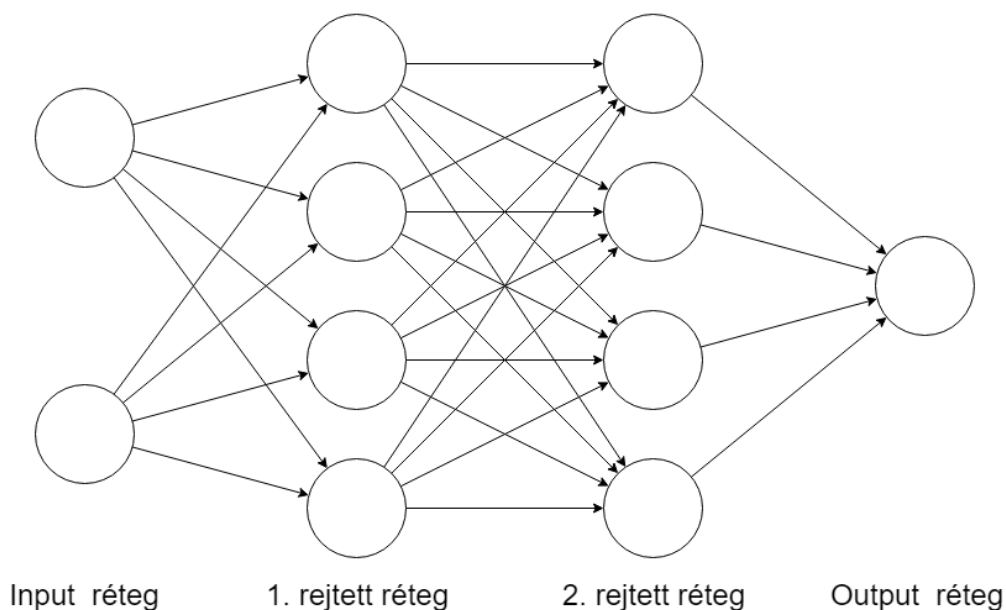


elérhet egy stabil, nyugalmi állapotot, de akár folyamatos változásban is maradhat.

**A programban előrecsatolt hálózatot használtam.**

#### 5.2.4 Előrecsatolt hálózatok

A mesterséges neuronokból tehát gráfot építünk. A hálózat mélységén a bemenet és a kimenet közötti út hosszát értjük. A bemeneti réteget input rétegnek (input layer), az azt követő csúcsokat 1. rétegnek (layer 1 vagy dense 1) nevezzük. A legutolsó réteget, ahonnan a kimeneti értéket leolvassuk, kimeneti rétegnek (output layer) nevezzük. A kimeneti és bemeneti rétegek közötti rétegeket rejtett rétegeknek (hidden layer vagy dense  $n$ ) nevezzük. Kérdés, hogy vajon hány rétegtől nevezzük a neurális hálózatunkat mély neurális hálózatnak (deep neural network)? A kérdés érdekes, de abban biztosak lehetünk, hogy 10 már nagyon mély. A több réteg előnye, hogy kiterjeszti a hipotézisfüggvény által leképezhető értékeket, tehát bonyolultabb logikák leírására képes.



21. ábra

*Teljesen kapcsolt mesterséges neurális hálózat*

A bemeneti réteg felépítése a bemenet jellegétől függ, ez egy az egyben továbbítja azokat, módosítás nélkül.

A rejtett rétegek tehát a bemeneti és a kimeneti rétegek között helyezkednek el, feladatuk a logikai műveletek elvégzése, adatok transzformációja, logikai kapcsolatok felépítése. Számuk és kapcsolatuk választható paraméterei a hálózatnak.

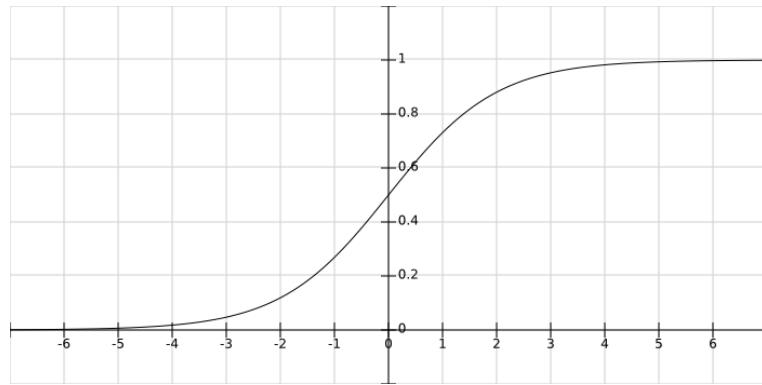
A kimeneti réteg felépítését a feladat határozza meg, tehát, hogy milyen kimenetet várunk a hálózattól. Regressziós (regression) probléma esetén ez egyetlen egy érték lesz. Osztályozás (classification) esetén annyi kimenet lesz, ahány lehetséges kategóriánk van. A kimeneti értékek pedig az adott kategóriába tartozás valószínűségét reprezentálják. **A program regressziós kimenetet használ.**

## 5.2.5 Aktivációs függvények

Az előző ábrán szemléltettük, ahogy a neuronok a többi neuronokhoz kapcsolódnak, értékeket továbbítanak a mélyebb rétegeknek és az output-nak. Az aktivációs függvények az értékek hálózaton belüli továbbításában, illetve transzformációjában töltenek be fontos szerepet. A transzformáció során olyan értékeket állítunk elő, amik megfelelő intervallumon kell, hogy elhelyezkedjenek, tehát transzformáljuk őket. A következő részben ismertetjük a 3 legelterjedtebb függvényt, de elméletben végtelen sokféle konstruálható.

### 5.2.5.1 Sigmoid

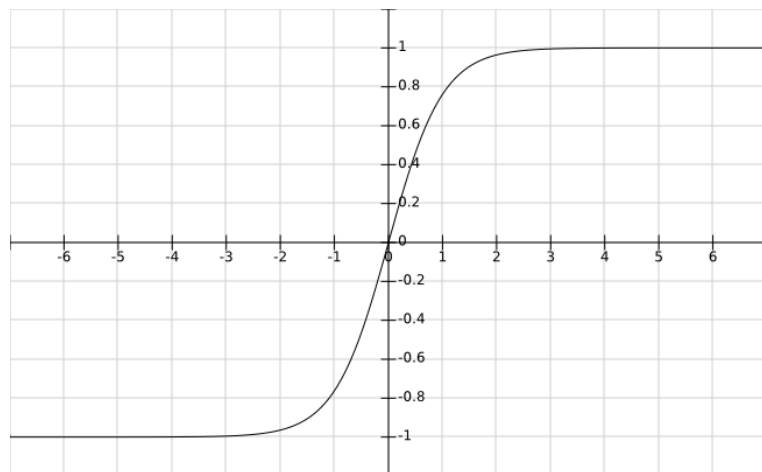
A Sigmoid függvényt főleg az output rétegben szokták alkalmazni.



$$s(x) = \frac{1}{1 + e^{-x}}$$

### 5.2.5.2 Tangens hiperbolikus

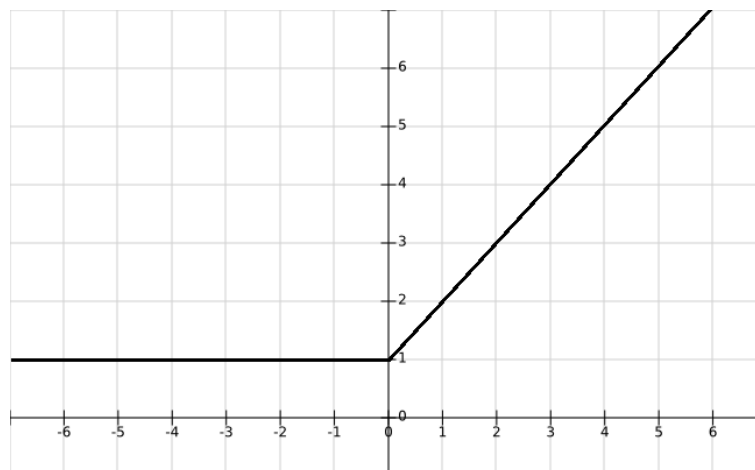
A TanH hasonlít a Sigmoid függvényre, de 0 környezetében van a középpontja.



$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

### 5.2.5.3 ReLU

A ReLU főleg a rejtett rétegekben bizonyul jól használhatónak. A „rectified linear unit” rövidítése (helyesbített lineáris függvény).



$$ReLU(x) = \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases}$$

### 5.2.6 Optimalizáció és veszteségfüggvény

A korábbi fejezetekben megállapítottuk, hogy a mesterséges neurális hálózat a mesterséges neuronok gráf alapú modellbe rendezését jelenti. Ezen túl ismertetésre került a neuronok értékfüggvénye és aktivációs függvénye.

**A hálózati modell tanítása a véletlenszerűen beállított súlyértékek optimalizálását jelenti.** Tanításkor a tanuló adatbázisból általában nem egy mintát, hanem annak egy részét adjuk át. A tanító algoritmusok alapgondolata, hogy előállítjuk a mintára a kimenetet, utána megvizsgáljuk, hogy mennyivel tér el a kívánt értéktől, és annak mértékben eltoljuk a súlyokat. A súlyeltolás érték bias néven is ismert. Ezt több iterációban ismételve taníthatjuk a neurális hálózatot, aminek folyamán a kimenet konvergálni fog az elvárt kimenethez. A tanítási iterációkat eposzoknak (epocs) nevezzük. Az iterációk száma megadható. A tanítás során tehát szükségünk van egy tanító algoritmusra (optimizer) és egy veszteség függvényre (hiba függvény, avagy loss function).

### 5.2.6.1 Veszteségfüggvények

Választanunk kell egy olyan függvényt, mi jól reprezentálja a hálózat hibáját, tehát a különbséget a kapott kimenet és az elvárt kimenet között. Gyakorlatilag végtelen számú veszteségfüggvény konstruálható, de áttekintjük a legelterjedtebbeket.

- Átlagos négyzetes eltérés (mean squared error): Sigmoid kimeneti aktivációs függvény és lineáris aktivációs függvénnyel használják regressziós problémákhoz. A képletben  $Y$  az elvárt kimenet (expected value),  $O$  pedig az aktuális kimenet (actual value).

$$C = \frac{1}{2n} \|O - Y\|^2$$

- Bináris keresztentropia (binary crossentropy): Sigmoid kimeneti aktivációs függvénnyel használható leginkább két értékű osztályozási problémákhoz.

$$C = -\frac{1}{n} \sum_{i=0}^n \|Y \log(O) + (1 - Y) \log(1 - O)\|^2$$

- Többkategóriás keresztentropia (multiclass crossentropy): az információelméletben két bináris közötti információveszteség mértékét adja meg.

$$C = -\sum_{i=0}^n Y_i \log(O)$$

### 5.2.6.2 Optimalizációs algoritmusok és hibavisszaterjesztés

A mesterséges intelligenciában korszakváltást jelentett a gradiens ereszkedésen alapuló hiba-visszajelzés. Ez lehetővé teszi a rejtett rétegekben is a neuronok tanítását, amennyiben az aktivációs függvényük differenciálható. Az ötlet alapján a kimeneti értékekből előállított függvényt és az elvárt kimenetből előállított

függvényt kivonjuk egymásból és annak vesszük a gradiens-ét parciális deriválással. Ezt a gradiens vektort próbáljuk minimalizálni. Több módszer is készült a gradiens ereszkedésen kívül, de szinte teljesen egyeduralkodóvá vált az ADAM optimalizáció. Az algoritmusokat, melyek veszteségfüggvényt csökkentik, optimalizációnak nevezzük, függetlenül attól, hogy gradiens ereszkedést használnak vagy mást. Tekintsük át az optimalizációkat:

- Gradiens ereszkedés módszer
- Lendület módszer
- Nesterov módszer
- Adaptív gradiens
- RMSprop módszer
- Adaptív lendület módszer (ADAM)

Az összes optimalizálás esetében a mintaadatbázist szeletenként adjuk át az optimalizálónak (batch). Az optimalizáló a bemenetre teszi a mintákat és kiszámítja a kimeneteket. A kapott kimeneteket összehasonlítja az elvárt kimenetekkel. A cél tehát a két kimenet által reprezentált függvények különbségének csökkentése. Ezt a műveletet ismételjük több alkalommal. Ismétléssel a kimeneti érték és az elvárt érték közelíteni fog.

## 5.3 Tanulási adatbázis

A felügyelt tanulás (supervised learning) megkezdéséhez szükségünk van egy adatbázisra, ami alapján a neurális hálózatot be tudjuk tanítani. Az adatbázisban található minták alapján fog működni a hálózat. A mi esetünkben ez korábban lejátszott játékok sokasága. Szerencsére, a Computer Hex [4] program alapján a NeuroHex [5] projektben előállítottak már adatbázist, ami tartalmaz nagyságrendileg 11 000 játékot és 551 000 állapotot. Az adatbázis tehát 11 ezer sorból áll, amiben mezőpozíciók találhatóak szóközzel elválasztva. A mi

viszonyításunk mindig az lesz, hogy az első játékos nyert-e. A dolgozatban mindig a kék az első játékos.

Másik fontos paraméter annak megállapítása, hogy pontosan hány lépésből jutottunk a végállapotba. Ez a játék hossza. Ugyanis más jellegű lépések szükségesek a játék elején, a közepén és a végén. A teljes lépésszám ismeretében tudunk következtetni a játék szakaszára: eleje, közepe és végjáték. Ha 45 lépésből áll a játék, akkor 15 és 30 lépésnél húzzuk meg a határokat.

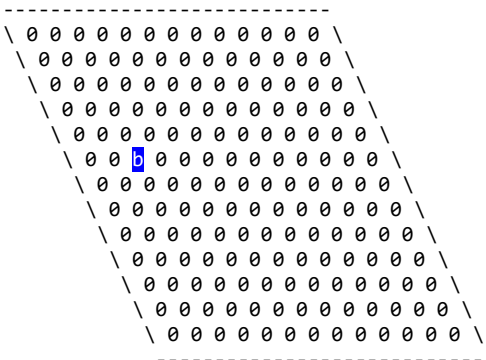
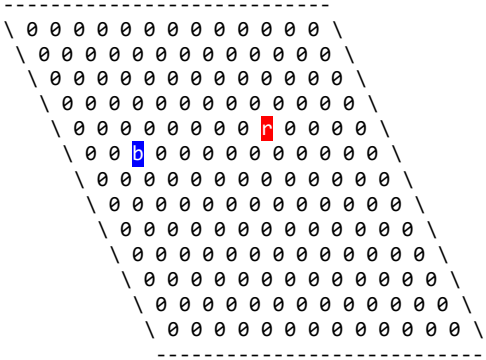
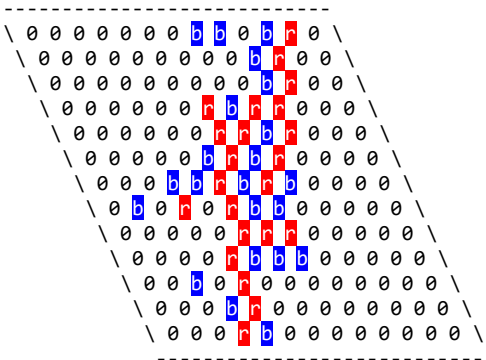
A tanítás megkezdése előtt az adatbázist két részre osztjuk. Az adatbázis 80%-át betanulásra használjuk, a fennmaradó 20%-át ellenőrzésre.

## 5.4 Adat reprezentáció

Először az összes játékhoz előállítjuk a lépések számát és hogy a kezdő játékosnak sikerült-e nyernie. Ezt követően előállítjuk az összes állapotot, melléteszünk még két információt: a játék éppen melyik szakaszban jár, illetve, hogy a játék során az első játékosnak sikerült-e nyernie. Vegyük a következő példát, ami a lépések (koronglerakások) sorozata a táblán, természetesen váltakozó színnel.

*f3 e9 h7 h8 b12 b11 i5 i7 f9 d12 e11 e10 g10 f10 k5 k6 j7 i9 l7  
e8 j9 j6 d8 l8 g9 i2 f7 g12 c10 e7 d6 c11 h4 f5 f12 h5 i4 g8 f8  
g5 h6 h1 j2 j3 i3 f11 g11 g6 g7 j1 k1 f13 e13*

A fájl beolvasásához és az állapotok elkészítéséhez a **numpy** python könyvtárat használtam fel. A numpy többek között többdimenziós tömbök nagyon gyors kezelésére használható.

Bemenet			Kimenet
Állapottér	Lépés	Szakasz	
	1	0/2	negative
	2	0/2	negative
<p>.</p> <p>.</p> <p>.</p>	<p>.</p> <p>.</p> <p>.</p>	<p>.</p> <p>.</p> <p>.</p>	<p>.</p> <p>.</p> <p>.</p>
	53	2/2	negative

2. táblázat:  
reprezentáció



## 5.5 Neurális hálózat tanítása

A hálózat elkészítéséhez megfelelő formába kell hozni az adatokat, azaz előfeldolgozás szükséges. Ehhez tudnunk kell, hogy a neurális hálózat milyen formában fogja várni az adatokat. A **Keras API**-t fogjuk használni, ami egy Python magas szintű API a **Tensorflow** számára. A hálózat bitek formájában várja az inputokat, így ez lesz a célforma. Ennek eléréséhez először minden lehetséges bemeneti értéket címkékké (labels) kell konvertálni. Ehhez a **scikit-learn machine learning** python könyvtárt fogjuk felhasználni, amit teljesen kompatibilis a numpy tömbökkel.

A bemeneti értékek címkékké való alakítását a **LabelEncoder** fogja elvégezni. A LabelEncoder egy adathalmazban található értékeket numerikus címkékké alakítja. Habár, ha külön nem nyilatkozunk róla, hogy melyik numerikus címkehez milyen értéket rendeljen, akkor is automatikusan megállapítja. Az első értékhez 0. sorszámú címkét, a második értékhez 1. sorszámú címkét rendeli és így tovább, de ezt nem szükséges igénybe venni.

Mi explicit megadjuk neki a párosítást. Ehhez először kettéválasztjuk az bemeneti és kimeneti értékeket 2 kategóriára és külön fogjuk használni LabelEncoder-t. Kezdjük a bemeneti értékek kódolásával. Az üres mezőhöz rendelje a 0. sorszámú címkét, a kék korongokhoz 1. sorszámú címkét, a piros koronghoz pedig a 2. sorszámú címkét. Bevezetünk egy szakaszváltozót is, ami leírja, hogy a játék melyik fázisban tart. (A szakaszváltozók egy az egyben hozzárendelhetők az állapottér eddigi címkéihez: 0 – játék eleje, 1 – játék közepe, 2 – végjáték.) A kimeneti értékek kódolását egy másik LabelEncoder végzi. A 'negative' értékhez az 0. számú címkét rendeljük, a 'positive' értékhez az 1. számú címként.

Miután az összes értéket kicseréltük numerikus címkékre, nekiláthatunk az utolsó lépésnek. Mivel a neurális hálózat bitekből ért, bit-ek szintjén kell reprezentálnunk a bemeneti és kimeneti paramétereinket. Az **OneHotEncoder** minden lehetséges bemeneti attribútumra létrehoz megfelelő számú bitet. A bitek

csoportja reprezentálnak egy címkét. Tehát, a táblán lévő az A1-es mező értéke lehet üres, kék és piros, így a mezőt három bittel reprezentálja.

Figyeljük meg, hogy a bemeneti értékek között csak 3 címke szerepelhet, ennek reprezentálásához elég lenne összesen kettő bit. Ha egy mező nem kék és nem piros, abból következik, hogy üres. A bemenet így redundáns információkat tartalmaz, ezért, a három bitből az elsőt el kell távolítani. De igazából választhatjuk a második vagy harmadik bitet is, nem befolyásol semmit, a lényeg, hogy minden harmadik bitet távolítsuk el a bemenetről.

Ez alapján az állapottér kódolása így képzelhető el:

Bemenet												Kimenet					
Állapottér										Szakasz							
0		b		r		...				1		positive					
0		1		2		...				1		1					
1	0	0	0	1	0	0	0	1	...			0	1	0	1		
0		0		1		0		1		...			1		0		1

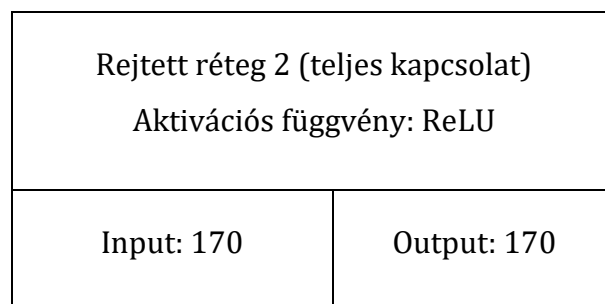
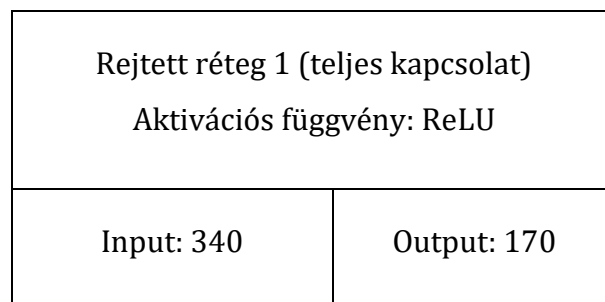
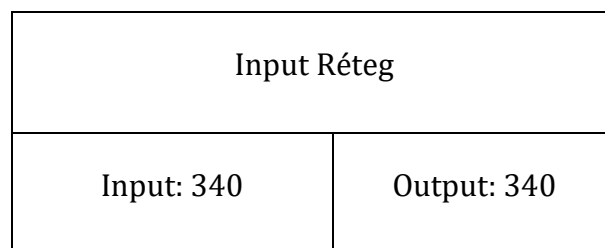
3. táblázat:

*LabelEncoder, OneHotEncoder és redundancia eltávolítás*

Így könnyen kiszámolhatjuk, hogy 13x13 táblaméret esetén 169 mezőnk, plusz egy szakaszváltozó paraméterünk van, az összesen 170 mezőt jelent, amit egyenként 3 bit reprezentál a OneHotEncoder használata után. A kapott 510 bitből minden harmadikat elhagyjuk, a bemeneti bitek száma tehát 340.

A regresszió mentén egy  $[0, 1]$  közötti számot fogunk kapni a kimenetre, ahol a 0 reprezentálja a negatív, tehát az első játékos számára kedvezőtlen állapotot, az 1 pedig a pozitív, azaz kedvező állapotot. Az aktuális játékalapból megvizsgáljuk a közvetlenül elérhető állapotokat, vagy bizonyos mélységben játékfát építhetünk és az állapotokra egyenként predikciót kérünk a neurális hálózattól.

## 5.6 Rétegek



Output layer (teljes kapcsolat) Aktivációs függvény: Sigmoid Algoritmus: ADAM Hibafüggvény: Binary crossentropy	
Input: 170	Output: 1

## 5.7 Predikció

Az adatbázis 20%-át meghagytuk tesztelésre. Következő lépésben a betanított és lementett modellt betöltjük, a teszt adatbázisból sorról sorra vesszük ki az állapotokat és transzformáljuk a háló számára megfelelő bementi formába. Minden sorra egy  $[0, 1]$  intervallum közötti számot fogunk kapni. A kapott értékeket összehasonlítjuk a tesztadatbázisban található értékekkel. Mivel a regresszió miatt nem diszkrét (0 vagy 1) értékeket kapunk vissza, így kerekítésre lesz szükségünk. Egészekre kerekítjük a predikciós értékeket és megvizsgáljuk, hogy megegyezik-e az általunk várt értékkel. Az elvártnak megfelelő teszteseteket elosztva az összes tesztesettel egy becslést kaphatunk, hogy a tanítás megfelelő-e. **A feni módszer esetében 97.51%-ot sikerült elérni, így a tanítás megfelelőnek ítéljük.** A neurális hálózat készen áll a grafikus alkalmazással való integrációra.

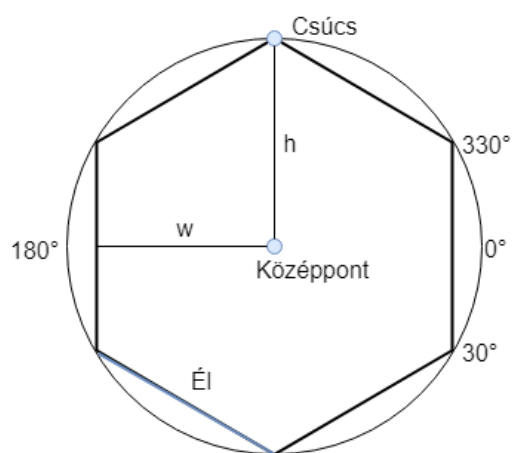
## 6 Programozási áttekintés

Ebben a fejezetben rövid kitekintéseket fogok tenni az implementáció érdekesebb részleteire. Forráskódot, osztálydiagrammokat és osztályleírásokat nem tartalmaz ez a szekció. Az itt kiválasztott témák inkább diszkrét témakörök mintsem teljes leírások.

### 6.1 Felület rajzolása

A felhasználói felület megvalósításánál szempont volt, hogy matematikailag szép megoldás készüljön el, és a felület kirajzolása páraméretezhető legyen. A felület vászonra (canvas) van rajzolva. A felület generálása a hatszögek csúcsainak geometriai kiszámításával, a szomszédos csúcsok összevonásával készül, a rámutatás pedig normál érték segítségével történik. Ennek köszönhetően nem csak a tábla mérete, hanem a hatszögek mérete is dinamikusan beállítható.

Egy hexagon lehet csúcsra állított, vagy élére állított. Mi az csúcsra állított változatot fogjuk megmutatni. Az élére állított geometriából egy egyszerű  $30^\circ$ -os elfogatással kapható meg a csúcsára állított változat, ilyenkor a  $h$  és  $w$  értékek is felcserélődnek. A hexagon mérete legyen  $S$  (mint size), ami a hexagon köré írható kör sugarát jelenti, de fogalmazhatunk úgy is, hogy a középpont és a csúcsok távolsága. A



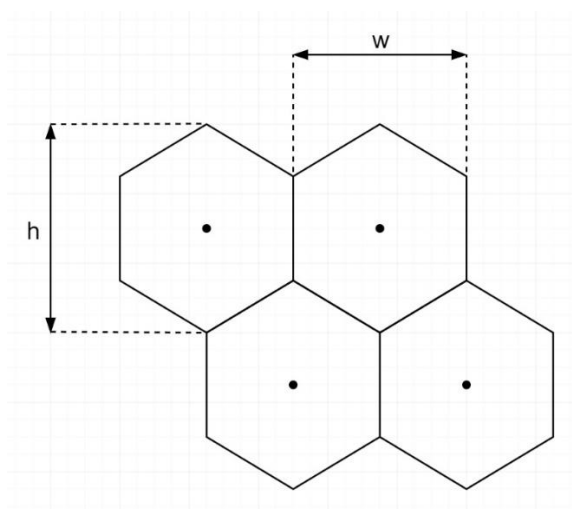
22. ábra

hexagon geometria

középpontból a csúcsok felé vezető szakaszok által bezárt szög a következőképpen alakul lapjára fordított hexagon esetében:  $0^\circ, 60^\circ, 180^\circ, 240^\circ, 300^\circ, 360^\circ$ , az említett  $30^\circ$ -os forgatással a csúcsára állított változat esetében:  $30^\circ, 90^\circ, 150^\circ, 210^\circ, 270^\circ, 330^\circ$ .

Az csúcsra állított geometria esetén  $w$  és  $h$  értékei a következőképpen számíthatóak ki:

$$h = 2 * S, \quad w = \sqrt{3} * S$$



23. ábra  
tábla geometria

A tábla kirajzolása előtt megadjuk, hogy az  $[1,1]$  indexű hatszög (a bal felső) középpontja hol helyezkedjen el. Ezután a többi koordinátát ehhez viszonyítva számítjuk ki. Egy hatszög jobb oldali szomszédjának középpont koordinátája a következő képlettel határozható meg:

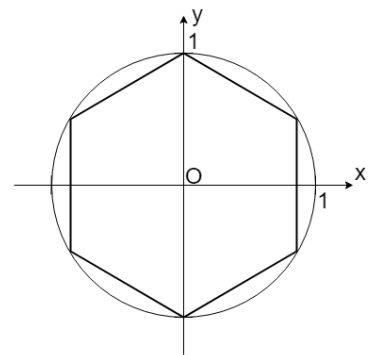
$$x_{i,j+1} = x_{i,j} + w, \quad y_{i,j+1} = y_{i,j}$$

Egy hatszög lenti szomszédjának (ami igazából lent és jobbra helyezkedik el) középpont koordinátája a következő képlettel határozható meg:

$$x_{i+1,j} = x_{i,j} + \frac{1}{2}w, \quad y_{i+1,j} = y_{i,j} + \frac{3}{4}h$$

A tábla kirajzolása után következő lépés a lebegőpontos hibák eltávolítása. Két hexagon hiába szomszédos, mivel a középpont koordinátájuk különböző, és a csúcs helyének kiszámításához szükséges szögfüggvények használata, gyökvonás és osztás miatt előfordul, hogy kerekítés után nem ugyanarra a pixelre esnek, emiatt a kirajzolt tábla pontatlan lesz. Mint ami kicsit szétcszúzott, ha lehet így fogalmazni. Ezt a problémát a „csúcsok lopásával” oldottam meg. Amikor egy új hexagont kirajzolunk, a szomszédos csúcsokat nem számítjuk ki, hanem koordinátáit átmásoljuk a szomszédtól.

Kirajzolás után nem kattintható objektumokat kapunk, mivel az egész játéktábla egy vászonra lett felrajzolva. Valahogy meg kell állapítani, hogy az egér kurzor melyik hexagon felett van. Ehhez a kurzor koordinátáit, a hexagon középpontjának koordinátáit, a hexagon körülírt körének sugarát és mátrixnormát fogunk felhasználni. Nevezzük most ezt a normát **H normának**.



24. ábra  
vektornorma

$$\|V\|_H = \max \left\{ |x|, \frac{x + y\sqrt{3}}{2}, \frac{x - y\sqrt{3}}{2} \right\}$$

Ahhoz, hogy eldönthessük, hogy a kattintás (illetve annak x és y koordinátái) benne van-e egy adott hexagonban, a következőket végezzük el:

1. Megadjuk az egységvektor mértékét. Egységnek a hexagonok köré írható kör sugarát definiáljuk.
2. A hexagon középpontjából a kattintás pontjába előállítunk egy  $V$  vektort. Ezt az abszolút koordináták kivonásával kapjuk meg.
3. Kiszámítjuk az 1. pontban előállított  $V$  vektor hosszát az euklideszi vektornorma használatával. Ez a normalizáláshoz (4) és az eldöntéshez (6) is kell. Ez az érték reprezentálja, hogy milyen távolságban van a kattintásunk egy hexagon középponttól.

4. Normalizáljuk a hossz érték (3) segítségével a 2. pontban előállított V vektort. Így megkaptuk az N vektort, ami a kattintott pont a hexagon középpontjától megadja a kattintás irányát. Ez a vektor egységnyi (általunk definiált egység) hosszúságú. Tehát a hexagon középpontjából a köré írható kör egy pontjára mutat.
5. A (4) pontban előállított N vektornak vesszük a H normáját. Ez az érték reprezentálja, hogy a kattintás irányába milyen távolságban van a hexagon határvonala.
6. A 3. pontban előállított hosszúságot összehasonlítjuk az 5. pontban előállított hosszúsággal. Ha  $(3) < (5)$ , akkor a kurzor a hexagon felett van. Tehát, ha a kattintott pont távolsága kevesebb mint a hexagon oldalának távolsága.

## 6.2 Többszálás programozás

Jelenleg az egyszerű processzorok is egy fizikai maggal és azon 2 szállal, de inkább több fizikai maggal, magonként 2 szállal rendelkeznek. Tehát valószínűleg 4, de legalább 2 szálon bármelyik géppel dolgozhatunk.

Az alfa-béta algoritmus párhuzamosítása egy huszárvágással megoldható. Elegendő szál esetén akár mindegyik szabad mezőre indíthatunk egy gráfkeresést. Amikor találunk egy nyerő stratégiát egy mezőre, akkor a többi szálát le is állíthatjuk. Amennyiben a vizsgált mezőn nem találunk nyerő stratégiát, akkor a felszabaduló szálát a következő szabad mező kiértékelésére használjuk tovább. Ez a megoldás egyszerűen implementálható, de nem veszi figyelembe a részfák méretét. Akkor is egy szál fog dolgozni egy adott mezőn, ha kicsi a részfa mérete, és akkor is, ha el kell menni a legmélyebb szintekig.

A Qt keretrendszerben a QThreadPool használható erre a célra. A QThreadPool QThread leszármazott osztályokat tud futtatni. Az állapottér dekompozíciója során egy mezőből egy QThread lesz, ami bekerül a QThreadPool-ba. A szálak



végrehajtását a rendelkezésre álló szálakkal a QThreadPool automatikusa végzi. Gondoskodni kell a szálak kilépési feltételeiről, hiszen, amennyiben egyik szál talál egy nyerő stratégiát, a többi szálnak megszakítási szignált kell küldeni, aminek hatására a többi szálnak terminálnia kell. A szignál által vezérelt változó állapotát folyamatosan figyelni kell. A módszer hátránya tehát, hogy egy mező vizsgálata nem párhuzamosított. Ebből következik az előnye is, hogy több mező kiértékelése fut párhuzamosan.

Lehetőségünk van mélységi korlát, vagy időkorlát bevezetésére, így a korábban említett effektus, miszerint egy szál túl sok ideig futhat, eltűnik. Egy min-max algoritmusnál megfelelő módszer lehet, hogy korlátot szabjunk a kiértékelés mélységének. Természetesen lehetőség van a részfákat tetszőlegesen bepakolni a ThreadPool-ba, de ha ezt a logikát rosszul választjuk meg, előfordulhat, hogy egyik szál a másikra vár, ugyanis szükség lehet az összes adat megismerésére a heurisztikus érték propagáláshoz. Jó választás lehet minden ezredik csúcs új szálon történő számítása. Mivel adatokat propagálunk felfelé a gráfban, ez megköveteli közös, konkurens gráfmodell kezelését a szálak között.

Ha alfa-béta és min-max kombinációt használunk, akkor szükséges lehet a korábban elindított szálak és azok által a ThreadPool-ba helyezett feladatok terminálása. Hiszen, ha alfa szinten találunk egy nyerő ágat, a többit azon a szinten felesleges kiértékelni, az alfa-béta vágás áldozatai lesznek.

A Monte Carlo módszer használatánál (ami szintén egy változó mélységű kiértékelés) a párhuzamosítás már egy komplexebb kérdés, hiszen nem látjuk előre, melyik gráf csúcsot milyen mélységben kell kiterjeszteni. Tehát változó mélységű kiértékelésnél kell egy ötlet. Itt is adatokat propagálunk felfelé a gráfban, ami megköveteli közös, konkurens gráfmodell kezelését a szálak között. A Monte Carlo módszer esetében máshogy érdemes előnyt szerezni a többszálú környezetből.

A Monte Carlo algoritmus esetén lehetőségünk van a szimulációk több szálon való elvégzésére. A kulcs a helyes korlát megállapítása: hány szimulációtól érdemes

szálat indítani. Az egészen biztos, hogy kis táblaméret mellett, rövid szimulációkkal nem éri meg több szál indítása.

A Monte Carlo Keresés, avagy Monte Carlo Tree Search (MCTS) egy módszer részleges játékfában történő döntés meghozatalára. Ez az algoritmus statisztikai módszerre épül és véletlenszerű.

Keresési fát építünk csúcsonként, ezekből a legígéretesebbet vagy a legígéretesebbeket kiterjesztjük. A kiterjesztett csúcsokra szimulációt futtatunk, ami teljes egészében véletlenszerűen előállított lépések sorozata, egészen a terminálási állapotig. A kapott értéket (végeredményt) felfelé propagáljuk a keresési fában. A szimulációkat futtathatjuk egy mennyiségi korlátig, vagy időkorlátig, de végig az algoritmus 4 lépését ismételjük:

- Szelekció  
A gyöker csúcsból kiindulva kiválasztjuk a legígéretesebb gyermeket vagy gyermekeket. Ezt egy  $V$  értékfüggvény segítségével számítjuk ki.
- Kiterjesztés  
Amennyiben a gyerek nem levél, azaz terminálási állapot, a kiválasztott gyereket kiterjesztjük és a gyermekeiből kiválasztunk egyet.
- Szimuláció  
Futtatjuk a szimulációt a kiválasztott csúcsra a terminálási állapotig, véletlenszerű lépésekkel.
- Propagálás  
Frissítjük a fát a kapott végeredményt felfelé propagálva.

A terminálási állapot hasznosságfüggvénye nyert/vesztett, azaz 1/0 értékeket képez, ezeket az értékeket propagáljuk felfelé. Ahogy a korábbi fejezetben tárgyaltuk, dönthetünk úgy, hogy többször futtatjuk a szimulációt, és visszaadhatjuk a nyertes szimulációk számát. Így  $n$  alkalommal futtatott véletlenszerű szimulációval elérhetjük, hogy az egyszerű nyert/vesztett értékek helyett inkább egy  $[0, n]$  intervallumra képezzen a hasznosságfüggvény.

Láthatjuk, hogy a 4 lépés közül a szelekció, kiterjesztés, illetve a propagálás során nehezen tudunk párhuzamosítani, hiszen ezek a műveletek egymástól függenek. A szimulációk párhuzamos végrehajtására megoldás egyfajta brach-predictor implementálása.

A brach-predictor ismerős lehet a pár évvel korábbi processzor botrányból, mely során a Spectre és Meltdown névre keresztelt módszerekkel, a processzorok brach-predictor biztonsági hiányosságait kihasználva tették kiolvashatóvá a memóriát. A módszerünk lényege igazából megegyezik a hardveres branch-predictor működési elvével.

A futás gyorsítása végett minden csúcsra futtat szimulációt, függetlenül attól, hogy ki lett-e terjesztve. Így ha több csúccsal, illetve csúcsonként számos szimulációval számolunk, a párhuzamos végrehajtás egy mennyiségi korlát átlépése után egészen biztosan pozitív hatással lesz a futási időre. A szimulációk eredményét felírjuk a csúcsokra. Amikor a csúcs kiválasztásra kerül, megnézzük, hogy van-e előre kiértékelt eredményünk, ha igen, akkor felhasználjuk azt és propagáljuk. Előfordulhat, hogy a lefuttatott szimulációk nem kerülnek felhasználásra, ha az a csúcs nem kerül a továbbiakban kiterjesztésre. Ez előfordulhat, hiszen változó mélységű kiértékelő módszerről beszélünk.

## 6.3 Virtuális környezet

A demo program elkészítésénél virtuális gépet használtam. A natív környezet lecserélését virtualizált környezetre biztonsági megfontolások és többszörös hardvermeghibásodások indokolták.

Egy napi használatban lévő gép megköveteli szoftverek up-to-date tartását. Ilyen a Python vagy a MS Visual C++ redistributable minimal runtime environment is. Egy frissítéssel további komponensek frissítését követelheti a meg, mint ahogy a cpp runtime a Tensorflow frissítését, ami a Keras frissítését, ami a meglévő kódok refaktorálásához vezethet. A megfontolás szerint a frissítés ne egy automatikus

folyamat legyen, hanem egy döntés, ami visszaállítható. A virtuális gépek másolhatóak és pillanatképek készíthetők róluk.

Hardverhiba esetén (pl. háttértároló meghibásodás) az operációs rendszer ismételt telepítése szükséges. Ekkor külön kis projekt egy környezet az összes akkori verziójának beszerzése és visszaállítása. Virtuális gépek esetén ez nem jelent problémát, hiszen a virtuális gépből könnyen készíthető másolat egy leválasztható háttértárolóra vagy felhő tárhelybe. A napi szinten változó kódok pedig egy szintén felhő alapú verziókövető rendszer (pl. github repository) tárhelyre kerülhetnek, ahonnan visszaállítható a legutolsó push-olt commit.

A virtuális gép további előnye, hogy megosztható. A másik félnek nem szükséges a környezetek telepítése, csak a virtuális gép futtatása szükséges.

A szépségei mellett több hátránya van. Egy „jól felszerelt” virtuális gép nagy, az én esetemben ~30GB. Ez tartalmaz telepített Linux rendszert, 8GB swap partíciót és ~4GB szabad területet. Ehhez valószínűleg nem elég egy ingyenes felhőtárhely, de egy USB drive is kevés lehet. Tehát szükség lesz egy nagyobb méretű felhős tárhelyre vagy külső adathordozóra.

A másik probléma a biztonsági mentés nehézsége. Itt egy közel 30GB méretű állományból kell másolatot készíteni, akár többet is, ami felhőtárhelybe való feltöltés esetén, de még külső meghajtóra való másolás során is extra időráfordítást igényel. Tegyük hozzá, hogy a teljes virtuális gép mentése nem szükséges napi szinten, hiszen a forráskódok akár naponta többször is feltölthetők távoli verziókövetőbe, ami viszont maximum másodpercek kérdése.

A rugalmasságért nem csak tárhellyel, idővel, hanem performancaival is fizetnünk kell. A virtuálisat környezetek nyilvánvalóan lassabbak és nagyobb memóriaigénnyel kell számolni, hiszen a natív rendszer folyamatosan be van töltve a memóriába és futtatja a vendég környezetet. A lassulás főleg akkor érezhető, ha a virtuális környezet elkezd swap-elni, azaz a fizikai memória elfogytával a háttértárolóra kezdi el kiírni a memóriából az adatokat.

A virtuális környezet sebességének a többmagos processzorok egyértelműen jól tesznek, egy 4 magos 8 szálas processzor esetében, ami rendelkezik virtualizációt támogató technológiával (pl. i7 vPro), illetve mindkét rendszer számára elegendő memóriával, de nincsenek melegedési problémái, a virtuális környezetek gördülékenyen használhatóak.

Összességében, a virtuális gépek kiegészítve felhős verziókövető rendszer (VCS) használatával, erőforrásért cserébe biztonságot adnak, hogy a teljes infrastruktúra hordozható és lementhető. Virtuális környezetek egyébként bérelhetőek is (pl. AWS, AZURE), de ezek nem ingyenesek.

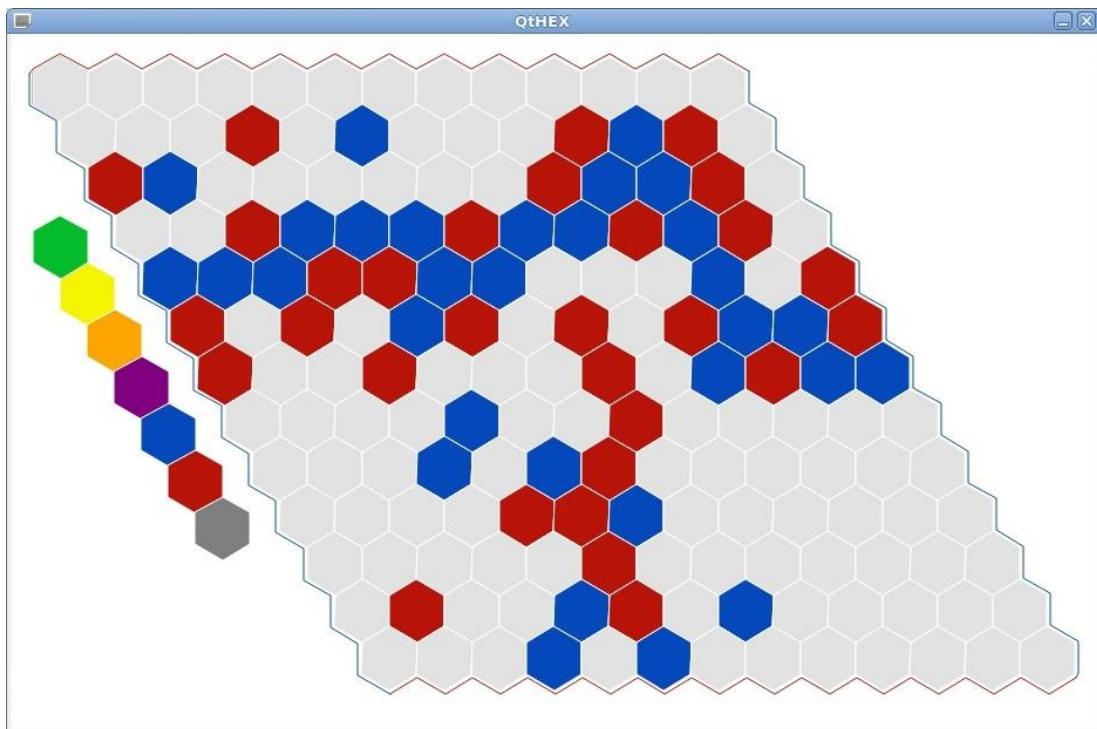
Bizonyos virtualizációs technikák úgynevezett hypervisor technológián alapulnak, amikor a virtuális gép közvetlenül veszi igénybe a fizikai erőforrásokat. Ebben az esetben a virtuális gépekre teljes operációs rendszer telepítése szükséges. Ilyen például az Oracle VirtualBox. Ma ettől fejlettebb és praktikusabb technológia a hypervisor nélküli technológia, mely során a virtuális gép (guest) a gazda rendszer (host) kerneljétől veszi igénybe az erőforrásokat. Ekkor nem szükséges egy teljes operációs rendszer telepítése a virtualizált környezetben, annak egy kis méretű részlete elégséges. Docker esetében rengeteg ilyen előre elkészített virtuális rendszer elérhető. A nagy hátránya, hogy a vendég környezet nagy mértékben függ a gazda operációs rendszertől, hiszen annak kerneljére épül. Így, például egy Linux alatt kialakított virtuális környezet átmozgatva egy Windows operációs rendszer alá nem fog működni.

## 6.4 Python virtuális környezet

A Python lehetőséget ad a futtató környezet hordozhatóságára, ennek a neve vEnv. A scriptek futtatásához szükség van egy python értelmező környezetre (python interpreter), melynek részeit képezik a telepített Python csomagok (pip packages) is. Ha ezeket nem rendszerszinten telepítjük és használjuk, hanem a projektnek egy erre dedikált könyvtárában helyezzük el, a projekt hordozhatóvá válik. Részt fogja képezni a forrásfájlok és projektfájlok mellett a futtató környezet is.

## 7 Felhasználói kézikönyv

A programot futtatva megjelenik a lent látható felület. Amennyiben a táblaméret 13x13, akkor a neurális hálózat betöltése a háttérben fut, melynek állapotát a konzol ablakon lehet követni. A tábla bal oldalán találhatóak különböző funkciógombok. A kék vagy piros játékos egy üres mezőre kattintva léphet. A program használata során szükséges a konzol ablak használata is.



25. ábra:  
*felhasználói felület*

A funkciógombok kattintásával a következő műveletek érhetőek el:

- Kiírja a soron következő játékos színét a konzolra.

🟡 Játékfa építése és alfa-béta vágások alkalmazása a fa teljes mélységében. Vigyázat, kihasználja a teljes számítási kapacitást és hosszú ideig is eltarthat. Az eredményt kiírja a konzolra.

🟠 Neurális hálózat használata. Az eredményt kiírja a konzolra. Kizárólag 13-as táblaméreten értelmezett.

🟣 Heurisztika használata. Amennyiben a neurális háló előkészítette az értékeket, (tehát meg lett nyomva a narancssárga gomb előtte) akkor kombinálja őket, súlyozza a kapott heurisztikát. Az eredményt kiírja a konzolra.

🟢 Utolsó lépés visszavonása. Üres tábla esetén nem értelmezett.

🔴 Új tábla kérése, új játék indítása.

🟤 Játék állás betöltése fájlból.

```
1.03093 1.02041 1.02041 0.000592245 1.03093 1.03093 1.03093 1.03093 1.03093 1.03093 1.03093 1.03093 1.03093 1.02041 1.02041
1.03093 1.03093 1.03093 1.03093 1.03093 1.03093 1.03093 1.03093 1.03093 1.03093 1.03093 3.07829e-13 1.03093 1.03093 1.03093
1.03093 1.03093 1.03093 1.03093 1.03093 1.03093 1.03093 1.03093 1.03093 1.03093 1.03093 1.03093 0.989054 1.03093
1.03093 1.03093 1.02635 1.03093 1.03093
resultVect.size(): 147
The Predicted field's array index is: 1
The Predicted field's matrix index is: [1. row, 2. col]
The Predicted field's p = 1.03093
```

26. ábra:

konzolablak részlet

## 7.1 Követelmények

A megfelelő szoftver verziók telepítése esetén Linux és Windows rendszeren is futtatható az alkalmazás.

A tesztelést 8GB-os 2333MHz memóriával, 2 magos 4 szásas, 3.8GHz-es, 6. generációs Intel i7-es vPRO technológiát támogató CPU-val, virtualizált környezetben végeztem.



### 7.1.1 Python

Python 3.5.3 (default, Jul 9 2020, 13:00:10) [GCC 6.3.0 20170516] on linux

Fontos megemlíteni, hogy a PATH változóban rögzített python parancs ne a python 2-es verzióját hivatkozza.

### 7.1.2 Python PIP csomagok

```
absl-py==0.10.0
astunparse==1.6.3
Brlapi==0.6.5
cachetools==4.1.1
certifi==2020.6.20
chardet==3.0.4
colorama==0.4.3
gast==0.3.3
google-auth==1.20.1
google-auth-oauthlib==0.4.1
google-pasta==0.2.0
grpcio==1.31.0
h5py==2.10.0
idna==2.10
importlib-metadata==1.7.0
joblib==0.14.1
Keras==2.3.1
Keras-Applications==1.0.8
Keras-Preprocessing==1.1.2
louis==3.0.0
Markdown==3.2.2
numpy==1.18.5
```

oauthlib==3.1.0  
opt-einsum==3.3.0  
pandas==0.25.3  
protobuf==3.13.0  
pyasn1==0.4.8  
pyasn1-modules==0.2.8  
pygobject==3.22.0  
python-dateutil==2.8.1  
pytz==2020.1  
pyxdg==0.25  
PyYAML==5.3.1  
requests==2.24.0  
requests-oauthlib==1.3.0  
rsa==4.6  
scikit-learn==0.22.2.post1  
scipy==1.4.1  
six==1.15.0  
sklearn==0.0  
tensorboard==2.2.2  
tensorboard-plugin-wit==1.7.0  
tensorflow==2.2.0  
tensorflow-estimator==2.2.0  
termcolor==1.1.0  
urllib3==1.25.10  
Werkzeug==1.0.1  
wrapt==1.12.1  
zipp==1.2.0

A fenti PIP csomagok a következő paranccsal telepíthetők:

```
pip install 'PackageName==1.0'
```

### 7.1.3 Cpp Boost

A Boost egy hordozható C++ könyvtár, ami a kiegészíti C++ standard könyvtárait, kiemelten támogatva az oktatást és kutatást. A Boost C++ könyvtár a következő webhelyről érhető el:

[https://www.boost.org/users/history/version\\_1\\_62\\_0.html](https://www.boost.org/users/history/version_1_62_0.html)

Debian alapú Linux disztribúciók esetében a következő panccsal telepíthető:

```
apt-get install -s libboost-all-dev=1.62.0.1
```

### 7.1.4 Qt

A Qt egy C++ alapokon készített, hordozható, főleg grafikai és ablakozó keretrendszer, habár parancssoros (command line) alkalmazások támogatására is kiváló. Az online telepítő innen érhető el:

<https://www.qt.io/download>

Qt Creator 4.13.0

Based on Qt 5.15.0 (GCC 5.3.1 20160406 (Red Hat 5.3.1-6), 64 bit)

Built on Aug 25 2020 10:11:25

## 8 Üzembe helyezés

### 8.1 Neurális hálózat elkészítése

A feltüntetett verziók és függőségeik telepítése után a mellékleten található hex13 projektet nyissuk meg. Amennyiben a mellékelt neurális hálózat modellt használja, ugorjon az „Intefész tesztelése” pontra.

#### 8.1.1 Játékadatbázis előkészítése

A tanuló adatbázisban kb. 10 ezer végigjátszott játék található. Egy játék egy sor, a soron belül a lépések felsorolása található. A fájlt soronként feldolgozzuk és előállítjuk az összes tábla állapotot és megállapítjuk a nyertes játékost, valamint a szükséges lépések számát.

Futtassuk le a preprocessing1.py sciptet.

#### 8.1.2 Játékadatbázis előkészítése

Második lépésben az összes állapothoz hozzárendeljük a megtett lépések számát (mint szakaszváltozót) és a játék kimenetelét.

Futtassuk le a preprocessing2.py sciptet.

### **8.1.3 Tanítás**

Az előkészített értékekkel betanítjuk a neurális hálózatot. Ehhez az adatok további előkészítése, a tanítási és tesztadatok szétválasztása szükséges.

Futtassuk le a `learn.py` scriptet.

A tanítás több órát vesz igénybe, teljesítménytől függően.

### **8.1.4 Ellenőrzés**

A teszteléshez futtassuk le a `test.py` scriptet. Amennyiben 97%-ot közelítő eredményt kapunk, úgy a tanítást megfelelően sikerült elvégeznünk.

### **8.1.5 Interfész tesztelése**

Futtassuk le a `service.py` scriptet. Adjunk meg neki egy input adatot. Egyet talál a scriptfájlban. A bemenet egy tömböt vár, amiben tetszőleges számú állapot kerülhet. Az állapotok szintén tömbök, melyek egészeket tartalmaznak. Az első 169 érték a mezőket reprezentálja, az utolsó pedig a szakaszváltozó.

## **8.2 Applikáció indítása**

A feltüntetett verziók és függőségeik telepítése után a mellékleten található `qthex` projektet nyissa meg. Futtassa a mappában található binárist. Amennyiben módosítaná a tábla méretét, azt a `tablesize.size` fájlban kell átírnia.

## 9 Összefoglalás

### 9.1 Eredmények

A dolgozatban ismertettem a mesterséges intelligencia klasszikus és korszerű módszereit a HEX táblajáték vonatkozásában. Szerettem volna egy egyszerű felépítésű neurális hálózattal minél általánosabban megoldást találni, ami átvihető más táblajátékokra is. A dolgozat eredményeképp kombinált módszerekkel sikerült játszani a HEX játékot.

A korábban említett MoHex és Neurohex projektek sokkal nagyobb volumenű és komplexitású projektek. Ezzel szemben a dolgozatban általánosabb módszereket, illetve azok kombinációját vizsgáltam és kerestem módszert a HEX táblajáték játszására. Sajnos a készítőik nem biztosították a fordításhoz szükséges összes információt, sem pedig binárist. Néhány ismert hiba javítása után is a program elszáll, így az együttműködési lehetőség a könyvtárral meglehetősen korlátozott.

Kisméretű táblán (maximum 4, esetleg 5 méretű táblán) sikerült alfa-béta vágások használatával az első játékos számára nyerő stratégiákat valós időben keresni és azzal játszani. Nagyobb méretű táblán a min-max algoritmus és az alfa-béta algoritmus mélységi korlátozással valós időben javasolja a következő lépést. 13x13 méretben, ami a tipikus versenytábla méret, a betanított neurális hálózat predikciója súlyozottan kerül beszámításra.

A neurális háló tesztelése során a bemutatott módszerrel 97.51%-os pontossággal sikerült megállapítani, hogy egy állapotból az első játékos nyert, vagy sem.

A dolgozatban megvizsgáltam a gráf kiértékelő módszerek párhuzamosítását, illetve megoldást adtam a Monte Carlo algoritmus párhuzamos végrehajtására. Az algoritmus azon adaptációja, mely a hasznosságfüggvény helyett  $n$  szimulációt futtat és ez alapján viszi át az értéket a hasznosságfüggvényre, a dolgozat saját ötlete, illetve eredménye. A bemutatott heurisztikával való kombináció szintén az eredmények közé tartozik.

Sajnos a neurális hálózat terén elért eredmények nem elég jók egy humán játékos ellen, így megállapíthatjuk, hogy a bemutatott módszer túl egyszerű egy ilyen komplexitású probléma megoldására.

A számítási szálak növelésében, a neuronháló további tanításában, módosításában, a heurisztikák módosításában, nagyobb hatékonyságú gráf kiértékelések használatában rengeteg kiaknázatlan lehetőség maradt. Felügyelt tanítás esetén a neurális hálózattal mélyebbre kell menni.

## **9.2 Köszönetnyilvánítás**

Ezúton szeretnék köszönetet mondani Családomnak, legfőképpen Feleségemnek és Kisfiamnak, akik nagy segítséget nyújtottak, hogy elkészíthessem a diploma dolgozatomat.

Szeretném megköszönni témavezetőmnek, Szabó Lászlónak hihetetlen türelmét és rugalmasságát.

Köszönet illeti a Neurohex csapatát, akik biztosították az általuk generált adatbázist.

Végül, szeretném megköszönni munkaadómnak is, hogy soron kívüli szabadnapok formájában támogatott.

# 10 Referenciák

- [1] „Wikipedia Hex page,” [Online]. Elérés:  
[https://en.wikipedia.org/wiki/Hex\\_\(board\\_game\)](https://en.wikipedia.org/wiki/Hex_(board_game)). [Hozzáférés dátuma: 2020. 11. 30.].
- [2] „John Nash Wiki page,” [Online]. Elérés:  
[https://en.wikipedia.org/wiki/John\\_Forbes\\_Nash\\_Jr..](https://en.wikipedia.org/wiki/John_Forbes_Nash_Jr..) [Hozzáférés dátuma: 2020. 11. 30.].
- [3] „Piet Hein Wiki page,” [Online]. Elérés:  
[https://en.wikipedia.org/wiki/Piet\\_Hein\\_\(scientist\)](https://en.wikipedia.org/wiki/Piet_Hein_(scientist)). [Hozzáférés dátuma: 2020. 11. 30.].
- [4] „Computer Hex page,” Computer Hex research group, [Online].  
Elérés: <https://webdocs.cs.ualberta.ca/~hayward/hex/>. [Hozzáférés dátuma: 2020. 11. 30.].
- [5] Kenny Young, Gautham Vasan és Ryan Hayward, „NeuroHex: A Deep Q-learning Hex Agent,” [Online]. Elérés:  
<https://arxiv.org/pdf/1604.07097.pdf>. [Hozzáférés dátuma: 2020].
- [6] „OHex database of moves in the game of Hex,” [Online]. Elérés:  
<http://hex.kosmanor.com/hex-bin/board/10/>. [Hozzáférés dátuma: 2020. 11. 30.].
- [7] „Six Wins Hex Tournament,” [Online]. Elérés:  
<http://webdocs.cs.ualberta.ca/~hayward/papers/rptTorino.pdf>.  
[Hozzáférés dátuma: 2020. 11. 30.].



- [8] „Chessprogramming Hex page,” [Online]. Elérés: <https://www.chessprogramming.org/Hex>. [Hozzáférés dátuma: 2020. 11. 30.].
- [9] David Gale, „The Game of Hex and the Brouwer Fixed-Point Theorem,” *The American Mathematical Monthly* vol. 96, 818-827, 1979.
- [10] Stefan Reisch, „Hex ist PSPACE-vollständig,” in *Acta Informatica*, Springer, 1981, p. 167–191.
- [11] Cameron Browne, Hex strategy - making the right connections, A K Peters, 2000.
- [12] Fekete István, Gregorics Tibor és Nagy Sára, in *Bevezetés a Mesterséges Intelligenciába*, ELTE Eötvös Kiadó, 2006.
- [13] Ian Goodfellow, Yoshua Bengio és Aaron Courville, Deep Learning, MIT Press, 2016.
- [14] „Tensorflow webpage,” [Online]. Elérés: <https://www.tensorflow.org/>. [Hozzáférés dátuma: 2020. 11. 30.].
- [15] „Keras webpage,” [Online]. Elérés: <https://keras.io/>. [Hozzáférés dátuma: 2020. 11. 30.].
- [16] „Wikipedia Tensor page,” [Online]. Elérés: <https://en.wikipedia.org/wiki/Tensor>. [Hozzáférés dátuma: 2020. 11. 30.].
- [17] „Wikipedia Neuron page,” [Online]. Elérés: <https://simple.wikipedia.org/wiki/Neuron>. [Hozzáférés dátuma: 2020. 11. 30.].
- [18] „Wikipedia Sigmond Function page,” [Online]. Elérés: [https://en.wikipedia.org/wiki/Sigmoid\\_function](https://en.wikipedia.org/wiki/Sigmoid_function). [Hozzáférés dátuma: 2020. 11. 30.].
- [19] Philip Thomas Henderson, Playing and solving the game of HEX, Edmonton, Alberta, 2010.