GENETIKUS ALGORITMUSOK

Készítette: Kovács Dániel László (dkovacs@mit.bme.hu)

1. Felkészülés a mérésre

Az optimális megoldás keresésére számos módszer ismeretes, melyek között a természetes evolúció (Darwin, 1859) számítógépes modelljei rangos helyet foglalnak el. Az evolúciós számítási modelleknek három fajtáját szokták megkülönböztetni aszerint, hogy milyen absztrakciós szinten interpretálják a darwini posztulátumot, miszerint a közös erőforrásokért versengő populáció azon egyedei kerülnek előnyösebb helyzetbe, melyek a versenyben előnyt jelentő tulajdonságokat hordoznak. A legabsztraktabb megközelítés evolúciós programozás (Fogel, Owens és Walsh, 1966; Fogel, 1995) néven vált ismertté, és a fajok közti versengést modellezi. Köztes absztrakciós szinten mozognak az evolúciós stratégiák (Schwefel, 1995), melyek az egyed szintjén vizsgálják a természetes kiválasztódás darwini folyamatát. A legelemibb megközelítésnek a genetikus algoritmusok (Holland, 1975; Goldberg, 1989) családjába tartozó módszerek tekinthetők, melyek a gének közti versengés szintjén követik nyomon az evolúció folyamatát. A mérés során az ún. klasszikus genetikus algoritmussal fogunk foglalkozni.

Néhány ajánlott irodalom annak, akit általában mélyebben is érdekel a téma:

- **Álmos A., Horváth G, Várkonyiné dr. Kóczy A., Győri S. (2003). *Genetikus algoritmusok*. Typotex.**
- Darwin, C. (1859). *On the origin of species by means of natural selection*. Murray.
- Fogel, L. J., Owens, A. J., Walsh, M. J. (1966). *Artificial Intelligence through Simulated Evolution*, New York: John Wiley.
- Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press.
- Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley.
- Fogel, D.B. (1995). *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence, Piscataway*, NJ: IEEE Press.
- Schwefel, H.P. (1995). *Evolution and Optimum Seeking*. Sixth-Generation Computer Technology Series. John Wiley & Sons, Inc., New York.
- Goldberg, D. E., Sastry K. (2010). *Genetic Algorithms: The Design of Innovation (2nd edition)*. Springer.

Ellenőrző kérdések

- Mik a klasszikus genetikus algoritmus főbb lépései?
- Mit ad eredményül a `main('bin')` parancs, és miért?
- Hogyan működik a rulett-kerék alapú szelekció?
- Mi a genotípus és a fenotípus? Mi a kapcsolat/különbség kettejük közt?
- Hogyan kódolhatunk egy órarendet (kromoszóma, gének, dekódolás)?

2. A mérésről általában

A mérésre hozni kell az elkészített házi feladatot (lásd később)!

A mérés során egy megadott könyvtárban található genetikus programcsomagot fogunk használni (erről a mérésvezető rövid ismertetést ad a mérés elején). Ezt a MATLAB indítása után hozzá kell adni a path-hoz, hogy a MATLAB mindenhonnan elérje. A programcsomag letölthető a tárgy honlapjáról (http://www.mit.bme.hu/oktatas/targyak/vimim306). Töltse le, és ismerkedjen meg vele!

A mérés során jegyzőkönyvet kell készíteni. A jegyzőkönyv lényegre törő, világos kell, hogy legyen. Minden feladnál tartalmaznia kell a feladat megfogalmazását, a megoldás menetének főbb részleteit, az eredményeket és az *eredmények értékelését*.

A jegyzőkönyv, és az egyéb anyagok (pl. forráskódok) beadásával kapcsolatos információkat (beadás módja, határideje, stb) a mérésvezető hirdeti ki a mérés folyamán.

3. Otthoni feladatok (és elméleti alapok)

A **genetikus algoritmus** (röviden: **GA**) és különböző változatai globálisan optimális sztochasztikus keresőeljárások. Ez azt jelenti, hogy – a véletlenszerűség felhasználásával – bizonyítottan konvergálnak a keresési tér jósági mérce által meghatározott valamely *globális optimumához*. Céljuk és hasznuk tehát: jól-definiált problémák megközelítőleg globálisan optimális megoldása.

A klasszikus genetikus algoritmusok esetében az **egyed**eket bináris bit-füzérek (**kromoszómá**k) reprezentálják, melyek különböző pozícióiban (**allél**jain) 0 és 1 értékek lehetnek: ezeket az értékeket nevezzük **gén**eknek. Természetesen nem csak bináris kromoszómák lehetségesek. Egy-egy gén értéke tetszőleges lehet (pozitív egész szám, valós szám, vagy akár valamiféle struktúra, stb) a megoldani kívánt problémától függően.

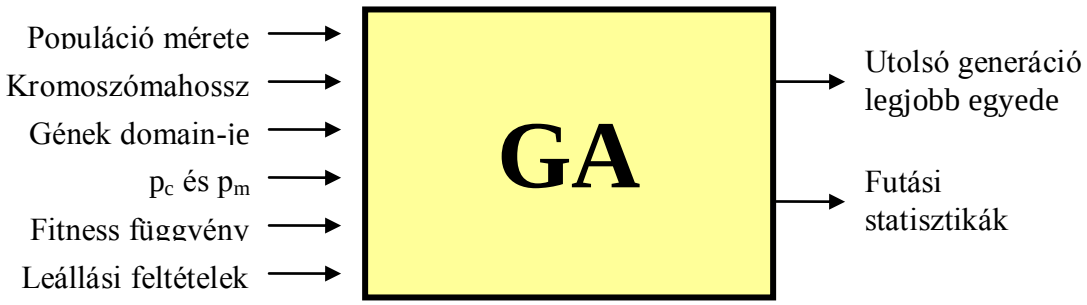
Az algoritmus nagy vonalakban a következő: egyedek egy **kezdeti**, véletlenszerűen előállított **populáció**ját úgynevezett **genetikus operátorok** felhasználásával addig-addig módosítja, mígnem az aktuális **generáció** megfelelő jellemzői (pl. generáció sorszáma, legjobb egyede, egyedeinek átlagos jósága) eleget nem tesznek bizonyos **leállási feltétele**knek. A következő genetikus operátorokat használjuk:

1. **Szelekció:** adott generáció egyedeinek jóság szerinti kiválasztása a következő generáció előkészítéseképp. Az egyedek jóságát meghatározó jósági mércét **fitness függvény**nek nevezzük. A fitness függvény alapesetben egy-egy valós számértéket rendel a populáció egyedeihez, mely az adott egyed populáció viszonylatában vett jóságát reprezentálja.
2. **Keresztezés:** a kiválasztott egyedek kromoszómáit párokba rendezzük (ezek lesznek a szülők), és adott valószínűséggel egy véletlenszerűen választott génpozíciótól (alléltól) kezdve megcseréljük a génkészletüket.[1]
3. **Mutáció:** a kiválasztott (és esetleg keresztezett) egyedeket továbbá még mutálhatjuk is. Ez azt jelenti, hogy végighaladunk a génkészletükön, és minden egyes gén értékét adott valószínűséggel véletlenszerűen módosítjuk (pl. bináris esetben invertáljuk).

1. ábra: genetikus algoritmusok felépítése

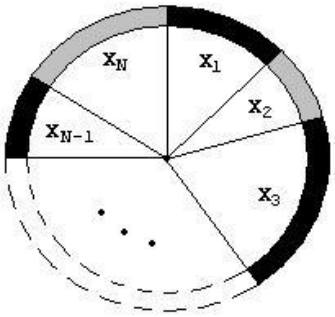
[1] Lényeges szempont, hogy a genetikus operátorok vajon értelmes/kiértékelhető egyedeket állítanak-e elő.

Az algoritmus menete tehát a következő (lásd. 1. ábra): az algoritmus bemenete **(1)** *populáció mérete*, **(2)** az *kromoszómák hossza*, **(3)** a *gének értékkészlete* (és esetleg egyéb tulajdonságai), **(4)** az *operátorok valószínűsége* (a keresztezéshez tartozó valószínűséget általában p_c, míg a mutációs valószínűséget p_m jelöli), **(5)** a *fitness függvény* (és paraméterei), és végül **(6)** a *leállási feltételek*. Ezek alapján az algoritmus indításakor előállítjuk az egyedek egy véletlenszerű populációját (0. generáció). A kezdeti populáció minden egyedének kiszámoljuk a fitness-ét, mely alapján kiválasztunk közülük egy – a szelekció típusától függő – mennyiséget. A kiválasztott egyedeket párokba rendezzük, a párokat (szülőket) keresztezve és mutálva kapjuk a **következő generáció** egyedeit (gyerekeket). Az új generációban esetlegesen rendelkezésre álló további, még üres helyeket általában az előző generáció legjobb egyedeivel töltjük ki (**K-elitizmus**). Így haladunk generációról generációra, mígnem teljesül valamely leállási feltétel (pl.. elérjük a generációk maximális számát, vagy az utolsó generáció legjobb egyedének jósága elér egy adott szintet, vagy az utolsó generáció egyedeinek átlagos jósága ér el egy adott szintet, vagy a legjobb egyed jósága adott számú generáció óta változatlan, vagy e jóság a növekménye van adott küszöb alatt, vagy növekedés gradiense nem elég nagy már, vagy valami más egyéb, akár probléma-specifikus szempont). Ekkor tehát leáll az algoritmus, és általában **(1)** az *utolsó generáció legjobb egyedével*, és **(2)** esetleg a *futásra vonatkozó statisztikákkal* tér vissza (ez tehát az algoritmus kimenete).



2. ábra: genetikus algoritmusok ki- és bemenete

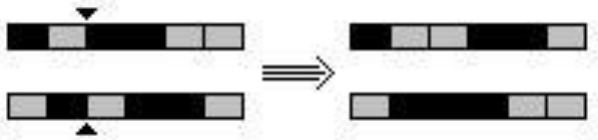
A genetikus operátorok megvalósítására több lehetőségünk is van. Például a **szelekció rulett kerék elven** működik, ha az egyedeket jósági értékükkel arányos valószínűséggel választjuk ki (és így a természetben látottak mintájára mindenkinek van esélye továbbörökíteni génjeit, csak legfeljebb kinek több, kinek kevesebb eséllyel). Legyen a P populáció $x_i \in P$ egyedének jósága $f(x_i)$. Tegyük fel, hogy a populáció N számú egyedből áll. Képzeletben osszunk fel egy egységnyi kerületű rulett-kereket N darab $\frac{f(x_i)}{\sum_{j=1}^{N} f(x_j)}$ ívhosszú körcikkre, majd „perdítsük meg a kereket”, azaz állítsunk elő egy *[0,1)* tartományba eső, valós véletlen számot egyenletes valószínűséggel. Ahol a „golyó megáll”, azaz amely egyednek megfelelő körcikk íve által meghatározott *[0,1)*-beli zárt rész-intervallumba esik az előállított véletlen szám, azon egyed kerül kiválasztásra. A 3. ábra szemlélteti a rulett-kerék elvet.



3. ábra. rulett-kerék alapú szelekció

A **mérkőzés alapú szelekció** is a rulett-kerék elvre támaszkodik: *K*-szor „pörgetjük meg a rulett-kereket", majd az így kiválasztott *K* egyed legjobbjával térünk vissza. Az új generáció előállításához ezt nyilván *N*-szer ismételjük. A **felső százalék elve** szerint a populáció egyedeinek legjobb *N* százalékából választjuk ki véletlenszerűen a következő generáció egyedeit. A **valahány legjobb kiválasztás** lényege, hogy az egyedeket jóság szerint sorba rendezi, és az első valahány legjobbat választja ki közülük.

A szelekcióhoz hasonlóan a keresztezésnek is többféle módja lehetséges. Létezik például **egy-pontú** és **több-pontú keresztezés.** Egy-pontú keresztezésről már szót ejtettünk az előzőekben. Erre mutat példát a 4. ábra. A több-pontú keresztezés ennek egy értelemszerű kiterjesztése.



4. ábra. bináris kromoszómák egy-pontú keresztezése

A kiszelektált szülőket (a két kromoszómát) p_c valószínűséggel keresztezzük, mielőtt mutációra kerülne sor. A keresztezési pont egyenletes eloszlás szerint, véletlenszerűen kerül kiválasztásra. Ezek után a mutáció végighalad mindkét kromoszóma génjein, és p_m valószínűséggel átbillenti őket egy másik értékre (bináris esetben invertálja, valós esetben adott értékkészleten belüli valós véletlen számmal helyettesíti, míg egész számok esetén pl. kivonja a gén értékét egy maximumból, vagy a valós esethez hasonlóan, az értékkészleten belül generál egy másik véletlen értéket).

Mindeddig csak olyan esetekről beszéltünk, ahol a kromoszómákat direkte értékeltük ki. Elképzelhető azonban olyan eset/probléma is, ahol a kromoszómákat kiértékelés előtt dekódolni kell (pl. ha neurális hálók struktúráját, vagy Forma1-es autók karosszériáját, kari órarendet, döntési fákat, vagy esetleg Java-nyelvű programokat kódolnak). Ilyenkor a kromoszómát **genotípus**nak nevezzük, a dekódolási eljárást **episztázis**nak, míg a dekódolt egyedet, amelyre a fitness-t meghatározzuk, **fenotípus**nak. Ezekben az esetekben a fitness kiértékelése igen számításigényes lehet, és számottevő hatékonyság növekedést eredményezhet, ha kihasználjuk a **genetikus algoritmusban rejlő inherens párhuzamosíthatóság**ot (pl. azt, hogy adott generáción belül az egyedek fitness-ét egymástól függetlenül, párhuzamosan számíthatjuk; a genetikus operátorokat gyakorlatilag parallel hajthatjuk végre rajtuk, stb).

1. HÁZI FELADAT: algoritmus indítása

Töltsük le a labortárgy weblapjáról a méréshez tartozó forráskódokat, bontsuk ki őket egy megfelelő könyvtárba (pl. `c:\MATLAB\work\ga`), és állítsuk be ezt a könyvtárat aktuálisnak: `cd('c:\MATLAB\work\ga');`

Az algoritmust a `ga.m` függvény valósítja meg. Viszont ennek paraméterezése nem triviális (annak ellenére, hogy gyakorlatilag egy-az-egyben megegyezik a fentebb leírtakkal). Ezért a `ga.m` függvényt egy megfelelően kialakított `main.m` függvényből hívogathatjuk. Írjuk is be a következőt: `main;`

A `main.m` függvénynek egyetlen bemenete van, ami azonosítja a probléma-területet (és problémát is), ahol az algoritmus futtatjuk. Az előbbi parancs hatására ezek a string-típusú switch-ek listázódnak ki.

```
The main script should be called with only one input parameter that specifies
the problem domain where GA is used:

       main(probdom);

probdom = 'bin'     (for an evolution of binary chromosomes)
probdom = 'dio'     (for an evolution of solutions to diophantos' equations)
probdom = 'netw'    (for an evolution of neural network weights/biases)
probdom = 'nets'    (for an evolution of neural network structures)
probdom = 'tmt'     (for an evolution of timetables)
```

A listából is látszik, hogy a mérés során a következő problémák megoldására fogjuk használni az algoritmust: **(1)** *bináris kromoszómák*, **(2)** *diofantoszi egyenlet-megoldások*, **(3)** *neurális hálózatok súlyainak és bias-értékeinek*, **(4)** *neurális hálózatok struktúrájának*, és **(5)** *villanykari órarendek* evolúciójára.

Indítsuk el tehát az algoritmust a legegyszerűbb esetben, bináris kromoszómák evolúciójára:

```
main('bin');
```

A parancs kiadását követően elindul az algoritmus, és generációról generációra haladva kijelzi számunkra az aktuálisan legjobb egyed fitness értékét, és az adott generáció átlagos fitness-ét. Mikor teljesülnek a `main.m` függvényben megadott leállási feltételek, az algoritmus leáll, és visszaadja a legjobb egyedet (mármint magát a kromoszómát), és megjeleníti a futási statisztikát is (amin nyomon következő a legjobb és az átlagos fitness alakulása a generációk során).

A futás során a fitness-t a kromoszómák decimális értékeként kaptuk (ezt kívántuk maximalizálni), így nem meglepő, hogy csupa 1-esből álló egyed jön ki végeredményképpen. – **Teszteljük többszöri futtatással az algoritmus robusztusságát!**

2. HÁZI FELADAT: algoritmus kódjának áttekintése és megértése

Nyissuk meg a `main.m` függvény forráskódját a MATLAB kód-szerkesztőjében, és tekintsük át a kód működését nagy vonalakban!

Amit látunk, a következő: a script a köszöntő üzenet kiírása és bizonyos környezeti paraméterek (pl. számformátum, többszálúság) beállítása után egy nagy `switch`-hez érkezik. Ez a `switch` a `main.m` függvény egyetlen bemenete szerint ágazik el.

Számunkra egyelőre még csak a `case 'bin'` eset érdekes. Vessünk erre egy pillantást!

Amit látunk, a következő: a köszöntő üzenet után beállításra kerülnek azok a változók, amik később a genetikus algoritmus bemenetét képezik:

1. populáció mérete `[pop_size]`
2. kromoszómák hossza `[chr_length]`
3. gének `[genes]`
4. keresztezési valószínűség `[pc]`
5. mutációs valószínűség `[pm]`
6. fitness függvény `[fitness]`
7. leállási feltételek `[stopcrit]`

Ez a 7 db. változó kerül tehát átadásra a `ga.m` függvény számára.

```
[best, history] = ga(pop_size, chr_length, genes, pc, pm, fitness, stopcrit);
```

A `ga.m` függvény két kimenettel tér vissza:

1. utolsó generáció legjobb egyedének kromoszómája `[best]`
2. a legjobb és az átlagos fitness alakulása a generációk során `[history]`

Ebből az utolsó generáció legjobb egyedének kromoszómáját megjelenítjük (`disp`), míg a `history`-t maga a `ga.m` függvény jeleníti még meg kilépés előtt (`plot`).

Vegyük sorra `ga.m` függvény bemenő paramétereit!

`1. pop_size:`	páros pozitív egész szám (a szelekció miatt páros).
`2. chr_length:`	tetszőleges pozitív egész szám.
`3. genes:`	struktúra, melynek 2 slot-ja van: **(1)** a `genes.domain` egy 2-elemű sorvektor, amely a gének értékkészletének alsó és felső határát határozza meg, és **(2)** a `genes.discrete` egy boolean-típusú érték, és akkor hamis, ha a gének értéke az adott probléma esetén folytonos, egyébként igaz (mint pl. esetünkben is).

4. `pc`: 0 és 1 közti tetszőleges valós szám. Általában 0.6 körüli értékre szokás beállítani.

5. `pm`: 0 és 1 közti tetszőleges valós szám. Általában 0.001 körüli értékre szokás beállítani, de problémától függően akár jóval magasabbra is vehetjük.[2]

6. `fitness`: struktúra, melynek 2 slot-ja van: **(1)** a `fitness.fname` a fitness függvényt megvalósító MATLAB script neve, míg **(2)** a `fitness.fparams` egy újabb struktúra. Ebben adjuk át a `fitness.fname` által meghatározott függvény számára a megfelelő, általában probléma-specifikus paramétereket. Esetünkben egyetlen ilyen paraméter van: a kromoszómák decimális értékének értéktartománya (`fitness.fparams.range`).[3]

7. `stopcrit`: struktúrák egy nem üres listája. A lista elemei az egyes leállási kritériumok, amelyek közül bármelyik teljesülése a genetikus algoritmus leállását eredményezi. Az egyes leállási feltételek tehát struktúrák, melyeknek 2 slot-ja van: **(1)** az `s.ctype` azonosítja a leállási kritérium típusát. Momentán 3 típust ismert az algoritmus: `mingennum` (a generációk számának maximális száma), `minbestf` (felső limit a legjobb fitness-re), és `minavgf` (felső limit az átlagos fitness-re). A leállási feltételek másik slot-ja **(2)** az `s.cparams`. Ez gyakorlatilag bármi lehet, leállási feltételtől függően. A jelenleg használt 3 leállási feltétel esetén most vagy egész, vagy éppen valós szám.[4]

Miután nagy vonalakban megértettük, hogy mire való a `main.m` függvény, vessünk egy pillantást a genetikus algoritmust megvalósító `ga.m` függvény kódjára, és értsük meg ezt is alaposabban!

Ez a kód már valamivel komplexebb, mint a `main.m` függvény kódja. Lényegében 2 részből áll: **(1)** bemenetek ellenőrzése, és **(2)** minden egyéb funkcionalitás. A bemenetek ellenőrzésére most nem térnénk ki (ennek megértése önálló feladat). Térjünk inkább ki a `ga.m` függvény funkcionális részére.

---

[2] Arra viszont ügyeljünk, hogy a mutációs valószínűség adott szint fölött teljesen **randomizálja a keresést**, amit végső soron a genetikus algoritmus végez a lehetséges kromoszómák által kifeszített keresési térben, a fitness függvény globális maximumát keresve egyszerre populációméret számú pontban, generációról generációra iterálva. Megjegyzés: ökölszabály, hogy ha az átlagos fitness a legjobb fitness környékén van, akkor célszerű megnövelni a genetikus operátorok (pl. a mutáció) valószínűségét, míg hogyha az átlagos fitness fehér zaj szerű tendenciát mutat, úgy érdemes ezeket az értékeket addig csökkenteni, amíg az átlagos fitness alakulása legalább minimálisan korrelál a legjobb fitness alakulásával.

[3] Később látni fogjuk, hogy a fitness függvény és paraméterei is problémáról problémára változnak.

[4] A leállási feltétel-struktúrákból képezett listát a `check_stopcrit.m` függvény értékeli ki az eddigi futási történet fényében (`history`).

Először is generáljuk a 0. generáció egyedeit:

```
pop     = initpop(pop_size, chr_length, genes);
```

Ezek után kiszámoljuk a 0. generáció egyedeinek fitness-ét:

```
fvals   = feval(fitness.fname, pop, fitness.fparams);
```

Ezek után frissítjük a `history`-t, és belépünk egy `while` ciklusba:

```
while ~check_stopcrit(history, stopcrit)
```

A ciklust (amennyiben még nem teljesült egyik leállási feltétel sem) azzal kezdjük, hogy átvesszük az aktuális generáció legjobb egyedét a következő generációba:

```
newpop = best;
```

Ezt nevezik tehát **1-elitizmusnak.**[5] Ezt követi a rulett-kerék alapú szelekció, keresztezés, és mutáció:

```
for i=1:pop_size/2,

      pair        = selection(pop, fvals);
      pair        = crossover(pair, pc);
      pair        = mutation(pair, pm, genes);
      newpop      = [newpop; pair];

end
```

Itt tehát az aktuális generáció egyedeit a hasznuk fényében válogatjuk ki páronként. A párokat p_c valószínűséggel keresztezzük, majd p_m valószínűséggel mutáljuk minden egyes génjüket, és végül hozzáadjuk őket az új generációhoz.[6]

Ezek után az új generáció lép az aktuális helyébe:

```
pop = newpop;
```

[5] Az elitizmus általában meggyorsítja az algoritmus konvergenciáját. Viszont kisméretű populáció és/vagy alacsony genetikus operátor valószínűségek esetén a populáció homogenizálódásához vezethet, ami az algoritmus lokális optimumban ragadását, és a konvergencia „megfeneklését" eredményezheti. „*In varietas delectat*" – lehetne a genetikus algoritmusok, és általában az evolúció egyik fő mottója. Megjegyzés: épp ezért a kiindulási populáció előállítása is létfontosságú lehet az algoritmus futása, konvergenciája szempontjából. Ha ügyesen generáljuk a kezdeti populációt, akkor akár egyetlen iteráció nélkül is elérhetjük a globális optimumot. Minél több apriori információnk van a globális optimumról, annál könnyebb „körberakni" a kezdeti populáció egyedeivel (mint a keresési tér globális optimum közelébe eső pontjaival). A homogén populáció tehát egyetlen keresési pontnak felelne meg, ami éppen az algoritmus écájának, a **párhuzamosan több ponton történő keresés**nek mondana ellent…

[6] Látható, hogy egy-egy egyedet akár többször is kiválaszthatunk a rulett-kerék szelekció során.

Az új generáció egyedeinek is kiszámítjuk a fitness-ét:

```
fvals   = feval(fitness.fname, pop, fitness.fparams);
```

Majd, az előzőekben leírtakhoz hasonlóan, kiegészítjük a `history`-t, és újrakezdjük a ciklust. Ha teljesül a leállási feltétel (ezt ugye a ciklus kezdetén ellenőrizzük), úgy kilépünk a ciklusból, kirajzoljuk (`plot`) a `history`-t, és visszatérünk a `ga.m` függvényt hívó függvény felé a megfelelő visszatérési értékekkel.

Ezzel gyakorlatilag megértettük a `ga.m` függvény működését is. Pillantsunk bele a genetikus operátorok kódjába (önálló feladat), és értelmezzük azt is![7]

Mielőtt tovább lépnénk, még érdemes kitérnünk a fitness függvény működésére. Bináris kromoszómák esetén, mint láttuk (a `main.m` függvényben) az `f_binary.m` függvény kerül meghívásra.

Az `f_binary.m` függvény is (a `ga.m`-hez hasonlóan) 2 részből áll: **(1)** bemenő paraméterek vizsgálata, és **(2)** minden egyéb funkció. Nézzük a függvény főbb funkcionalitását (a többi rész áttekintését és megértését önállóan kell megoldani)!

A függvény gyakorlatilag 2 soros:

```
(a)         fval = x * power(2*ones(n, 1), linspace(n-1, 0, n)');
(b)         fval = fval / (params.range(2) - params.range(1));
```

Az **(a)** sorban kiadott parancs arra szolgál, hogy kiszámolja a bemeneten érkező `x` generáció egyedeinek decimális értékét. Az `x` tehát egy mátrix, aminek sorai az egyedek kromoszómái (most binárisak, és `n`-hosszúak), és a mátrixnak annyi sora van, ahány egyed van a populációban (`k`). Ezt a mátrixot megszorozzuk jobbról egy `n`-hosszú oszlopvektorral, amiben a 2 különböző hatványai szerepelnek csökkenő sorban (`(n-1)`-től 0-ig). A jobbról szorzás eredményeképp, ami végső soron egy skaláris szorzás, egy `k`-hosszú oszlopvektor képződik az sorokban található bináris egyedek decimális értékével.

A **(b)** sorban kiadott parancs ezek után [0,1]-be normalizálja a kapott fitness értékeket. Ehhez nyilván ismernie kell a fitness értékek elméleti minimumát és maximumát.

Az `f_binary.m` függvény végül e normalizált fitness-szel tér vissza.[8]

[7] Legyünk tekintettel arra, hogy bár a kód viszonylag jól kommentezett, mégsem egészen triviális az értelmezése, ugyanis hatékonysági okokból igen tömörre/trükkösre kellett venni. A genetikus operátorok végrehajtása jelentős mértékben meghatározza az algoritmus futását. Egyszerűbb problémák esetében még a fitness kiértékelésénél is meghatározóbb lehet. Természetesen ez nem jelenti azt, hogy a jelen kódot nem lehet tovább optimalizálni… Szorgalmi feladat. ☺

[8] Ezzel tehát magyarázatot kaptunk arra, hogy miért csupa 1-esből álló egyedek jöttek ki a genetikus algoritmus futtatása során végeredményként…

3. HÁZI FELADAT: algoritmus kódjának módosítása

Kísérletezzünk az algoritmus különböző paraméterekkel való futtatásával (az előbbi bináris domain esetén)! Állítsuk át a populáció méretét, a kromoszómák hosszát, a genetikus operátorok valószínűségét, illetve a leállási feltételeket. **Mit tapasztalunk? Melyik beállításnak mi a hatása, és miért?**

Az előbbi kísérleteket követően állítsuk most át a `main.m` függvény `case 'bin'` részében a gének értékkészletét binárisról oktálisra. **Milyen változtatások kell még tennünk ahhoz, hogy oktális (8-as számrendszerbeli) számok maximalizálására tudjuk használni az algoritmust?**

Tipp: a fitness-nek megfelelő range-et kell átadni, illetve az `f_binary.m` függvényben immár nem bináris számrendszerben, hanem oktálisban kell számítanunk a kromoszómák értékét.

4. HÁZI FELADAT: algoritmus kódjának kiegészítése

Az előbbiekben végső soron azzal kísérleteztünk, hogy adott hosszúságú (bináris, oktális, stb) számok decimális értékét maximalizáljuk genetikus algoritmussal. Felmerülhet a kérdés, hogy: ***minek kell ehhez genetikus algoritmus? Miért lövünk ágyúval verébre?*** …hiszen mindenki tudja, hogy pl. adott hossz mellett a csupa 1-esből álló bináris számoknak van a legnagyobb számértéke. ***Mi értelme van így ennek az egésznek?***

A válasz egyszerű: a bináris eset csupán tesztelésre szolgált. Ily módon meggyőződhettünk az algoritmus helyességéről, konvergenciájáról, sebességéről. Másrészt az algoritmus működését is megérthettük egy egyszerűbb esetben. Természetesen bináris számok értékének maximalizálására nem éppen a genetikus algoritmus a leghatékonyabb, javasolt módszer. A genetikus algoritmus univerzális, domain-független probléma-megoldó eljárás. Számos területen be lehet vetni, de leginkább olyankor célszerű alkalmazni, mikor az adott domain-en belül nem ismeretes hatékonyabb módszer (pl. órarendek, neurális hálózatok, vagy más komplex rendszerek felépítésének optimalizálására). Olyan eset is elképzelhető, mikor nem is igen tudunk más módszert az adott probléma értelmes, globálisan optimális megoldására (pl. komplex függvények globális maximumának közelítése). Az algoritmus univerzalitásának nyilván megvannak a maga hátulütői (véges időben szub-optimalis; a problémától függő, optimális paraméterezés beállítása néha nehéz; a futási idő és konvergencia sebessége előre nem jósolható; a megoldás-jelöltek populációjának kezeléséből adódóan nagy a számítási igénye; a fitness függvény explicit megadása sokszor nem triviális, stb), de mindennek ellenére az algoritmust számos területen a mai napig sikerrel alkalmazzák. Előnye éppen rugalmasságában, moduláris, probléma-független felépítésében, és általános felhasználhatóságában rejlik. Analógia gyanánt gondolhatunk esetleg a Prolog-ra: egy programnyelv, végső soron univerzális, gyakorlatilag „bármit” megvalósíthatnánk benne, mégsem ezt használjuk mindenre. Bizonyos problémák esetén azonban mégis ez adja a leghatékonyabb megoldást (ahol más módszerekkel csak kifejezetten nehezen boldogulnánk (pl. természetes nyelvfeldolgozás, szakértői rendszerek)). Ne feledjük el, csak deklarálnunk kell a problémát, a megoldást már az algoritmus szállítja. A mi feladatunk „csak” a probléma megfelelő megfogalmazása – a „*mit?*” megadása –, a „*hogyan?*”-ra már az algoritmus válaszol.

Kicsit komolyabb probléma gyanánt keressük most meg egy saját komplex függvény globális maximumát!

Ehhez ki kell találnunk a függvényt, és megfelelően ki kell egészítenünk a forráskódot.

- A `main.m` fájlban hozzunk létre egy saját, `own` nevezetű `switch-case`-t a bináris esethez hasonlóan (annak copy-paste-elésével, és átnevezésével).
- Találjunk ki egy tetszőlegesen bonyolult numerikus függvényt, és próbáljuk meg a genetikus algoritmust arra használni, hogy megtalálja ennek a függvénynek a maximumát, azaz azt az `x` bemenetet, amire a függvény értéke globálisan maximális.
- Az `f_binary.m` fájl mintájára (annak duplikálásával, átnevezésével, és alkalmas átírásával) készítsünk egy saját fitness függvényt `f_own.m` néven, amely az előbb kigondolt komplex függvény értékét számítja ki a populáció minden egyes egyedére.
- Teszteljük és dokumentáljuk az algoritmus futását, továbbá verifikáljuk, illetve jelenítsük is meg a kimenetét!

Tipp 1: a sebesség érdekében próbáljunk meg beépített MATLAB eljárásokat használni a saját függvény kialakításakor (ekkor akár `for`-ciklus nélkül is, egy lépésben kiszámíthatjuk az összes fitness értéket, azaz a teljes `fval` oszlopvektort).[9]

Tipp 2: ügyeljünk arra, hogy a `ga.m` függvényt a `main.m` függvényből helyesen hívjuk meg. Azaz a függvényünkhöz illő fitness paraméterekkel (pl. range), génekkel (minimum, maximum, folytonosság), és leállási feltételekkel (pl. legjobb fitness limitje) dolgozzunk!

[9] Legyünk tekintettel arra, hogy az algoritmus konvergenciája igen számottevően függ a keresési tér sajátosságaitól. Ha példának okáért olyan saját függvényt konstruálunk, amely mindenhol zérus kivéve egyetlen helyet, ahol 1 az értéke, úgy ne csodálkozzunk azon, ha algoritmusunk „a sötétben fog tapogatózni", és csak nagyon lassan konvergál (a szerencsés kiindulási esetektől eltekintve). Ha mégis így döntenénk, úgy szorgalmi feladatként készítsünk egy `own_initpop.m` függvényt is, amely a saját függvényünkhöz illő kezdeti populációt generál. Ne felejtsük el a `ga.m` fájlt is átírni, hogy ezt használja!

4. Mérési feladatok

A mérés során (4 óra leforgása alatt) az otthon megkezdett munkára alapozva folytatjuk a genetikus algoritmusok, és alkalmazásaik megismerését.

0. MÉRÉSI FELADAT: globális maximum keresése

Aki odahaza nem jutott a 4. házi feladat végére (de már érdemben belekezdett), a mérés elején a mérésvezető engedélyével, és esetleg segítségével még befejezheti.

1. MÉRÉSI FELADAT: egyenletek megoldása

A következőkben arra használjuk algoritmusunkat, hogy viszonylag általános diofantoszi egyenleteket oldjuk meg vele (lévén ezeknek a megoldására nincs általános matematikai módszer)[10]. Diofantoszi egyenlet alatt tehát a következőt fogjuk érteni:

$$a_1 x_1^{n_1} + a_2 x_2^{n_2} + \cdots + a_k x_k^{n_k} = -c,$$

ahol $x_1, x_2, \ldots, x_k, n_1, n_2, \ldots, n_k \in Z^+$ és $c, a_1, a_2, \ldots, a_k \in R$

Ennek egy speciális esete például a **pitagoraszi egyenlet**, ahol $k = 3$ (azaz három **változó**val dolgozunk: x_1, x_2, x_3), $a_1 = a_2 = 1$ és $a_3 = -1$ (az **együttható**k), $c = 0$ (a **konstans**), és $n_1 = n_2 = n_3 = 2$ (a **hatvány**ok), azaz $x_1^2 + x_2^2 - x_3^2 = 0$, avagy $x_1^2 + x_2^2 = x_3^2$.

Az egyenleteket megoldás előtt homogén alakra hozzuk az algoritmus számára:

$$a_1 x_1^{n_1} + a_2 x_2^{n_2} + \cdots + a_k x_k^{n_k} + c = 0$$

Az *a* együtthatókat, az *n* hatványokat, és a *c* konstanst a következő változók reprezentálják a `main.m` függvény `case 'dio'` részében:

- `fitness.fparams.coeffs`: a változókhoz tartozó együtthatók listája
- `fitness.fparams.powers`: a változókhoz tartozó hatványok listája
- `fitness.fparams.c`: a homogén egyenletben szereplő konstans

Az egyedek (avagy a kromoszómák) rendre az *x* változók értékét tartalmazzák.

[10] Az itt szereplő diofantoszi egyenletek annyiban speciálisak, hogy a változóknak csak egy bizonyos hatványa szerepelhet bennük. Természetesen kis kiegészítéssel általánosabbá tehetők (szorgalmi feladat).

Az egyenletek homogén alakra hozásának az az értelme, hogy ilyen módon igen könnyen számítható az egyedek fitness értéke, ugyanis pusztán csak az egyenlet adott egyedben szereplő x értékekkel való behelyettesítése után adódó érték zérustól való abszolút értékben vett eltérését kell vennünk (pontosabban minimalizálnunk). Tegyük fel tehát, hogy adott $\underline{x} = x_1, x_2, ..., x_k$ egyed esetén adott együtthatók, hatványok, és konstans esetén

$$a_1 x_1^{n_1} + a_2 x_2^{n_2} + \cdots + a_k x_k^{n_k} + c = d$$

Az $\underline{x}$ egyed fitness-ét ekkor a következőképp kaphatjuk:

$$f(\underline{x}) = \frac{1}{1 + |d|}$$

Látható, hogy ez a függvény akkor maximális (=1), ha d minimális (azaz ideális esetben d=0). Ha a genetikus algoritmussal ennek a fitness függvénynek a maximalizálására törekszünk, akkor gyakorlatilag a homogén alakú egyenleteket próbáljuk meg megoldani. A kódban mindezt az `f_diophantos.m` függvény valósítja meg. Az előbbi kifejezésnek a következő sor felel meg benne:

```
fval = (1 + abs(params.c + (x.^repmat(params.powers, k, 1))*params.coeffs')).^-1;
```

Itt az x már nyilván nem csak egy sorvektor, hanem az egyedeket reprezentáló kromoszómák sorvektoraiból alkotott populáció-mátrix.[11]

1.1. mérési feladat

Az alapbeállításokat használva futassuk az algoritmust:

```
main('dio');
```

Értékeljük az eredményeket (magyarázzuk meg, hogy miért éppen azt kaptuk, amit kaptunk), majd írjuk át a `main.m` függvény `case 'dio'` részét úgy, hogy más értéktartományban keressünk megoldásokat!

1.2. mérési feladat

Írjuk át a `main.m` függvény `case 'dio'` részét úgy, hogy más, az eddigi pitagoraszi egyenlettől eltérő diofantoszi egyenleteket oldjon meg az algoritmus! Például többek közt megpróbálkozhatunk a nagy Fermat sejtés megdöntésével is, ha $x_1^n + x_2^n = x_3^n$

[11] A kód ily módon tömörebb, és ebből következően talán egy árnyalatnyival nehezebben is érhető, viszont legalább gyorsabb (mint pl. egy `for`-ciklus), hiszen egy lépésben számítja ki az összes egyed fitness-ét, továbbá igen kevéssé érzékeny a populáció méretének növelésére.

alakú egyenletek megoldásait keressük (feltehetően nem sok sikerrel), ahol $x_1, x_2, x_3, n \in Z^+$ és $n > 2$.

Többféle, lehetőleg összetettebb, sok-változós egyenlettel és különböző paraméter beállításokkal is kísérletezzünk!

2. MÉRÉSI FELADAT: evolúciós tanulás

A gépi tanulást tanulmányaink során visszavezettük már függvény approximációra/regresszióra (avagy – ezek speciális eseteként – osztályozásra). Mindazonáltal tekinthetünk a tanulásra még ennél is általánosabban úgy, mint egyfajta speciális keresésre.

A genetikus algoritmusok, mint láttuk, általában jól definiált problémák globálisan optimális megoldásának közelítésére alkalmasak. Mindezt több ponton párhuzamosan zajló sztochasztikus kereséssel oldják meg. *Mi lenne, ha most ezzel a kereséssel tanuló struktúrák, például neurális hálózatok paramétereit próbálnánk meg „megtanulni”?* *Megjegyzés: nyilván a neurális hálózatok tanítására számos dedikált, többé-kevésbé speciális módszer (tanuló/tanító eljárás) létezik, melyek – erre a feladatra – többnyire hatékonyabbak a genetikus algoritmusoknál, mégis az evolúciós tanulás neurális hálózatok, mint alapvető tanuló struktúrák esetén a legszemléletesebb (arról nem is beszélve, hogy több évtizedes kutatás áll a téma hátterében, illetve genetikusan nem csak ellenőrzött, hanem megerősítéses neurális tanulást is megvalósíthatunk).*

Ebben a feladatban tehát most az a célunk, hogy az evolúciót tanulásra használjuk. Konkrétan: kérdés-válasz (*x*,*y*) formában adott problémához szeretnénk megtalálni egy adott struktúrájú neurális hálózat azon paraméterezését, amely mellett a hálózat megfelelően kis általánosítási hibával reprodukálja a kérdésekre a válaszokat.

Az előző feladathoz hasonlóan itt is a kódolás kialakítása az első lépés. Előbb diofantoszi egyenletek megoldását szerettük volna kódolni (és ez még viszonylag triviális is volt, hiszen csak az egyenletben szereplő változók értékeit kellett felsorolni), most azonban ennél valamivel komplexebb struktúra paramétereit kell kódolnunk. A felparaméterezett neurális hálózat a fenotípus, aminek egy általunk meghatározott szintaxisú kromoszóma a gentotípusa. *Mi is legyen tehát ez a szintaxis?*

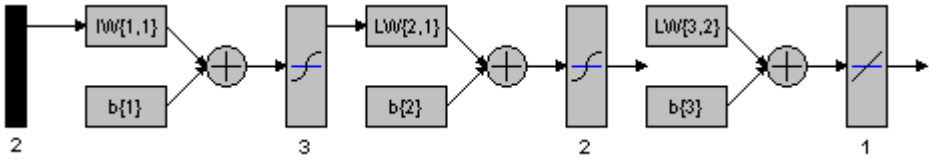
Ahhoz, hogy válaszolni tudjunk erre a kérdésre, először is el kell döntenünk, hogy *pontosan mit is szeretnénk kódolni? A hálózat súlyait, bias értékeit, neuronjai nemlinearitásainak paramétereit, esetleg ezek típusát, vagy a hálózat összeköttetéseit is, a teljes struktúrát?*

Az egyszerűség kedvéért egyelőre maradjunk abban, hogy – egy szokványos neurális tanító eljáráshoz hasonlóan – „csak” a háló súlyait és bias-értékeit keressük. A kérdés ekkor már csak az, *hogyan kódoljuk ezeket?*

Ennek eldöntését célszerű bizonyos feltevéseket tennünk a neurális hálók struktúrájára vonatkozóan. Feltevéseink most legyenek a következők:

- Csak előrecsatolt (feed-forward), több-rétegű (MLP) hálókkal foglalkozzunk.
- Minden réteg minden egyes neuronja álljon összeköttetésben a következő réteg összes neuronjával.
- Minden neuronhoz tartozzon bias érték.
- A rejtett rétegekben *tansig* nemlinearitás legyen, míg a kimenetiben *purelin*.

Ezzel tehát már egészen jól be is határoltuk, hogy mit és hogyan szeretnénk kódolni, illetve mindezt hogyan lehet majd szemantikailag helyesen dekódolni és interpretálni. Próbáljunk meg mindvégig a MATLAB **Neural Network Toolbox**-ához igazodni! Az említett toolbox (röviden: `nnet`) a következő ábrán látható struktúrában tárolja a fenti kritériumoknak eleget tevő feed-forward MLP-ket.



5. ábra. feed-forward MLP struktúra a MATLAB nnet toolbox-ában

Az ábrán egy 2 rejtett rétegű neurális hálózatot látunk példaképp, amit `p` kérdések és `t` válaszok mellett a következő utasítással hozhatunk létre:

```
net = newff(p, t, [3, 2]);
```

A hálózat (`net`) bemenetéről elég most annyit tudnunk, hogy 2-dimenziós (a mintapontok számát a struktúra kapcsán nem kell ismernünk). A bemenet tehát egy `2*size(p,2)` méretű mátrixként adott (lásd. 5. ábra bal szélső részén szereplő fekete téglalap). Az első réteg súlyait a `net.IW{1,1}` mátrix tartalmazza, aminek mérete `3*size(p,1)==3*2`-es (a `net.b{1}` bias-vektor mérete pedig nyilván `3*1`-es). Az első rejtett réteg kimenetét ezek alapján a következőképp számítjuk:

```
a1 = tansig(net.IW{1,1}*p + net.b{1});
```

Az `a1` oszlopvektor tehát 3 elemű – rendre az első rejtett réteg mindhárom neuronjának válaszát tartalmazza. Az első és a második rejtett réteg közti összeköttetések súlyait a `net.LW{2,1}` mátrix tartalmazza, aminek mérete `2*3`-as. A második rejtett réteg kimeneteit a következőképp számítjuk:

```
a2 = tansig(net.LW{2,1}*a1 + net.b{2});
```

Ennek a jelen esetben utolsó rejtett rétegnek a kimeneteit kell a `t` válaszok dimenziójához igazítottan súlyozva összegeznünk. Magyarán a rejtett rétegeken felül kell még egy kimeneti réteg is, amiben annyi neuron szerepel, ahány dimenziós `t` (tegyük fel a példa kedvéért, hogy `size(t,1)==1`, azaz a kimenet 1-dimenziós), és ezekkel kell összekapcsolnunk az utolsó rejtett réteg neuron-kimeneteit. A kimeneti réteg kimenetei lesznek neurális hálózatunk válaszai. Ezeket tehát a következőképp számítjuk:

```
y = purelin(net.LW{3,2}*a2 + net.b{3});
```

Itt tehát a `net.LW{3,2}` mátrix tartalmazza az utolsó (második) rejtett réteg, és a kimeneti réteg közti összeköttetések súlyait, és ennek megfelelően mérete `size(t,1)*2==1*2`-es.

Kitérő: ahhoz, hogy a fenti módon számított `y` kimenet megegyezzen az `nnet` toolbox `sim(net,p)` függvényének kimenetével, normalizálnunk kell a hálózat bemeneteit, majd az ezek alapján – fenti módon – számított kimeneteket is. Ezt a következőképp tehetjük meg:

```
p = processinputs(net, p);
     ...
y = processoutputs(net, y);
```

Visszatérve, példaképp próbáljuk meg előbbi feltevéseinknek megfelelően kódolni az előbbi 2 rejtett rétegű neurális hálózatot! Haladjunk végig sorra a rétegeken, és linearizáljuk a rétegek súlymátrixait és bias-vektorait – rakjuk sorba egymás után az oszlopokat! A `net` hálózatot kódoló `x` kromoszóma ekkor a következő:

```
IW    = net.IW{1,1};
b1    = net.b{1};
LW1   = net.LW{2,1};
b2    = net.b{2};
LW2   = net.LW{3,2};
b3    = net.b{3};

x     = [IW(:,1); ...
         IW(:,2); ...
         b1; ...
         LW1(:,1); ...
         LW1(:,2); ...
         LW1(:,3); ...
         b2; ...
         LW2(:,1); ...
         LW2(:,2); ...
         b3]';
```

Látszik tehát, hogy `x` egy sorvektor, melynek hossza `(3*2+3)+(2*3+2)+(1*2+1)=20`. Ezt a sorvektort (kromoszómát) bármikor dekódolhatjuk egy megfelelő neurális hálózattá, ha a bemenetek, elvárt kimenetek, és rejtett rétegekben lévő neuron-számok fényében `newff`-fel létrehozunk egy `nnet`-es neurális hálózatot, és az abban szereplő megfelelő mátrixokat az `x` kromoszóma megfelelő darabjaival töltjük fel. Ez a bemenet és kimenet dimenziójának, illetve a rejtett rétegekben lévő neuronok számának fényében egyértelműen megtehető. Erre szolgál például a `decode_chr2netw.m` függvény. Felmerül azonban a kérdés:

Mi lesz a kromoszómák által fenti módon kódolt neurális hálózatok fitness-e?

A kromoszómák által kódolt neurális hálózatok fitness-ét nyilván a `p` bemenetek, és az elvárt `t` kimenetek fényében tudjuk csak meghatározni (adott probléma esetén van csak értelme egy-egy hálózat jóságáról beszélni). A konkrét megoldást a `main.m` függvény `case 'netw'` részéhez kapcsolódó `f_neuralw.m` függvényben találjuk:

```
f = 1 / (1 + mse(t - sim(net, p)));
```

Ha tehát egy adott `x` kromoszómából dekódolt `net` hálózat kapcsán ezt az `f` értéket maximalizáljuk, úgy egyben a hálózat átlagos négyzetes hibáját minimalizáljuk.

2.1. mérési feladat

Az alapbeállításokat használva futassuk az algoritmust:

```
main('netw');
```

Értékeljük az eredményeket (magyarázzuk meg, hogy miért éppen azt kaptuk, amit kaptunk). Ha nincs türelmünk kivárni egy teljes futást, úgy vagy vegyük lejjebb a generációk maximális számát, vagy szakítsuk meg a futást Ctrl+C gombokkal.

2.2. mérési feladat

Cseréljük le a `main.m` függvény `case 'netw'` részében a fitness függvényt az `f_neuralw2.m` függvényre, és futtassuk újra az algoritmust! *Milyen változást tapasztalunk az előbbiekhez képest, és miért? Érdemes neurális hálózatokat genetikus algoritmusokkal tanítani? Ha igen, mikor? Ha nem, miért nem?*

2.3. mérési feladat (szorgalmi)

Kísérletezzünk más neurális háló struktúrákkal és problémákkal!

3. MÉRÉSI FELADAT: struktúra-optimalizálás

Folytassuk tovább a neurális hálózatokkal kapcsolatos vizsgálódásainkat, és próbáljuk most meg a hálózatok struktúráját is optimalizálni. Az adott struktúrájú neurális hálózatok tanítására immár ne genetikus algoritmust, hanem dedikált tanító eljárást használjunk (Levenberg-Marquardt). Mindez akkor lehet hasznos, ha például adott probléma (pl. mérési adatok) kapcsán nem egyértelmű, hogy mi lenne a legoptimálisabb struktúrájú neurális hálózat. *Adott bemenet és elvárt kimenet esetén melyik neurális háló struktúra képes az adott problémát a lehető legkisebb általánosítási hibával megtanulni?*

Amennyiben a neurális hálókat szokványos tanító algoritmussal tanítjuk, és csak a struktúrájukat kell kódolnunk, úgy az előbbi feladatnál egyszerűbb a dolgunk: most pusztán csak az egyes rétegekben rejlő neuronokat kell kódolnunk. Kromoszómáink hossza tehát a rejtett rétegek maximális számával lesz azonos, míg a génjeink értékkészlete az egyes rejtett rétegekben található neuronok számát fogja meghatározni. Ha megengedjük a 0 értéket is, akkor dinamikusan változhat a rejtett rétegek száma.

3.1. mérési feladat

Tekintsük át a `main.m` függvény `case 'nets'` részét, és értelmezzük az ott látottakat! *Milyen beállításokkal milyen problémát oldunk meg? Hol történik a 0-elemű rétegek kiszűrése?*

3.2. mérési feladat

Az alapbeállításokat használva futassuk az algoritmust:

```
main('nets');
```

Értékeljük az eredményeket (magyarázzuk meg, hogy miért éppen azt kaptuk, amit kaptunk). *Miért annyival lassabb a futás, mint a 2-es feladatban? Miért nem monoton nem-csökkenő a legjobb egyed fitness-e az algoritmus futása során? Mivel magyarázza az `f_neurals.m` függvényben az `fval(i, 1)` számításánál az egyenlet jobb oldalán szereplő tört nevezőjében lévő járulékos tagot?*

3.3. mérési feladat (szorgalmi)

Kísérletezzünk különböző beállításokkal és problémákkal! *Mire milyen megoldást ad az algoritmus? Mennyire tekinthető stabilnak egy-egy futás eredménye?*

4. MÉRÉSI FELADAT: órarendkészítés

A mérés hátralévő részében egy mindnyájunk számára ismert valós probléma megoldásával foglalkozunk: genetikus algoritmussal szeretnénk optimalizálni a kari órarendet. Ezt a feladatot jelenleg gyakorlatilag egyetlen ember végzi a karon, és bizony, amint ezt a későbbiekben látni fogjuk, feladata nem egyszerű.

A kari órarendben előadások szerepelnek (tól-ig), az előadások kurzusokhoz tartoznak (egy kurzusnak több előadása is lehet egy héten), a kurzusok pedig tárgyakhoz tartoznak, amiket különböző tanszékek hirdethetnek meg. A különböző kurzusoknak különböző előadóik lehetnek. Egy tárgynak lehetnek gyakorlati, elméleti, és labor típusú kurzusai. Az egyes kurzusokra jelentkező hallgatók számát figyelembe kell venni a kurzusokhoz tartozó előadások tantermekbe való beosztásánál.

Az órarendek értékelésekor kétféle szempontot veszünk figyelembe: **(1) elsődleges szempontok** (pl. *ütköznek-e előadások?; van-e olyan előadás, amelyik olyan terembe lett beosztva, ahol nem fér el?; van-e olyan előadás, amelyik túl korán kezdődik, vagy túl későn fejeződik be?; beosztottuk-e valamely tanárt egy időben több helyre?*), illetve **(2) másodlagos szempontok** (*mennyire hatékony az előadások terem-kihasználása?; mennyi üresjárat van az órarendben a diákok és a tanárok számára; mennyire vannak közel helyileg az egymás utáni előadások?; stb*).

Az elsődleges szempontoknak mindenképp teljesülnie kell ahhoz, hogy egy órarendet elfogadhassunk, míg a másodlagos szempontok már csak „hab a tortán”.

4.1. mérési feladat

Tekintsük meg a méréshez mellékelt órarend táblázatot (`orarend.xls`), melyben a 2009/2010-es tanév tavaszi félévének villanykari B.Sc. órarendje, és az azzal kapcsolatos tételek (tárgyak, termek, stb) szerepelnek. *Pontosan milyen tételek szerepelnek a táblázatban, és milyen kapcsolatban állnak egymással?*

4.2. mérési feladat

Tegyük föl, hogy a kari órarend felelős személy helyében vagyunk, és már csak 5 db. előadást kell elosztanunk a félév kezdete előtt a Neptunban. Már csak 2 db. teremmel gazdálkodhatunk, és azok is már csak hétfőn reggel 8-tól kora délutánig szabadok (**8:15-től 10:15-ig kezdődhet bennük előadás, legfeljebb órás gyakorisággal**). Vegyük figyelembe a kurzusokra jelentkezett hallgatók számát, a termek kapacitását, és az előadások hosszát! A következő előadásokat kell beosztanunk:

Tárgy neve/kódja: Bevezetés a számításelméletbe 2. (BMEVISZA110)
Előadás típusa: előadás
Kurzus kódja: 1
Hallgatók száma: 500 fő
Előadás hossza: 2 óra

Tárgy neve/kódja: A folyamatirányítás és –terv. gyak. módszertana (BMEVIIIJV81)
Előadás típusa: előadás
Kurzus kódja: SZV-E
Hallgatók száma: 50 fő
Előadás hossza: 4 óra

Tárgy neve/kódja: A programozás alapjai 1. (BMEVIHIA106)
Előadás típusa: előadás
Kurzus kódja: E
Hallgatók száma: 300 fő
Előadás hossza: 2 óra

Tárgy neve/kódja: A programozás alapjai 1. (BMEVIHIA106)
Előadás típusa: gyakorlat
Kurzus kódja: G0
Hallgatók száma: 30 fő
Előadás hossza: 1 óra

Tárgy neve/kódja: A programozás alapjai 1. (BMEVIHIA106)
Előadás típusa? gyakorlat
Kurzus kódja: G1
Hallgatók száma: 30 fő
Előadás hossza: 1 óra

Az előbbi előadásokat a következő 2 terembe osszuk be:

Terem neve: I.B.028
Férőhelyek száma: 500

Terem neve: E.602
Férőhelyek száma: 50

Írjuk fel időben növekvő sorban az egyes **időpont (tól-ig)–terem–előadás** hármasokat!

4.3. mérési feladat

Az előbbi feladatban láthattuk, hogy végső soron azért mégsem annyira triviális még egy ilyen egyszerűbb órarend elkészítése sem. Képzeljük csak el, hogy a karon a B.Sc. képzés keretein belül a 2009/2010-es tanév tavaszi félévében összesen 862 db. ilyen előadást kellett beosztani a villamosmérnök és informatikus hallgatók számára! Próbáljunk most ezért gépi megoldást találni a problémára – oldjuk meg a kari órarend-készítés feladatát genetikus algoritmussal! Futassuk az algoritmust egymás után többször alapbeállításokkal:

```
main('tmt');
```

Miképpen viszonyulnak a kapott órarendek az Ön által előbb manuálisan előállítotthoz, illetve egymáshoz? Jobbak, rosszabbak? Miért? Mennyire megbízható az algoritmus a jelen beállítások mellett (futási idő, eredmény szempontjából)?

4.4. mérési feladat

Tekintsük át a `main.m` függvény `case 'tmt'` részét, és értelmezzük az ott látottakat! *Honnan érkezik az előbb kapott órarendben szereplő információ (tárgyak, tanárok neve, tantermek, stb)?*

4.5. mérési feladat

Tekintsük át az `f_timetable_params.m` függvény felépítését, és működését! *Mire jó ez a függvény, hol használjuk, és lényegét tekintve hogy működik?*

4.6. mérési feladat

Írjuk át a `main.m` függvény `case 'tmt'` részét úgy, hogy immár ne az 1-es, hanem a 2-es számú problémára fusson le az algoritmus, majd futtassuk is le! *Mi a különbség a 1-es és a 2-es probléma, illetve a rájuk adott megoldások között?*

4.7. mérési feladat

Mostanra már sejthetjük, hogy az órarendek kódolása a következő: a kromoszómák hossza gyakorlatilag az előadások számával ekvivalens, mivel minden egyes gén a megfelelő előadás időpontját és helyét kódolja. Az időpontok, és a helyek is 1-től induló sorszámmal (ID-vel) rendelkeznek. A két azonosító szorzata (kezdés idejének ID-je * hely ID-je) szerepel a génekben. Osztással, és maradékképzéssel a szorzat faktorizálható a megfelelő összetevőkre, így nem kell összetett génekkel, vagy egyszerre több kromoszómával dolgoznunk egy-egy egyed kapcsán.

A kromoszómák visszafejtését leginkább a `disp_timetable.m` függvényben figyelhetjük meg, amelyet a `main.m` függvény `case 'tmt'` részének végén hívunk meg abból a célból, hogy kilistázza a genetikus algoritmus futása eredményeképp előállt legjobb egyedet. **Tekintsük át, és értsük meg ezt a függvényt!**

4.8. mérési feladat

Futtassuk az algoritmust a teljes kari órarend problémára (3-as számú eset). *Mit tapasztalunk, és mi következik ebből?*

4.9. mérési feladat

Vessünk egy pillantást az `f_timetable.m` függvényre, amely az órarend-populációk fitness-ének kiértékeléséért felelős. Vegyük észre, hogy a függvényben kétféle fitness adódik össze: **(1) primary**, és **(2) secondary**. *Mi ennek az oka és értelme?*

4.10. mérési feladat (szorgalmi)

Egészítsük ki az algoritmust egy újabb másodlagos kritérium figyelembevételével! Egészítsük ki az `f_timetable.m` függvényt egy második (`fs2`) secondary fitness faktorral, amely azon előadások számát jelöli egy-egy kromoszóma kapcsán, amelyek végpontja egy megadott időpontnál előbb van. A faktort készíthetjük az `fp2` faktor mintájára, és úgy vegyük bele a végső fitness-be, hogy maximalizálva legyen. Futtassuk ezek után a 2-es számú problémára az algoritmust, és vizsgáljuk meg, hogy ésszerű időpont limit (pl. hétfő 19 óra) mellett miképpen változik a beosztás!