ELTE

GENETIKUS ALGORITMUSOK
SPECI ANYAGA

Készítette:
Salamon András

2003. Január 30.

Kivonat

A jegyzet a Genetikus Algoritmusok című speci előadásait fogja össze. Az első fejezet megpróbálja elhelyezni a genetikus algoritmusokat a számítástechnikán belül. A biológia játékokat tartalmazó bevezetés után az egyszerűbb optimalizálási módszereket tekinti át, végül a genetikus algoritmusokat is tartalmazó evolúciós algoritmusokat ismerteti.

A második fejezet mutatja be a genetikus algoritmusokat. A kanonikus GA ismertetése után, az algoritmus egyes elemeit vizsgálja meg alaposabban. A fejezet végén a matematikai háttér alapjai is megtalálhatóak.

A következő fejezet az eredeti GA algoritmus továbbfejlesztéseiről szól. Az ismertetett módszereket használva az algoritmus képes az eredetinél bonyolultabb problémák sikeresebb megoldására is. A fejezet tartalmazza az utazóügynök probléma számos különböző reprezentációt használó megoldását is.

A negyedik fejezet a korábbi fejezetekben leírt elméletre épülő alkalmazásokat ismertet. A bemutatott problémákra példánként egy-két eltérő elven működő genetikus megoldást mutat a teljesség igénye nélkül. A bemutatott megoldások nem feltétlenül a legsikeresebbek, de megmutatják, hogy a korábban ismertett elméleti módszerek többségét miként lehet a gyakorlatban alkalmazni.

Az utolsó fejezet a GALOPPS nevű GA csomagot mutatja be. A csomag egyike azon GA csomagoknak, melyek ingyenesen elérhetőek és segítséget nyújtanak, ha egy problémát genetikus algoritmusok segítségével szeretnénk megoldani.

Tartalomjegyzék

1. fejezet

Biológiára épülő algoritmusok

A biológiában több olyan jelenséggel találkozhatunk, melyet érdemes lemásolni, utánozni. Elég csak a madarak repülését nézni, egyes élőlények hőszigetelési képességeit, vagy a gepárd futását megvizsgálni. Érdekes dolgokat vehetünk észre, ha az egyes állatok helyett állatok populációját vizsgáljuk. A populáció egyazon fajba tartozó, meghatározott időben és területen együtt élő, egymással kereszteződő egyedek csoportja [Moh96]. A populációban lévő egyedek hatással vannak egymásra, néha segítik, néha akadályozzák egymást. PopulációbiológiáróI bővebben a [SZ95] jegyzetben olvashatunk.

Most két olyan szimulációs „játékot” vizsgálunk meg, melyekben élőlénypopulációk változását figyelhetjük meg. Mindkét játék azt próbálja demonstrálni, hogy bár a szimulált élőlények viselkedését nagyon egyszerű szabályok határozzák meg, az előlénypopulációk megfigyelésével mégis olyan jelenségeket észlelhetünk, melyeket — csak a szabályok ismerete alapján — nem tudtunk volna megjósolni.

Erre azért van szükség, mert a — gyakorlatban sikeresen alkalmazható — genetikus algoritmusok alapjai is egyszerű szabályok. A következő rész bebizonyítja, hogy érdemes egyszerű biológiai szabályokra épülő algoritmusokkal foglalkozni.

1.1. Életjáték

A *Conway* által kitalált életjáték az egyik legegyszerűbb számítógépes szimuláció. Az életjáték egy olyan egyszerű biológiai szimuláció, melyben a szimulált élőlények mozdulatlanok, állapotukat nem változtatják (kivéve a születést és a halált). Az egyszerű szabályok miatt az életjáték könnyen implementálható számítógépen, problémát csak az okoz, hogy az élettér elméletileg végtelen nagy, ami nyilván nem valósítható meg számítógépen.

Az életjáték egy nagyon jó számítógépes megvalósítása a [Hen] shareware program. A program több érdekes példát is tartalmaz, melyekből jobban megismerhetjük a játékot.

1.1.1. Szabályok

A játéktér egy végtelen nagy sakktábla. A sakktábla mezőit egyformának tekintjük, tehát nem teszünk különbséget a mezők színe alapján. Minden mezőnek 8 szomszédja van.

A sakktábla mezőin egysejtűek tenyésznek, egy mező lehet üres, vagy foglalt, egy mezőn maximum egy egysejtű lehet. Nincs meghatározva az, hogy kezdetben melyik mezőn van egysejtű, és melyiken nincs, tetszőleges módon helyezhetjük el őket. A [Hen] számítógépes programban lehetőségünk van a kezdeti állást file-ból beolvasni, vagy a képernyőn interaktívan elhelyezni az egysejtűeket.

Egy szimulációs lépés során születhetnek új egysejtűek, a régi egysejtűek közül páran meghalhatnak. A születés és a halál a következő szabályok szerint történik:

- Ha egy mező üres, a következő lépésben is üres marad, kivéve, ha pontosan 3 szomszédos mezője foglalt.
- Ha az üres mezőnek pontosan 3 szomszédos mezőjén van egysejtű, akkor a következő lépésben új egysejtű születik a mezőn.
- Ha a mezőn egysejtű van, akkor az egysejtű a mezőn marad a következő lépésben is, ha a mezőnek 2 vagy 3 szomszédján van egysejtű.
- Ha egy egysejtű szomszédos mezőin 2-nél kevesebb, vagy 3-nál több egysejtű van, akkor a mezőn található egysejtű az elszigeteltség, illetve a túlnépesedés miatt meghal.

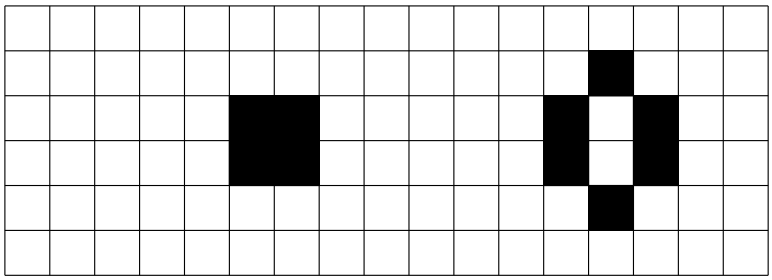
A fenti paraméterek természetesen módosíthatóak, de általában ezt a paraméterezést szokták alkalmazni. Conway is kísérletezett különféle paraméterezéssel, de úgy találta a többi esetben túl kiismerhető a szimuláció.

A játék egyes számítógépes megvalósításaiban egy időkorlátot is bevezetnek, egy sejt csak a megadott ideig élhet, az idő leteltekor akkor is meghal, ha továbbra is 2 vagy 3 szomszédja van. Ez a korlátozás teljesen megváltoztatja a játékot, ezért továbbiakban feltételezzük, hogy nincs ilyen korlátozás.

1.1.2. Alakzatok

Könnyen nyomon tudjuk követni az egysejtűeket, ha a számítógépes szimuláció során őket fekete, az üres mezőket fehér négyzettel jelöljük.

Ha a szimuláció elég gyors, és elég nagy területet figyelünk egyszerre, már nem tudjuk megkülönböztetni az egyes egysejtűeket, csak csoportjaikat tudjuk nyomon követni. Leginkább gyorsan, és látszólag össze-vissza változó csoportokat látunk a szimuláció során, de észrevehetünk érdekes csoportokat is. Ezentúl ezeket az egysejtű csoportokat alakzatnak nevezzük. Az alakzat egysejtűi nem feltétlenül érintkeznek fizikailag, de vizuálisan együvé tartozónak érezzük őket.



1.1. ábra. Mozdulatlan alakzatok

Most néhány érdekesebb alakzatot mutatunk be, természetesen a teljesség igénye nélkül. A bemutatandó alakzatok eleinte egyszerűek, majd egyre bonyolultabbak lesznek. Az egyszerűbb alakzatoknál könnyen ellenőrizhető az, hogy tényleg a leírtak szerint viselkednek, a bonyolultabb alakzatok ellenőrzéséhez számítógépes program javasolt. A [Hen] program segíti az ellenőrzést azzal, hogy lehetőséget nyújt a szimuláció lépésenkénti futtatására. (A lépések előtt gombnyomásra vár, így minden lépés után jól megfigyelhető a játéktér.)

Mozdulatlan alakzatok

A legegyszerűbbek a mozdulatlan alakzatok (1.1. ábra), vagyis azok az alakzatok, melyek nem változnak meg az idő múlásával. (Hacsak egy másik, hozzájuk közel kerülő alakzat meg nem változtatja őket.) A gyorsan változó játéktéren könnyen észrevehetőek, úgy tűnik, mintha nem vonatkoznának rájuk a szabályok. Egy ilyen alakzatot megvizsgálva már érthető a mozdulatlanságuk.

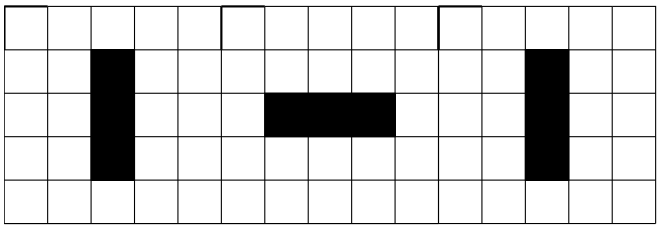
A 1.1. ábrán bal oldalon látható alakzatnál (melyet blokknak neveznek) az összes egysejtűnek 3 egysejtű-szomszédja van, így ők a szabályok alapján nem halnak meg. Az alakzattal szomszédos üres mezőknek 1 vagy 2 egysejtű-szomszédjuk van, így új egysejtű sem születhet.

A jobb oldali ábrán (méhkas) minden egysejtűnek 2 egysejtű-szomszédja van, a két üres belső mezőnek 5 (túl sok), a külső üres mezőknek 1 vagy 2 (túl kevés) egysejtű-szomszédja van.

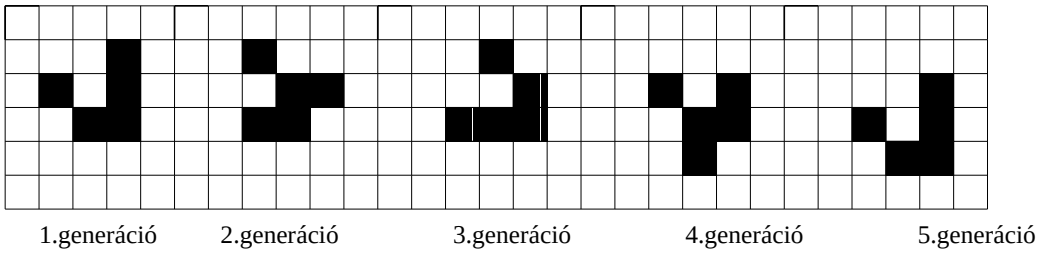
Oszcilláló alakzatok

Könnyen felismerhetőek az oszcilláló alakzatok is, melyek változnak ugyan az idő múlásával, de egy megadott periódusidő elteltével az eredeti alakzatot kapjuk vissza. A periódusidő nagysága a különféle alakzatoknál eltérő lehet, a legegyszerűbb ilyen alakzatnál, a *villogónál* (1.2. ábra) például 2.

A villogó 3 darab, sorban elhelyezkedő egysejtűből áll. Nézzük meg, mi történik egy szimulációs lépés során. A középső egysejtűnek két egysejtű-szomszédja van,



1.2. ábra. A villogó



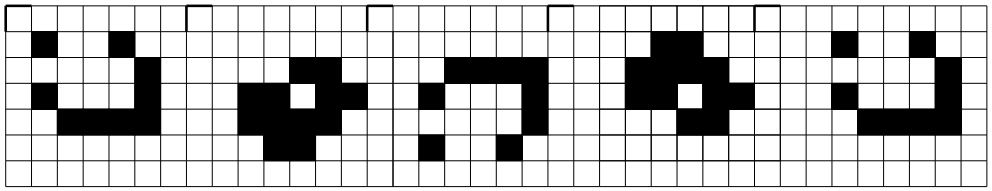
1.3. ábra. A sikló mozgása

ezért nem hal meg. A két szélső egysejtűnek azonban csak 1-1 egysejtű-szomszédja van, ezért azok meghalnak. A középső egysejtű mellett lévő üres mezők közül kettőnek pontosan 3 egysejtű-szomszédja van, ezért két új egysejtű születik a két mezőn. Könnyen ellenőrizhető, hogy a következő lépésben, a meghalt egysejtűek helyén új egysejtűek születnek, az előző lépésben született egysejtűek azonban meghalnak, így újra visszakapjuk a kezdeti alakzatot.

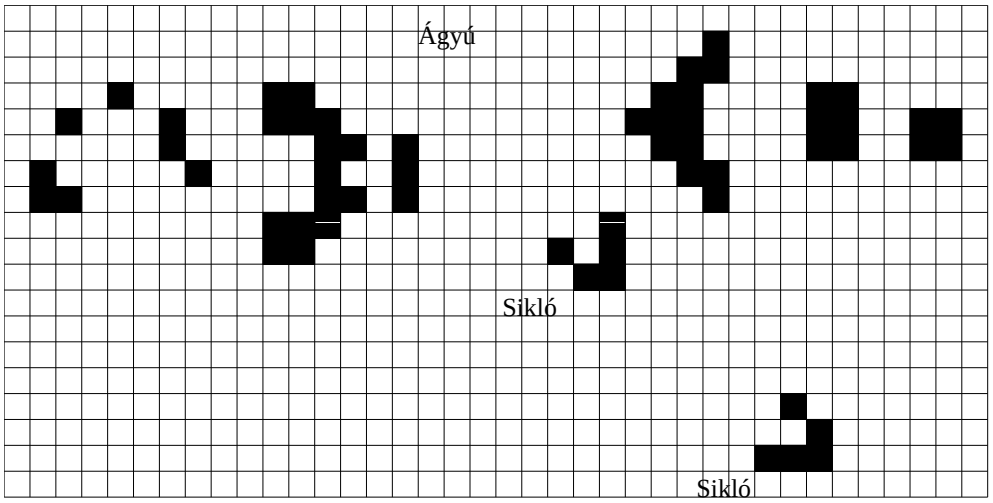
Mozgó alakzatok

Az egyik legérdekesebb alakzat, amit könnyen felfedezhetünk, az a *sikló* (1.3. ábra). A *sikló* egy olyan alakzat, amelyik az oszcilláló alakzatokhoz hasonlóan, egy adott periódusidő elteltével visszanyeri az alakját, de eközben elmozdul a játéktéren. Az 1.3. ábrán látható a *sikló*, mely 4 időegységenként egy egységgel jobbra-le mozdul el a játéktéren. Ha a szimuláció elég gyors, úgy tűnik mintha egy kis állat futna (siklana) át a képernyőn. Természetesen az eredeti alakzatot elforgatva, az elmozdulás a másik 3 átlós irányba is megtörténhet.

Találhatunk olyan alakzatot is, mely a siklóhoz hasonlóan mozog a játéktéren, de a mozgás nem átlós irányban történik, hanem függőlegesen vagy vízszintesen. A legkisebb ilyen alakzat az LWSS (Light Weight SpaceShip = Könnyű űrhajó) (1.4. ábra). Az alakzat a nevét onnan kapta, hogy egy űrhajó repüléséhez hasonlít a mozgása. Az ábrán látható LWSS vízszintesen mozog, de az alakzatot elforgatva könnyen megkaphatjuk a függőlegesen mozgó LWSS-t.



1.4. ábra. LWSS működés közben



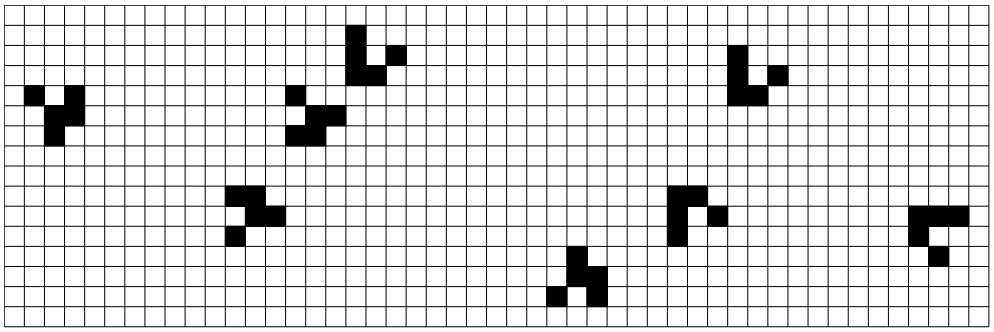
1.5. ábra. A siklóágyú működés közben

A siklóágyú

Az eddig vizsgált alakzatok oszcillálnak, mozognak, de nem képesek elszaporodni. Eleinte nem tudták megválaszolni azt a kérdést, van-e olyan alakzat, mely korlátlanul növekszik, vagyis az általa lefoglalt mezők száma minden határon túl növekszik. (Ez nem jelenti feltétlenül azt, hogy a mezők száma a végtelenbe tart, hiszen a konvergencia nem biztosított)

Conway 50 dollár jutalmat tűzött ki a feladatra, és rögtön adott egy tippet is: Egy *siklóágyút* kell alkotni, vagyis egy olyan alakzatot, mely végtelen számú siklót bocsájt ki magából. Az alakzatot végül a **MIT**[1] diákjai találták meg. Egy ilyen alakzatot láthatunk a 1.5. ábrán. Az alakzat nevét rögtön megértjük, ha működés közben figyeljük meg. Olyan, mintha egy hatalmas ágyú időnként kilőne egy-egy siklót, melyek a kilövés után gyorsan repülnének, minél messzebb az ágyútól. Az ábrán azt a pillanatot láthatjuk, amikor már két siklót kibocsátott az ágyú. A két sikló jobbra-le repül, a fent látható (a két sikló egysejtűit leszámítva az összes ábrán látható egysejtű az ágyú része)

[1]Massachusetts Institute of Technology



1.6. ábra. Siklóágyú születése

ágyú pedig a következő siklót gyártja.

Látható, hogy bár a siklóágyú segítségével egyre több cella lesz foglalt, a növekedés csak lineáris, és az élettérnek csak egy kis részén lesznek foglalt cellák (az ágyú területén, és a siklók vékony sávjában). Ismert azonban olyan alakzat, ahol a növekedés négyzetes, olyan ahol exponenciális. Mivel a játéktér is végtelen nagy, az élettér lefedettsége többnyire az exponenciálisan növekedő alakzatoknál is alacsony. Ismert azonban olyan alakzat is, ahol a lefedettség a 1/2-hez tart. Ilyen, és ezeknél még érdekesebb alakzatokat találhatunk a [Hen] shareware programban.

Ha megnézzük a siklóágyút (1.5. ábra), akkor egy olyan bonyolult alakzatot látunk, melyről fel sem tételezzük, hogy véletlenül is létrejöhet. A **MIT** diákjai, azonban meglepő felfedezést tettek: 13, megfelelően elhelyezett sikló ütközéséből siklóágyú születhet. A 1.6. ábrán egy egyszerűbb változatot látunk, itt elég 8 sikló ütköztetése a siklóágyú születéséhez. Tehát, ha az ábrán látható módon elhelyezünk 8 siklót, akkor a siklók összeütköznek, és az összeütközésekből létrejön egy siklóágyú, ami rögtön el is kezd siklókat kibocsátani magából.

Önreprodukáló automata

Az életjáték a sejtautomaták közé tartozik. A sejtautomaták megjelenése után szinte rögtön az önreprodukáció problémájával kezdtek el foglalkozni a kutatók. Olyan automatát próbáltak készíteni, mely képes lemásolni önmagát, így bizonyos szempontból mesterséges élőlénynek tekinthetjük. Már *Neumann János* is készített ilyen automatát, 29 különböző állapotot vehettek fel a sejtek, az automata kb. 200’000 sejtből állt. Később persze egyre kisebb önreprodukáló automatát készítettek, a rekordot egy 12 sejtből álló automata tartja, a sejteknek 6 különböző állapota, és 57 állapotátmeneti szabálya van.

Közös jellemzője ezeknek az automatáknak, hogy már a szabályok megalkotásakor figyelembe vették, hogy az automata célja az önreprodukció. Sokkal életszerűbb az életjáték, mely sokkal egyszerűbb szabályokkal rendelkezik, melyek megalkotásakor nem vették figyelembe az önreprodukciót.

A *Conway* által leírt önreprodukáló automata egy megfelelő programmal ellátott univerzális számítógép lesz, hasonlóan *Neumann* automatájához.

Négyféle alakzat elég az önreprodukáló automata felépítéséhez: sikló, siklóágyú, blokk, mindenevő. A mindenevő egy olyan alakzat, amely a nekiütköző siklót mintegy „megeszi”. Ezt az alakzatot használhatjuk a felesleges siklók eltüntetésére.

Két sikló összeütközése létrehozhat blokkot, mindenevőt, és összeütközhetnek úgy is, hogy mindkét sikló eltűnik (*Eltüntető ütközés*). Az ún. *Visszaverő ütközés*nél az egyik sikló eltűnik, a másik visszafordul. Siklók összeütközésével tehát mind a négyféle alakzatot elő lehet állítani.

Ha két megfelelően elhelyezett sikló nekimegy egy blokknak, akkor a blokkot 3 mezővel közelebb hozza. Harminc (30) megfelelő sikló, a blokkot három mezővel eltolja. Ezt felhasználva egyszerű memóriát tudunk készíteni, ahol egy blokk távolsága határozza meg a tárolandó számot.

A számítógép felépítéséhez huzalokra, és logikai kapukra lesz szükségünk. Mivel a sejtek két dimenzióban vannak, a huzalok keresztezése problémát okozhat.

Siklókat használhatunk huzalnak, a 0 és 1 számjegyekkel kódolt — önreprodukáló automatát leíró — információt siklók áramlatával tudjuk elküldeni.

Bár siklóágyúkkal siklókat könnyen tudunk gyártani, mivel az ágyúink elég gyakran tüzelnek, tartani kell a siklók összeütközésétől. Szerencsére három siklóágyúból *ritkított siklóágyú* készíthető, amely tetszőleges frekvenciával tüzelhet. Ezzel megoldottuk a huzalok keresztezésének problémáját.

Ha ritkított az áramlat akkor az üzenetek keresztezhetik egymást. A logikai kapukat is a négy alapalakzatból építjük meg. A *nem*-kapu egy ágyút, az *és*-kapu egy ágyút és egy mindenevőt tartalmaz.

A belső memória egy zárt hurokban keringő nagyon hosszú üzenet. A külső memória nem egy végtelenül hosszú papírszalag, hanem tetszőlegesen nagy szám. Az ilyen számoknak egy-egy blokk alakzat géptől való távolsága jelenti.

A további technikai problémák kiküszöböléséről (pl. huzalok meghajlítása) is a [Hen] könyvben olvashatunk.

Az univerzális számítógép *hardware*-e megkapható siklók összeütközésével. Négy különböző irányból érkező siklóflotta képes létrehozni a gépet.

1.2. A Róka-Nyúl játék

Az ebben a részben ismertetendő játék több néven ismert, például Hiúz-Nyúl, Macska-Egér, Cápa-Hal.

1.2.1. Szabályok

A játék lényege, hogy az élettéren egy ragadozó és egy préda állatfaj egyedei élnek. A példánkban természetesen a róka a ragadozó, a nyúl a préda. Az élettér különböző

alakú lehet, többnyire az élettér egy nagy sakktábla. Egy mezőn egyszerre csak egy állat lehet.

A nyulak véletlenszerűen mozognak a játéktéren (mindig valamelyik szomszédos mezőre lépnek), de nem ütközhetnek össze. A nyulak természetesen szaporodni is képesek, ez úgy történik, hogy egy lépés után az eredeti helyen hagynak egy újszülött nyulat. A játékban az egyszerűség kedvéért nem kap szerepet a nyulak neme, így a szaporodáshoz sem kell két nyúl. Természetesen nem minden lépés után születik új nyúl. Előre adott, hogy a lépések hány százalékánál születik új nyúl. Ez a százalék fejezi ki a nyulak szaporodási képességét.

A rókák a nyulakhoz hasonlóan mozoghatnak, szaporodhatnak (a rókák szaporodási képessége különbözhet a nyulak szaporodási képességétől), de összeütközhetnek a nyulakkal, sőt ez a céljuk, az összeütközés során a róka megeszi a nyulat. Ha egy róka hosszabb ideig nem találkozik nyúllal, nem jut táplálékhoz és elpusztul. (A nyulak füvet esznek, ami bőven található a játéktéren, így ők sohasem halnak éhen.)

1.2.2. A populációk változása a szimuláció során

Ha megvizsgáljuk a rókák és a nyulak számának változását, oszcillációt figyelhetünk meg. A nyulak létszáma időben ingadozik, és ezt az ingadozást szorosan követi a rókák létszámának ingadozása. Miért van ez? Képzeljük el, hogy a nyulak nagyon elszaporodnak az élettérben. Ekkor a rókák bőven jutnak táplálékhoz, ezért elszaporodnak. A több róka azonban nagyon sok nyulat megeszik, így a nyulak létszáma lecsökken. A kevés nyúl nem ad elég táplálékot a sok rókának, így a rókák kezdenek éhen halni. Miután így lecsökken a rókák száma, az életben maradt nyulakat nem fenyegeti nagy veszély, így el tudnak szaporodni, és ezzel visszaértünk a kezdeti állapothoz. Az igazsághoz tartozik, hogy ezt az oszcillációt megkapjuk, jól kell beállítani a rókák és nyulak szaporodási és egyéb paramétereit. Ha a rókák túl nehezen szaporodnak, és túl hamar éhen halnak, elképzelhető, hogy a rókák még nagyon sok nyúl között sem képesek elszaporodni, ezért kihalnak. Ha a nyulak szaporodnak túl lassan, és a rókák nagyon jól bírják az éhezést, elképzelhető, hogy a rókák elszaporodásukkor kipusztítják az összes nyulat, és ezzel önmagukat is halálra ítélik, mert ezután az összes róka éhen hal.

1.2.3. A populációk változása a Földön

A szimuláció során a természetet próbáltuk utánozni, a természetben látott példák alapján határoztuk meg a szimuláció szabályait. Felmerül a kérdés, mennyire sikerült utánozni a természetet, a valóságban is így oszcillál-e a préda és a ragadozó állatfajok populációjának nagysága.

A kérdés megválaszolását nehezíti, hogy a laboratóriumi körülményeket leszámítva nem találunk olyan területet, ahol csak két állatfaj él, az egyik egy préda, a másik egy csúcsragadozó. Az oszcilláció igazolására legtöbbször a Hudson Öböl Társaság részére

csapdába ejtett hiúzok és nyulak számának ingadozását használják fel. A társaság több mint két évszázadon keresztül vezetett feljegyzést a prémvadászok zsákmányáról, és adatok valóban a várt módon ingadoznak (tízéves periódussal). Bár legtöbben elfogadják bizonyítéknak az adatokat, vannak akik figyelmeztetnek, hogy a prémvadászok elsősorban nem az élővilágot szerették volna megfigyelni, hanem megélhetésük miatt ejtették csapdába az állatokat. Így elképzelhető, hogy az ingadozás nem ökológiai, hanem gazdasági ciklusokat követ.

1.2.4. A játék továbbfejlesztései

A játék több irányban is továbbfejleszthető. Az első lehetőség egy bonyolultabb tápláléklánc szimulálása. Tehát nem csupán 1 préda és 1 ragadozó állatfajunk lenne, hanem lennének préda állatfajok, lennének olyan fajok melyek a prédaállatokkal táplálkoznának, ugyanakkor zsákmányai lennének további állatokfajoknak, és így tovább egészen a csúcsragadozókig, melyeket már nem fenyegetne másik állatfaj. A bonyolultabb tápláléklánc szimulációjáról, illetve a valóságban megtalálható bonyolultabb táplálékláncokról a [Sig93] könyvben olvashatunk többet.

Egy másik továbbfejlesztési lehetőség, ha az állatokat felruházzuk intelligenciával, a ragadozókat különféle keresési stratégiával, a prédákat menekülési módszerekkel. Ha a különböző stratégiával rendelkező állatokat helyezünk el az élettéren, megfigyelhetjük melyik stratégia jobb, melyik rosszabb.

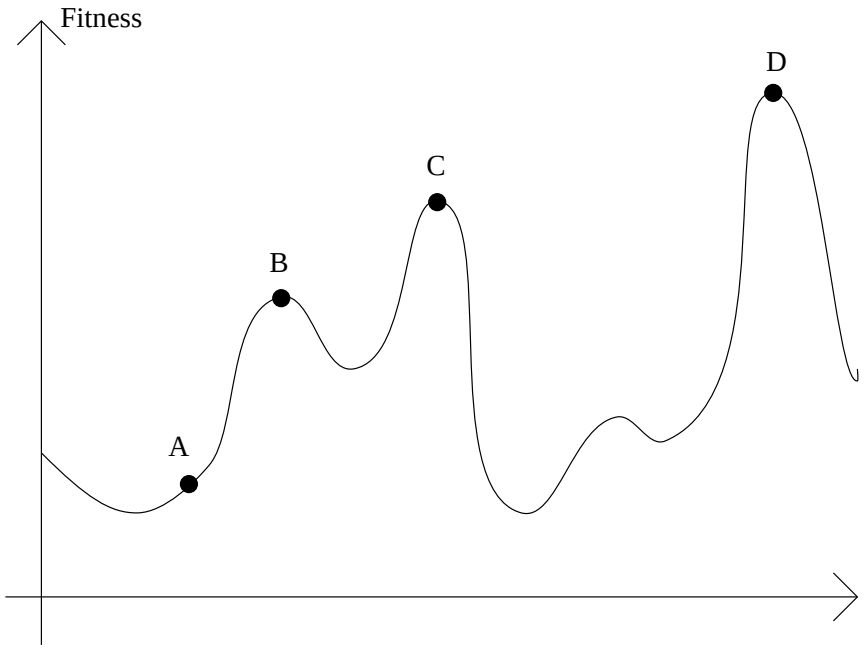
1.3. Optimalizálási feladatok

Mivel a dolgozat hátralévő részében ismertetendő algoritmusokat az esetek nagy részében optimalizálási feladatok megoldására használják, definiálnunk kell mit értünk optimalizálási feladaton. Ezután néhány hagyományos módszert mutatunk be, melyeket ilyen feladatok megoldására használnak.

Az optimalizálási feladatok során egy adott halmazon (melyet keresési térnek neveznek, és S-sel jelölünk) definiált függvény (fitneszfüggvénynek nevezik, f-fel fogjuk jelölni, $f : S \to \mathbf{R}$) maximumhelyét (vagy minimumhelyét) keressük. Az egyszerűség kedvéért a továbbiakban mindig maximumot keresünk.

1.3.1. Véletlen keresés

A legegyszerűbb keresési módszer, ha a keresési térből véletlenszerűen (esetleg szisztematikusan, de a kapott eredményektől függetlenül) választunk ki pontokat, és nézzük meg a fitneszfüggvény értékét az adott pontban. A megvizsgált pontok közül azt a pontot választjuk megoldásnak, amelyik pontban a fitneszfüggvény értéke a legmagasabb. A módszer önmagában nem alkalmazható a gyakorlatban.



1.7. ábra. A hegymászó módszer

1.3.2. Hegymászó módszer

A hegymászó, más néven gradiens módszer során kiválasztunk egy véletlen pontot a keresési térben, majd megnézzük a kiválasztott pont szomszédait, és közülük mindig a legmagasabb fitneszértékű pontot választjuk következő megvizsgálandó pontnak, hasonlóan egy ködös időben a csúcsra feljutni vágyó hegymászóhoz, aki mindig meredeken felfelé mászik. A módszer megtalálja a maximumot az egy csúccsal rendelkező fitneszfüggvényeknél, a több csúccsal rendelkezőeknél pedig lokális maximumot talál. Az 1.7. ábrán egy egyszerű példát láthatunk a hegymászó módszerre. Bár a módszer látványosabb, ha S több dimenziós, az egyszerűbb ábrázolás miatt a példában S 1 dimenziós. Az ábrán az **A** pontból indítva a keresést, a módszer megoldásként a **B** pontot adja, ami csak lokális maximum, hiszen mind a **C**, mind a **D** pontokkal jelölt csúcs magasabb nála.

1.3.3. Iterált hegymászó módszer

Az előbb megismert két módszer kombinációjaként kaphatjuk meg ezt a módszert. A keresési térben véletlenszerűen választunk ki pontokat, és a pontokban egy hegymászó keresést indítunk el. A hegymászás után új pontot választunk, és újraindítjuk a hegymászó módszert. A módszer az egyes hegymászások után kapott lokális maximumok maximumát adja eredményül. Látható, hogy a módszer, bár jobb az előzőeknél, továbbra sem őriz meg információt a már felderített keresési térről.

1.3.4. Szimulált lágyítás

A hegymászó módszer egy másik továbbfejlesztése a népszerű szimulált lágyítás. Nevét arról a folyamatról kapta, melynek során egy fémet először felhevítenek, majd fokozatosan lehűtenek. A módszernek több változata van, egy rövid összefoglalást olvashatunk róla a [Yao95] cikkben.

A keresési tér egy pontjában állva, véletlenszerűen választjuk meg a lépés irányát. Ha a lépéssel magasabb helyre jutunk, akkor megtesszük a lépést, míg ha alacsonyabb pontra lépnénk, akkor a lépést *p(t)* valószínűséggel fogadjuk el, ahol *t* az időt jelöli. A valószínűséget legtöbbször a következő képlet alapján számolják ki:

$$p(t) = e^{\frac{\Delta X}{T(t)}} \tag{1.1}$$

ΔX jelöli a magasságkülönbséget ($\Delta X < 0$), T a hőmérsékletet. A hőmérséklet kezdetben magas, majd az idő múlásával csökken, hasonlóan, ahogy a fém hőmérséklete is kezdetben magas, majd fokozatosan lehűl a normális hőmérsékletre.

A módszerben új, hogy az előző technikákkal szemben képes egy lokális csúcsról lejönni (a negatív irányú lépések segítségével), így a keresés során több csúcsot is megvizsgálhat. A működés során a pozitív irányú lépések előnyt élveznek, így egy magasabb csúcsról már csak kis eséllyel mászik le az algoritmus. A hőmérséklet fokozatos csökkentésével érik el, hogy az algoritmus futása során kezdetben minél több csúcsot megvizsgál, és végül egy csúcs tetején állapodik meg. A módszer sikeresen működik több alkalmazásban.

1.4. Evolúciós Algoritmusok

Az evolúciós algoritmusok a számítástudomány, közelebbről a mesterséges intelligencia (MI) jellegzetes problémamegoldó eljárásai. Az MI feladatait legtöbbször az jellemzi, hogy nem vagyunk birtokában a problématér teljes és átfogó szabályrendszerének, így az egzakt és optimális megoldást nem tudjuk algoritmikusan előállítani, hanem kis hatósugarú, lokális szabályok segítségével elemi lépésekből kereséssel állítjuk elő a megoldást.

A bemutatott biológiai játékok megismerése után talán nem meglepő, hogy a kutatók az evolúció területéről vett megoldó módszereket is sikerrel alkalmazták MI problémákra, hiszen a játékokban is azt láthattuk, hogy egyszerű szabályok alapján működő nagyszámú egyed érdekes és értelmes szerveződést és viselkedést mutat.

Azokat a problémamegoldó rendszereket, melyek az evolúció valamely ismert mechanizmusára épülnek, evolúciós algoritmusoknak nevezzük. Az evolúciós algoritmusok gyűjtőfogalom, a legismertebb evolúciós algoritmusok a genetikus algoritmusok, evolúciós programozás (1.4.1. rész), evolúciós stratégia (1.4.2. rész), osztályozó rendszerek (1.4.4. rész), genetikus programozás (1.4.5. rész). Mivel a módszerek könnyen

összekeverhetők, ebben a részben egy rövid leírást adunk róluk, melyben rávilágítunk a köztük lévő különbségekre. Ha az olvasót jobban érdekli valamelyik módszer, a [HB98] cikkben egy-egy rövid leírást, és bőséges irodalomjegyzéket talál.

Az összes algoritmus közös vonása, hogy egy populációt kezelnek, melyben különféle egyedeket találhatunk. Amint láttuk, az összetettebb biológiai játékokban és a valódi evolúcióban a populáció egyedei a környezettel szemben sajátos viselkedést mutatnak, mintegy valamilyen megoldást adnak egy környezeti problémára. Ennek megfelelően az evolúciós algoritmusokban az egyedek a megoldandó probléma egy-egy lehetséges megoldását reprezentálják. A cél, hogy egy minél jobb megoldást találjunk a problémára. Az esetek többségében nincs esély a legjobb megoldást megtalálni, megelégszünk azzal, ha egy elég jó megoldást tudunk adni.

A populációban lévő egyedek az állatokhoz hasonlóan változnak. Az egyedek képesek szaporodni, ilyenkor új egyedek születnek. Egy egyednek lehet több szülője is, ekkor a szülők között keresztezés történik. A szaporodás során előfordulhat mutáció is. Végül az egyedek meg is halhatnak, ekkor egy új egyed foglalja el helyüket a populációban.

A populáció minden egyedéhez egy fitneszérték van hozzárendelve, ez annál magasabb minél jobb az egyed által reprezentált megoldás. A szaporodás során előnyben részesülnek a magas fitneszértékkel rendelkező egyedek, ezért az ilyen egyedeknek várhatóan több utóduk lesz, így a populáció várhatóan egyre magasabb fitneszértékekkel rendelkező egyedeket fog tartalmazni.

Az általános evolúciós algoritmus pszeudo-kódja a következő:

1.1. algoritmus Evolúciós Algoritmusok

$t \leftarrow 0$ {Kezdeti idő beállítása}
initpopuláció P_t {Kezdeti populáció létrehozása (többnyire véletlenszerűen)}
fitnesz_számít P_t {Fitnesz értékek kiszámítása}
while amíg nincs kész **do**
  P'_t := szülőkiválasztás P_t {Szülők választása}
  keresztez P'_t {A szülők génjeinek keresztezése}
  mutáció P'_t {Véletlen mutáció}
  fitnesz_számít P'_t {Az új fitnesz kiszámítása}
  P_{t+1} := túlélő P_t, P'_t {Az új populációba kerülnek az egyedek}
  t := t + 1
end while

Vizsgáljuk meg a kódot! Az algoritmus elején feltöltjük a populációt. Ez többnyire véletlen egyedekkel való feltöltést jelent, előfordulhat azonban, hogy már ismerünk pár egyedet, és ezekkel töltjük fel a populációt. A feltöltés után kiszámítjuk az egyedek fitneszértékét.

Egy szimulációs lépés során az egyedek közül kiválasztunk néhányat, ezek lesznek a következő generáció szülői. A kiválasztott szülők keresztezésével létrehozzuk a

leszármazottakat. A természetben jelen lévő mutációt szimulálva, a leszármazottakat egy részét kissé megváltoztatjuk. Most már megszülethetnek az utódok, ezért kiszámítjuk a leszármazottak fitneszértékét. Mivel nem szeretnénk, ha túl sok egyed lenne a populációban, ezért az előző generáció egyedei, és a leszármazottak közül kiválasztjuk azokat, melyek a következő generációban is jelen lesznek. A többi egyedet eltávolítjuk a populációból, ők meghalnak.

Kérdés még, meddig tartson a szimuláció. A szimuláció során az egyedek egyre inkább hasonlítanak egymásra, ezért egy idő után a populáció már alig változik. Amikor a populáció változása már nagyon kicsi, azt mondjuk, konvergált az algoritmus. Ha az algoritmus konvergált, akkor megállítjuk, és a populációban található legjobb egyedet (pontosabban az általa reprezentált megoldást) adjuk megoldásként.

Nem határoztuk meg eddig, hogyan működik a szülők kiválasztása, a keresztezés, a mutáció, a túlélők kiválasztása, és azt sem miként reprezentálunk egy megoldást a populációban. Azért nem határoztuk meg ezeket, mert a különféle evolúciós algoritmusokban különféleképpen működhetnek, ez alapján tudjuk megkülönböztetni az evolúciós algoritmusok most bemutatásra kerülő osztályait. Az egyes osztályok között szoros kapcsolat, sőt gyakran átfedés van, vagyis könnyen tervezhetünk olyan algoritmust, mely bizonyos jellemzőiben az egyik, míg más jellemzőiben egy másik osztályba tartozik.

1.4.1. Evolúciós Programozás

Az evolúciós programozás során is egy adott probléma lehetséges megoldásaiból álló populációt kezelünk. Az evolúciós programozás során nincs megkötés a megoldások ábrázolási módjára, az eredeti feladat által meghatározott formában tároljuk a megoldásokat. Tehát ha például optimális neuronhálókat keresünk, magát a neuronhálót tároljuk, nincs szükség arra, hogy elkódoljuk a neuronhálókat (például bitvektorrá).

Itt is egy véletlenül választott populációval indul az algoritmus, egy lépés során először az összes egyedről másolatot készítünk, majd a lemásolt egyedek mutáción esnek át. A mutáció különböző mértékű lehet, de a kisebb mutációknak nagyobb a valószínűsége, mint a nagy változást okozóknak.

A módosult egyedek fitneszértékének kiszámítása után már csak annak az eldöntése van hátra, hogy az eredeti populáció és a megváltozott populáció mely egyedei kerülnek az új populációba. Ez a kiválasztás többnyire egy bajnokság segítségével történik. A legegyszerűbb változata a bajnokságnak, ha véletlenül kiválasztunk 2 egyedet, és a nagyobb fitneszértékkel rendelkező kerül az új populációba. Ezt addig ismételjük, amíg fel nem töltődik az új populáció.

Az evolúciós programozás során az esetek döntő részében nem alkalmaznak keresztezést, így biológiai szemszögből nézve ez olyan, mint amikor több faj fejlődését vizsgáljuk, hiszen a fajok között sincs kereszteződés. Tehát az egyedek itt egy-egy fajt képviselnek. Pusztán biológiai szemszögből nézve itt nem is használhatnánk a populá-

ció kifejezést, hiszen az azonos fajba tartozó egyedek csoportját jelöli. Az egységesség kedvéért itt is a populáció szót használjuk, remélhetően ez nem zavarja meg az olvasót.

1.4.2. Evolúciós Stratégia

A főként technikai problémák megoldására használt evolúciós stratégia az evolúciós algoritmusok legbonyolultabb, és egyben legsikeresebb fajtája. A technikai problémák megoldásai során a megoldás paramétereinek optimális értékét keressük. A populációban lévő egyedek tehát a paramétereket tartalmazzák, az esetek döntő részében számokat.

Az algoritmusnak több változata létezik. A legegyszerűbb [(1+1)-es] változatnál generációnként 1 szülő generál 1 leszármazottat, és az evolúciós programozáshoz (1.4.1. rész) hasonlóan a kisebb mutációk valószínűsége nagyobb, a nagyobb mutációk valószínűsége kisebb. (Mivel az egyedek számokat tartalmaznak, a mutáció a számok megváltoztatását jelöli. Könnyen elérhető, hogy a kisebb mutációk valószínűsége nagyobb, a nagyobb mutációk valószínűsége kisebb legyen, például egy 0 várható értékű, normális eloszlású véletlenszámot kell a paraméterhez adni) Keresztezésről 1 szülőnél nyilván nincs értelme beszélni. A szülő helyét nem feltétlenül foglalja el az utód, csak abban az esetben, ha a fitneszértéke nagyobb. Felfedezték, hogy az algoritmus akkor hatékony (vagyis akkor oldja meg sikerrel a feladatokat), ha az összes mutációnak az 1/5 része sikeres. (Vagyis átlagosan minden ötödik mutációnál nagyobb az utód fitneszértéke a szülő fitneszértékénél.)

Amennyiben nem 1 szülőt választunk, hanem m-et, akkor már a keresztezés is szerepet játszik. A következő általánosítás, ha nem 1, hanem l utódot generálunk minden generációban.

Attól függően, hogy a túlélők kiválasztásánál figyelembe vesszük-e a szülőket, beszélhetünk *plusz* és *vessző* stratégiáról. A plusz stratégiánál [$(m + l)$] a túlélőket az m szülő és az l leszármazott közül választjuk ki, a vessző stratégiánál [(m, l)] csak a leszármazottak közül választunk. Míg a plusz stratégiánál előfordulhatnak halhatatlan egyedek (melyek mindig átkerülnek a következő generációba), a vessző stratégiánál minden megszületett egyed előbb-utóbb meghal.

Az m/l arány megválasztása nagyban befolyásolja a konvergenciát. Ha gyors konvergálást szeretnénk valamelyik lokális maximum felé, akkor jó lehet az (5,100) arány, ha a globális maximumot célozzuk meg, és a sebesség kevésbé fontos, akkor jó lehet a (15,100) arány.

1.4.3. Genetikus Algoritmusok

A genetikus algoritmus során a lehetséges megoldásokat nem az eredeti feladatnak megfelelő formában tároljuk a populációban (mint az evolúciós programozás (1.4.1. rész) során), hanem a tárolás előtt minden lehetséges megoldáshoz egy-egy bitvektort

(kromoszómát) rendelünk hozzá. A továbbiakban minden műveletet (például: keresztezés, mutáció) a kromoszómákon hajtunk végre.

A szülők kiválasztása a genetikus algoritmus során a következőképp történik: fele annyi szülőpárt képezünk, mint a populáció mérete. Így átlagban minden egyed egy párban szerepel, de a kiválasztás során előnyt élveznek a magasabb fitneszértékkel rendelkező egyedek. Minden szülőpár két utódot generál. Az utódok generálása során keresztezés és mutáció történhet.

A túlélők kiválasztása nagyon egyszerű: a régi populáció minden egyede meghal, helyüket az új populáció egyedei foglalják el.

1.4.4. Osztályozó Rendszerek

Az osztályozó rendszereket néha az evolúciós algoritmusok önálló ágának, néha a genetikus algoritmusok egy speciális alkalmazásának tekintik. Az osztályozó rendszereket legegyszerűbben az animat segítségével ismerhetjük meg. Az animat [Asz96] szó az animal + robot (állat + robot) szavak összevonásával keletkezett.

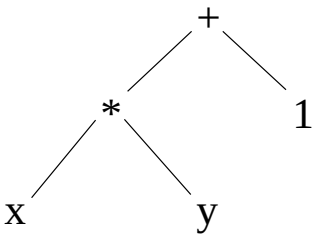
Az animat egy négy részből álló szerkezetet takar:

1. környezet
2. érzékelők
3. manipulátorok
4. irányító rendszer

Az animat egy *környezetben* él, ez a környezet általában csak a számítógépben létező virtuális világ, de ha megépítünk egy animatot, akkor lehet például egy szoba is. Az animat az *érzékelőivel* figyeli a környezetét. Az érzékelők egy megépített animatnál lehetnek fotocellák, nyomásérzékelők, mikrofonok, kamerák. Az animat a *manipulátoraival* reagál az érzékelői által észlelt dolgokra. A manipulátorok lehetnek kerekek, lábak, lánctalpak a mozgáshoz, kezek a környezet tárgyainak megfogásához, illetve száj a táplálkozáshoz. Az animat reakcióit az *irányító rendszer* irányítja, ez mondja meg, milyen ingerre, mi a helyes reakció.

Az irányító rendszer első megközelítésben egy fekete doboz. A fekete dobozt a továbbiakban egy egyszerű számítógépnek tekintjük. A számítógépen futó program nagyon egyszerű. Adott több if-then szabály a memóriában, a számítógép az inputnak megfelelő szabályt választ ki, és az adott szabály szerint viselkedik. Az if-then szabályokat viszonylag egyszerűen tudjuk azonos hosszúsága bitvektorokká konvertálni, az egyszerűbb kezelés érdekében.

A fentiek talán jobban érthetők, ha megnézzük az egyik legismertebb animatot, mely egy békát modellez. Az animat neve Kermit, részletesebb leírását a [HB98] cikkben találhatjuk. A békánk egy kis tóban él, szemei segítségével érzékeli a környezetet, lábai segítségével tud mozogni, és képes bekapni a legyeket. A béka mellett különféle



1.8. ábra. Egy egyszerű kifejezésfa

rovarokat (legyeket, méheket, darazsakat), és esetleg gólyákat találhatunk a tóban, illetve környékén. A béka célja minél több legyet enni, és elkerülni a gólyákat. A békát vezérlő if-then szabályok nagyon egyszerűek, például a következők lehetnek:

- Ha balra lát egy kicsi rovart forduljon balra
- Ha jobbra lát egy kicsi rovart forduljon jobbra
- Ha elöl lát egy kicsi rovart menjen előre
- Ha közvetlen maga előtt egy kicsi rovart lát, melynek nincsenek csíkjai, akkor egye meg a rovart (Kermit nem szereti a darazsakat, méheket)
- Ha valami nagy, kelepelő lényt lát, akkor meneküljön

Egy if-then szabályt nevezünk osztályozónak, ilyen szabályok halmazát osztályozó populációnak. Itt tehát az egyedeknek az osztályozók felelnek meg. Amennyiben a szabályok nem változhatnak, egyszerű osztályozó rendszerről beszélünk, ha megváltozhatnak, akkor tanuló osztályozó rendszerről.

A tanuló osztályozó rendszernél minden szabály kiválasztása függ még egy új paramétertől, mely változhat a tapasztalatoknak megfelelően. Ha egy szabály alkalmazása sikeres volt, az animat nagyobb valószínűséggel választja később újra a szabályt, ha sikertelen, akkor kisebb valószínűséggel. További lehetőség a fejlődésre egyes szabályok kitörlése, új szabályok generálása, szabályok mutációja, és több szabályból új szabály képzése.

1.4.5. Genetikus Programozás

Az eddig megismert evolúciós algoritmusok során a populáció egy egyede a probléma egy lehetséges megoldása volt. Ezzel szemben a genetikus programozás során a populáció nem lehetséges megoldásokat, hanem a problémát megoldó programokat tartalmaz. Vagyis minden egyed egy-egy olyan program, mely képes megoldani a kezdeti problémát. A cél egy olyan program, mely (közel) optimális eredményt ad.

A populációban található programokat többnyire nem hagyományos programozási nyelven írt programsoronként tároljuk, hanem kifejezésfában. Az $x \cdot y + 1$ értéket kiszámoló program kifejezésfáját láthatjuk az 1.8. ábrán. Mivel fákat nagyon könnyen kezelhetünk Lisp nyelven, a meglévő programok nagy része ezen a nyelven íródott. A

genetikus programozás során a keresztezés a legfontosabb operátor, mutációt az esetek legnagyobb részében nem használnak. A keresztezés a kifejezésfáknál a két kifejezésfa véletlen részfájának cseréjét jelenti.

Az amerikai MIT kutatóintézetben készítettek egy genetikus programozásra épülő alkalmazást, mely sikeresen működik. Érdemes még megemlíteni, hogy az alkalmazás során eredményül kapott programok, bár képesek megoldani az eredeti problémát, sőt elég jó eredményt adnak, áttekinthetetlenül bonyolultnak tűnnek egy ember számára.

2. fejezet

A genetikus algoritmusok ismertetése

Ebben a fejezetben a genetikus algoritmusok elméletét vizsgáljuk meg. Először az evolúció és a genetika alapfogalmaival ismerkedünk meg, majd a genetikus algoritmusok váza, alapfogalmai következnek. A továbbiakban a genetikus algoritmusok egyes részeinek alaposabb elemzése, illetve továbbfejlesztési lehetőségei következnek. A fejezetben megismerkedünk a genetikus algoritmusok matematikai elméletének alapjaival is.

2.1. Az evolúció

A továbbiakban megvizsgálandó algoritmusokhoz szükségünk van a darwini evolúció, azaz a törzsfejlődés ismeretére. Darwin előtt úgy gondolták, Isten megteremtette az összes állat- és növényfajt, és a fajok azóta lényegében változatlan formában élnek a Földön.

Ma már ismert, hogy a valóságban a fajok közt, és az egyes egyedek között szüntelen versengés folyik. Az életképesebb, a többi egyednél valamilyen területen jobbnak mutatkozó egyed nagyobb valószínűséggel éri meg az ivarérett kort, várhatóan több utóda lesz, mint egy gyengébb egyednek. A természetes kiválasztódás során az élőlények versengése rendkívül változatos lehet. Dönthet az élőlények között a táplálékszerzés ügyessége (melyik oroszlán éri utol az antilopokat), a táplálékká válás elkerülésének képessége (melyik antilop tud elfutni az oroszlánok elől), fajon belüli harc a nőstényekért (szarvasbikák küzdelme), a tűrőképesség mértéke (ki éli túl a szárazságot), az alkalmazkodás képessége (ki tud alkalmazkodni a felmelegedett időjáráshoz).

A kiválasztódást irányíthatja az ember is, ekkor beszélünk mesterséges kiválasztódásról. A fajok folytonos változása eredményezi azt, hogy egy faj több alpopulációra oszlása után, ha az elkülönülés hosszabb ideig tart új alfajok, még hosszabb elkülönülés után új fajok keletkeznek.

Az evolúcióban az egyik legérdekesebb vonás az, hogy minden irányítottság nélkül (kivéve az ember faj- vagy fajtanemesítéseit) látványos eredményeket ér el. Termé-

szetesen az evolúció is vezethet zsákutcába, amikor egy faj olyan irányban fejlődik, melynek végén kihal. Ez legtöbbször a túlzottan specializálódott fajokkal fordul elő, melyek képtelenek alkalmazkodni a környezet megváltozásához, például egy másik faj kipusztulásához.

2.2. Genetika

Már az ősidőkben észrevették az emberek, hogy az azonos fajhoz tartozó élőlények, noha nagyon hasonlítanak egymásra, mégsem egyformák. Szerettek volna rájönni, mi határozza meg az egyes tulajdonságokat. A tudományos kíváncsiság legfőbb oka az volt, hogy egyre tökéletesebb háziállatokat, termesztett növényeket szerettek volna.

Észrevették, bár az utódnak néha olyan tulajdonságai vannak, melyet egyik szülőben sem lehet észlelni, (Például két fekete kecskének tarka kiskecskéje születik) az utódok mégis nagymértékben hasonlítanak szüleikre. A felfedezés lehetővé tette a nemesítést, egyszerűen a pásztorok csak a legjobb állatok szaporodását engedték, a növénytermesztők pedig kigyomlálták a nemkívánt egyedeket. Nem tudták ugyanakkor megmagyarázni, mitől függ, hogy egy utód hasonlít-e valamelyik szülőjére.

Mint látjuk, alkalmazni már tudták a genetikát, de az öröklődés szabályait nem ismerték. A problémát nehezítette, hogy a középkorban még elfogadott volt az ősnemzés tana, azaz hogy bizonyos élőlények maguktól keletkeznek (romló húsból nyüvek, homokból bolha), ami nyilván ellentmond az öröklődésnek, hiszen nincs honnan örökölni a tulajdonságokat. Általánosan elfogadott volt az is, hogy az embereknél a gyermek tulajdonságait (nagyobbrészt) az apától örökli. Ez természetesen nem gátolta meg az embereket abban, hogy a leánygyermekek születéséért az anyákat okolják. További nehézséget jelentett, hogy több tulajdonságról nehéz eldönteni, hogy öröklődik-e. (Ma sem eldöntött, mitől lesz valakiből zseni, örökli a szüleitől a képességet, vagy a sok tanulás az oka, esetleg mindkettő)

Az öröklődés tulajdonságainak alapjait végül Mendel fedezte fel a XIX. század közepén. A továbbiakban a genetika alapvető fogalmait ismerjük meg, a genetikáról többet a [LG83], [Ber89] és [Moh96] könyvekből tudhatunk meg.

2.2.1. Gének

Az öröklött tulajdonságokat a *gének* határozzák meg. Egy génnek két fontos jellemzője van, a *funkciója*, és *lókusza* (helye). A gén funkciója azt mondja meg, melyik tulajdonságot határozza meg a gén.

A gének lehetséges értékei az *allélok*. Egy génnek az egyszerűbb esetekben két allélja van, de például az AB0 vércsoportrendszerben a vércsoportot leíró génnek három allélja van (0, I^A, I^B)

A gének *kromoszómákat* alkotnak, egy kromoszóma egymás után fűzött génekből áll. Egy gén lókusza a gén helye a kromoszómában. Ha egy élőlény kromoszómájában

kicserélnénk két gént, csak a gének lókuszai változnának meg, funkciójuk változatlan maradna, így az élőlény tulajdonságai elvileg nem változnának.

A kromoszómák összességét *kromoszómaszerelvénynek* is nevezik. Bizonyos élőlényekben egy kromoszómaszerelvény van, azaz egy gén egyszerűen meghatároz egy tulajdonságot. Ezeket *haploid* szervezeteknek nevezzük. Haploid szervezetek például az ivartalanul szaporodó egysejtűek, több gombafaj, és a hím méhek.

Az általunk ismert élőlények nagy részében (például az emberben is) két kromoszómaszerelvény található (ezeket nevezzük *diploid* szervezeteknek) , azaz minden tulajdonságot két gén szeretne meghatározni. Leggyakrabban a gén egyik allélja domináns a másikkal szemben, azaz ha a két allél különbözik, az élőlény olyan tulajdonságú lesz, amilyen tulajdonságot a domináns gén meghatároz.

A példából látható, hogy meg kell különböztetni az élőlény megjelenését (*fenotípusát*) és az élőlény által tartalmazott allélokat (*genotípusát*).

Léteznek olyan élőlények, melyek kettőnél több kromoszómaszerelvényt tartalmaznak, ezek a *poliploid* szervezetek. Ilyen növény például a burgonya.

2.2.2. Szaporodás

A szaporodás során az élőlények utódokat hoznak létre, és egyúttal saját génkészletüket mentik át a következő generációba.

A szaporodást két fő csoportra kell osztani, az ivartalan, és ivaros szaporodásra. Ivartalan szaporodásnál az utódnak egy szülője van, és az utód — a később említendő mutációt nem számítva — teljesen megegyezik szülőjével. Ivaros szaporodásnál az utódnak két szülője van, és genetikai anyaga a két szülő genetikai anyagának keveréke, tehát az utód *genotípusa* különbözik a szülők *genotípusától*. Érdemes megjegyezni, hogy az ivartalanul szaporodó élőlények jelentős részénél szintén van mód a genetikai anyagok keveredésére.

2.2.3. Mutáció

Mind az ivartalan, mind az ivaros szaporodásnál előfordul, hogy a genetikai anyag másolása közben hiba történik, vagyis *mutáció* jelentkezik. Megváltozhat egy gén értéke, kromoszómarészek maradhatnak ki, kettőződhetnek meg, esetleg meg is fordulhatnak. Előfordul a kromoszómák eltörése, a letört kromoszómadarabok elveszhetnek, vagy egy másik kromoszómához tapadhatnak hozzá. Előfordulhat a kromoszómaszám megváltozása is. (Kromoszómaszám megváltozás okozza a Down-kórt, Turner- és Klinefelter-szindrómát.) A mutációk egy része spontán mutáció, de a mutagén anyagok is kiválthatnak mutációt. Végezetül meg kell említeni, hogy a példáktól eltérően a mutáció nem egyértelműen negatív dolog, vannak pozitív mutációk, és a mutáció elősegíti a genetikai változatosságot is.

2.3. A genetikus algoritmusok alapmodellje

Mint tudjuk a genetikus algoritmusok egyfajta evolúciós algoritmusok. Meg kell határoznunk, hogy az evolúciós algoritmusok általános modelljében nyitottan hagyott pontok (reprezentáció, szülőkiválasztás, keresztezés, mutáció, túlélők kiválasztása) miként vannak a genetikus algoritmusokban meghatározva, miben különböznek a genetikus algoritmusok az evolúciós algoritmusok többi változatától.

A genetikus algoritmus során a populáció egy egyede a megoldandó probléma egy lehetséges megoldásának felel meg. (Tehát nem magának a megoldó programnak, mint a genetikus programozásban (1.4.5. rész).)

A lehetséges megoldásokat nem az eredeti feladatnak megfelelő formában tároljuk a populációban (mint az evolúciós programozás (1.4.1. rész) során), hanem a tárolás előtt minden lehetséges megoldáshoz egy-egy kromoszómát rendelünk hozzá. A továbbiakban minden műveletet (például: keresztezés, mutáció) a kromoszómákon hajtunk végre. Tehát e genetikus algoritmus működése során nem a keresési tér (S) pontjaival dolgozunk, hanem az úgynevezett kromoszómatér (C) pontjaival. A kromoszómatérrel és a kromoszómatérre történő leképezéssel ($S \rightarrow C$) a 2.6. részben foglalkozunk.

A szülők kiválasztása a genetikus algoritmus során a következőképp történik: fele annyi szülőpárt képezünk, mint a populáció mérete. Így átlagban minden egyed egy párban szerepel, de a kiválasztás során előnyt élveznek a magasabb fitneszértékkel rendelkező egyedek. Minden szülőpár két utódot generál. Az utódok generálása során keresztezés és mutáció történhet.

A túlélők kiválasztása a legtöbb genetikus algoritmus során nagyon egyszerű: a régi populáció minden egyede meghal, helyüket az új populáció egyedei foglalják el.

2.3.1. Az általános GA pszeudo-kódja

Az általános genetikus algoritmus pszeudo-kódja nem sokban különbözik az általános evolúciós algoritmusok pszeudo-kódjától (1.1. algoritmus), a különbség a szülők kiválasztásának módjában van.

A kezdeti populáció feltöltése az általános evolúciós algoritmusokhoz hasonló módon történik. Többnyire véletlenül választott egyedeket helyezünk a populációba, de ha már ismerünk sikeres egyedeket, velük is feltölthetjük a populációt.

A fitneszértékek kiszámítása problémafüggő, a genetikus algoritmusok során csak annyit követelünk meg, hogy a fitneszfüggvény értékei ne legyenek negatívok, és a jobb megoldásokhoz magasabb fitneszérték tartozzon. További feltétel, hogy a keresési térben található illegális megoldásoknál a fitneszfüggvény értéke 0 legyen. (Vagyis ha a fitneszfüggvény értelmezési tartománya szűkebb a keresési térnél, akkor a fitneszfüggvényt ki kell terjeszteni, hogy értelmezési tartománya megegyezzen a keresési térrel.) A fitneszfüggvényekről bővebben a 2.9. részben olvashatunk.

2.1. algoritmus GA

```
t ← 0 {Kezdeti idő beállítása}
initpopuláció P_t {Kezdeti populáció létrehozása (többnyire véletlenszerűen)}
fitnesz_számít P_t {Fitnesz értékek kiszámítása}
while amíg nincs kész do
  P_{t+1} = {} {Következő populáció itt még üres}
  for i:=0 to populáció_méret / 2 do
    E_1, E_2 := szülőpár P_t {Egy szülőpár választása}
    keresztez E_1,E_2 {A szülőpárok génjeinek keresztezése}
    mutáció E_1 {Véletlen mutáció}
    mutáció E_2 {Véletlen mutáció}
    fitnesz_számít E_1 {Az új fitnesz kiszámítása}
    fitnesz_számít E_2 {Az új fitnesz kiszámítása}
    P_{t+1} = P_{t+1} + E_1 + E_2 {Az új populációba kerülnek az egyedek}
  end for
  t := t + 1
end while
```

2.4. Egy példa a GA működésére

2.4.1. A feladat

A feladat az $x \cdot (sin(x) + 1)$ függvény maximalizálása, $x \in [0, 20 \cdot \pi]$ esetén.

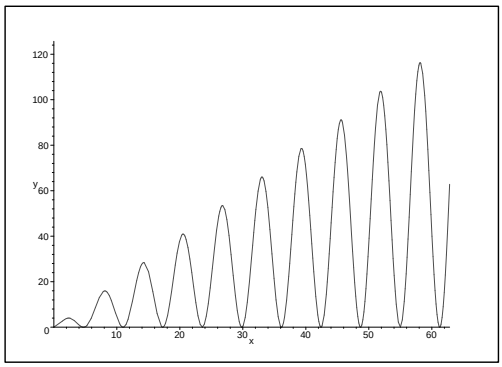
Először is vizsgáljuk meg a maximalizálandó függvényt. Az $x \cdot sin(x)$ függvénynek több csúcspontja van, ráadásul ezek a csúcspontok különböző magasságúak. Ideálisnak tűnik tehát a függvény, hogy teszteljük vele a genetikus algoritmusokat. Mivel a fitneszfüggvény értéke nem lehet negatív ezért módosítani kell az $x \cdot sin(x)$ függvényt. Ennek legegyszerűbb módja, ha $sin(x)$ értékét megnöveljük eggyel. Így kapjuk meg a $x \cdot (sin(x) + 1)$ függvényt.

2.4.2. A megoldás

Mivel csak x szerint kell optimalizálni, a kromoszómánk egyetlen gént fog tartalmazni. A gén lehetséges értékei megegyeznek x lehetséges értékeivel. A gén reprezentálásakor a legegyszerűbb módszert választjuk, bitvektorral reprezentáljuk (2.6.1. rész) a gént. A bitvektort tetszőlegesen hosszúra választhatjuk, minél hosszabb a bitvektor, annál pontosabb eredményt kapunk. (És annál lassabb lesz az algoritmus.) A teszt során a bitvektor hossza 20 bit lesz, így x két szomszédos értéke között a távolság kb. 0.00006.

A genetikus algoritmus további paramétereinek a következőket választottuk:

- Populáció mérete: 10



2.1. ábra. Az optimalizálandó függvény

- Mutáció: Bitváltás
- Mutáció valószínűsége: 0.01
- Keresztezés: Egypontos keresztezés
- Keresztezés valószínűsége: 0.7

Sorszám	Szülők	Kromoszóma	x	Fitnesz
1	-1,-1	00011011101110001010	6.80	10.19
2	-1,-1	01100001111000011001	24.02	2.52
3	-1,-1	01010001000001111100	19.89	37.02
4	-1,-1	11110000100101011011	59.05	94.40
5	-1,-1	10010110000010111011	36.83	8.62
6	-1,-1	00011111010101101000	7.69	15.28
7	-1,-1	10100101011110100111	40.61	49.72
8	-1,-1	10011101101100110011	38.71	71.41
9	-1,-1	10010010000110110111	35.86	1.28
10	-1,-1	10101111101001010011	43.11	10.09

2.1. táblázat. A kezdő populáció

A kezdő populációt teljesen véletlenül töltöttük fel. Az egyes egyedek fitneszértéke között nagy a különbség, a legnagyobb fitneszérték 94.40, az átlagos fitneszérték pedig 30.05. Érdemes megnézni mennyire hasonlítanak egymásra az egyedek. Legegyszerűbb, ha a kromoszómák közti Hamming-távolság (eltérő bitek száma) átlagát vesszük. Ez a kezdeti populációnál 10.22.

A 2.2. táblázat mutatja az algoritmus első lépését. A szülőkiválasztásnál legsikeresebbnek a 8., 4. és az 1. egyed bizonyult. A öt párosodás során négy esetben volt keresztezés. Mutáció egyetlen esetben történt. A legsikeresebb egyed fitnesze maga-

#	Szülő	Szülő kromoszóma	Keresztezés után	x	Fitnesz
11	10,4	10 101111101001010011	101100001001010**1**0011	43.34	17.38
12	10,4	11 110000100101011011	11101111101001010011	58.82	103.87
13	1,4	0001101110111 0001010	00011011101111011011	6.81	10.22
14	1,4	1111000010010 1011011	11110000100100001010	59.04	94.62
15	3,4	010100010 00001111100	01010001000101011011	19.90	37.18
16	3,4	111100001 00101011011	11110000100001111100	59.04	95.01
17	8,1	10011101101100110011	10011101101100110011	38.71	71.41
18	8,1	00011011101110001010	00011011101110001010	6.80	10.19
19	8,8	1001110110 1100110011	10011101101100110011	38.71	71.41
20	8,8	1001110110 1100110011	10011101101100110011	38.71	71.41

2.2. táblázat. Első lépés

Kromoszóma	Fitnesz
11101111111001010101	101.56
11101111101001010001	103.87
11101111111011010101	101.27
11101111101001010101	103.86
11101111111001010101	101.56
11101111111001010001	101.57
11101111101001010001	103.87
11101111101001010001	103.87
11101111111001010101	101.56
11101111111001001110	101.58

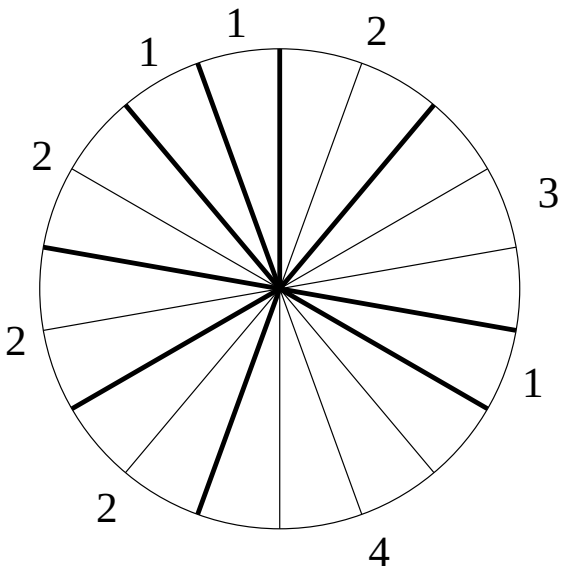
2.3. táblázat. A 20. generáció

sabb mint az előző esetben: 103.87. Az átlagos fitneszérték még nagyobb mértékben nőtt: 58.27. Az átlagos Hamming távolság is csökkent: 8.02.

Érdekességként érdemes megnézni a 20. generáció kromoszómáit. Látszik, hogy a kromoszómák már majdnem teljesen megegyeznek. Erről a pontról az algoritmus már csak mutáció segítségével mozdulhat el.

2.5. Szülőpárok és túlélők kiválasztása

A szülőpárok kiválasztásáról — melyet szelekció néven a genetikai operátorok közé is sorolnak — eddig csak annyit tudunk, hogy a nagyobb fitneszértékűeket arányosan többször kell kiválasztani. Ezt legegyszerűbben a *rulettkerék módszerrel* tehetjük meg.



2.2. ábra. A rulettkerék módszer

2.5.1. Rulettkerék módszer

Egy képzeletbeli rulettkereket készítünk, melyen minden egyedhez fitneszértékével arányos számú rekesz tarozik. A képzeletbeli rulettkereket megpörgetve megfigyeljük, hogy a képzeletbeli golyó melyik rekeszbe esik. Amelyik egyed rekeszébe esik, azt az egyedet választjuk szülőnek. Mivel a rekeszekbe egyforma eséllyel esik a golyó, és a rekeszek száma a fitneszértékkel arányos, a kiválasztás esélye is a fitneszértékkel lesz arányos. Egy ilyen rulettkereket láthatunk a 2.2. ábrán. Az ábra azt az esetet mutatja be, amikor a populáció 9 egyedből áll, melyek fitneszértékei rendre: 2, 3, 1, 4, 2, 2, 2, 1, 1. A rulettkerék tehát 18 (2+3+1+4+2+2+2+1+1) rekeszből áll, az azonos egyedhez tartozó rekeszeket vékony, a különböző egyedekhez tartozó rekeszeket vastag vonal választja el az ábrán.

2.5.2. Generációs szakadék

Generációs szakadéknak nevezzük azoknak az egyedeknek az arányát, melyeket kicserélünk a generációváltás során. A kanonikus GA során ez az arány 1. A természetben ez az arány csak a rövid életű állatoknál 1 (a szülő meghal mielőtt kikelne az utód), a hosszabb életű állatoknál a szülő is él még az utód születésénél. Ez egyrészt lehetőséget ad a leszármazottak nevelésére (ezt nem használjuk ki a GA során), másrészt az egymást követő nemzedékek közötti versengést is lehetővé teszi.

Ha a generációs szakadékot a lehető legkisebbre csökkentjük (két egyedet cserélünk mindig), akkor igen kiegyensúlyozott generációváltást kapunk. Ebben az esetben nemcsak a szülők kiválasztásával kell törődnünk, hanem el kell döntenünk, hogy me-

lyik két egyed helyére tesszük a leszármazottakat.

Több lehetőség közül néhány:

- Szülőkiválasztás fitnesz alapján, helyettesítés véletlenszerűen.
- Szülőkiválasztás véletlenszerűen, helyettesítés az inverz fitnesz alapján.
- Szülőkiválasztás fitnesz alapján, helyettesítés az inverz fitnesz alapján.

Fontos különbség a hagyományos GA-val szemben, hogy itt minden egyes párosodás után el kell végezni bizonyos statisztikai számításokat (pl. átlagos fitneszérték), és hogy a leszármazottak azonnal rendelkezésre állnak a párosodásra. Ez lehetőséget ad arra, hogy a sikeres gének hamarabb elterjedjenek a populációban. Egyes kutatók szerint ez komoly előny, mások szerint ugyanezt a hatást el lehet érni a kanonikus GA-nál is például a fitnesz módosításával.

2.6. A kromoszómareprezentáció módjai

A genetikus algoritmusok során a feladattól függően különböző kromoszómateret használhatunk. A kromoszómatér általában felírható $C = \Sigma^n$ alakban, ahol n a kromoszóma hossza és Σ egy véges ábécé. Σ leggyakrabban két elemet tartalmaz (0,1), ebben az esetben minden kromoszóma egy-egy bitvektor. Mivel ez a leggyakoribb és legjobban elemzett eset, a továbbiakban — kivéve néhány speciális esetet — a kromoszómákat bitvektornak tekintjük. Ha Σ kettőnél több elemet tartalmaz, akkor a kromoszómákat karaktervektornak (sztringnek) tekintjük, és Σ elemeit kis- és nagybetűkkel jelöljük.

2.6.1. $S \rightarrow C$ leképezések

Érdemes megvizsgálni, milyen módon tudjuk a keresési teret a kromoszómatérre leképezni. A továbbiakban megismerjük a keresési tér két leggyakrabban előforduló változatát és a változatokhoz tartozó lehetséges leképezési módokat. Látunk példát egy ritkábban használt keresési térre is, melyet neuronhálók optimalizálásánál alkalmazhatunk. A továbbiakban az egyszerűség kedvéért feltételezzük, hogy a keresési tér 1 dimenziós. Az 1 dimenziós esetek megvizsgálása után szót ejtünk a többdimenziós esetről is.

$S = [a, b] \subset \mathbf{R}$

A leggyakrabban előforduló eset, hogy egy adott intervallumba eső számot keresünk. Mivel Σ elemszáma véges, nem tudunk az intervallum minden pontjához egy kromoszómatérbeli pontot rendelni. Először tehát diszkretizálni kell az intervallumot.

2.4. táblázat. Diszkretizálás (a=1.0 b=4.5 k=3)

Diszkrét pontok:	1.0	1.5	2.0	2.5	3.0	3.5	4.0	4.5
Bitvektor:	000	001	010	011	100	101	110	111

Vegyünk 2^k pontot, a pontok legyenek egyenletesen elosztva az intervallumon belül. Az i. pont ($i \in [0, 2^k - 1]$) helye tehát $\frac{(2^k-1-i)\cdot a+i\cdot b}{2^k-1}$.

Az így kapott 2^k pontot már könnyen tudjuk kezelni. Legyen a kromoszómatér $C = \{0, 1\}^k$, vagyis a kromoszóma egy k bitből álló bitvektor. Ha a bitvektort mint bináris számot tekintjük, megkapjuk az ábrázolt pont sorszámát. Nézzünk egy egyszerű példát! Ha egy 1.0 és 4.5 közötti számot keresünk ($S = [1.0, 4.5]$), és 3 biten szeretnénk a számot ábrázolni ($C = \{0, 1\}^3$), akkor az intervallum 8 pontjához tudunk bitvektort hozzárendelni. A pontokat és a hozzájuk tartozó bitvektorokat a 2.4. táblázat tartalmazza.

Több kutató azt javasolja, hogy ne kódoljuk el a valós számokat bitvektorrá, vagyis a kromoszóma valós (lebegőpontos) számokat tartalmazzon. A módszer előnye, hogy könnyen alkalmazhatunk probléma-specifikus keresztezést és mutációt. Lehetséges mutációs módszer például, ha a valós számhoz egy 0 várható értékű, normális eloszlású véletlenszámot adunk hozzá.

$|S| <$**végtelen**

Ha a keresési tér elemszáma véges, akkor kromoszómatérnek választhatjuk a keresési teret ($S = C = \Sigma$). Ha például a keresett paraméter lehetséges értékei A, B, C, D, E, és F, akkor a kromoszóma egyetlen egy karakterből áll, a karakter lehetséges értékei megegyeznek a paraméter lehetséges értékeivel.

Ebben az esetben is lehetőség van arra is, hogy a kromoszómák bitvektorok legyenek. Ha a paraméterek lehetséges értékeinek száma kettő hatványa, akkor könnyen hozzárendelhetjük a bitvektorokat a lehetséges értékekhez. Ha a szám nem kettő hatvány, akkor nem ilyen egyszerű hozzárendelés. Az előbb említett példában 6 lehetséges érték van. Nyilvánvaló, hogy a kromoszóma hossza legalább 3 bit kell hogy legyen. A 3 bitből álló bitvektorok száma azonban 8, vagyis 2 bitvektorhoz nem tudunk értéket rendelni. Felmerül a kérdés mit tegyünk azokkal a kromoszómákkal, melyek ezt a 2 bitvektort tartalmazzák (ilyen kromoszómák létrejöhetnek a keresztezések és mutációk hatására)? *DeJong* szerint a következő lehetőségek közül választhatunk:

1. Tekintsük a kromoszómákat illegálisnak.
2. Rendeljünk a kromoszómákhoz alacsony fitneszértéket.
3. Rendeljünk az illegális kromoszómákhoz egy-egy legális kromoszómát

2.5. táblázat. A keresési tér pontjainak hozzárendelése a bitvektorokhoz

Keresési tér pontjai:	A	B	C	D	E	F	A	B
Bitvektor:	000	001	010	011	100	101	110	111

Az első két módszer nagyon hasonlít egymásra, mindkettő hibája, hogy olyan kromoszómákat is illegálisnak vagy nagyon gyengének tekint, melyekben a kromoszóma más területein sikeres gének találhatóak.

A 3. módszernek több változata létezik, a módszerek rövid összefoglalását például a [BBM93b] cikkben találhatjuk meg. A továbbiakban az egyik legegyszerűbb változatot ismerhetjük meg. A 8 lehetséges bitvektor közül az első 6-hoz sorban hozzárendeljük a keresési tér egy pontját (000 $\rightarrow$ A, 001 $\rightarrow$ B . . .). A fennmaradó 2 bitvektorhoz a keresési tér 2 tetszőleges pontját rendeljük hozzá (a példában (2.5. táblázat) 110 $\rightarrow$ A, 111 $\rightarrow$ B). Mivel mind a 8 bitvektorhoz hozzárendeltük a keresési tér egy pontját, nem kell attól tartani, hogy kereszteződés vagy mutáció hatására olyan bitvektor keletkezik, melyet nem tudunk kezelni. A módszer hibája, hogy a keresési tér egyes pontjaihoz egy bitvektort rendeltünk (C, D, E, F), míg más pontokhoz kettőt (A, B), ezért egyes pontok előnyt élveznek a többiekkel szemben.

Ha a felhasznált bitek számát megnöveljük, akkor az előny mértéke csökken. Ha például 4 bitet használunk az előző példában, akkor 16 lehetséges bitvektort kapunk. A keresési tér 2 pontjához kettő-kettő, 4 pontjához három-három bitvektort rendelhetünk ($2*2+4*3=16$). Ebben az esetben az előnyt élvező pontokhoz $\frac{3}{2}$-szer több bitvektor tartozik mint az előnyt nem élvező pontokhoz. Három bit felhasználásánál ez az arány 2 volt.

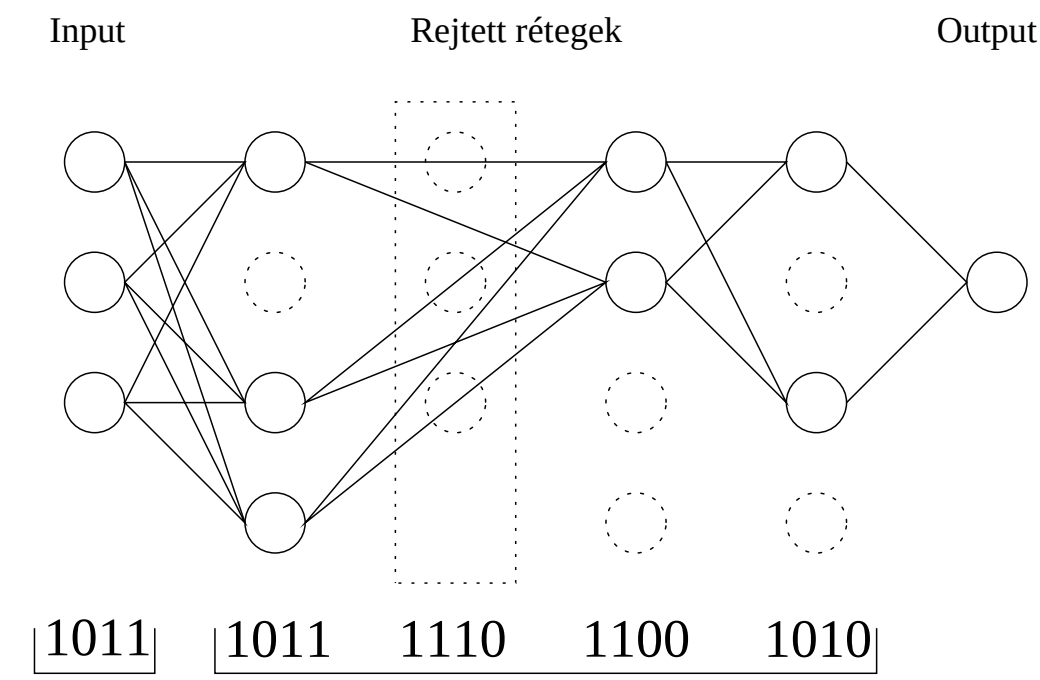
Neuronhálók reprezentálása

Ha mesterséges neuronhálókat szeretnénk optimalizálni, akkor a keresési tér a neuronhálók halmaza. A gyakorlatban a neuronhálóknak csak egy szűkebb csoportját szokták vizsgálni.

Most egy olyan reprezentálási módot ismerhetünk meg [Jel96], mellyel azokat a neuronhálókat tudjuk reprezentálni, melyek megfelelnek a következő feltételeknek:

- A neuronhálóban nincs irányított kör.
- A neuronhálónak 3 input neuronja van.
- A neuronhálónak 1 output neuronja van.
- A rejtett rétegek száma legfeljebb 4.
- Egy rétegen belül legfeljebb 4 neuron található.

A 2.3. ábrán láthatunk egy neuronhálót, és a neuronhálót reprezentáló bitvektort. A bitvektor réteg kontrol biteket és neuron kontrol biteket tartalmaz. Az input és output



2.3. ábra. Neuronháló reprezentálása

rétegek között találhatóak a rejtett rétegek, legfeljebb 4 darab. A réteg kontrol mondja meg, hogy a 4 lehetséges rejtett rétegből melyik szerepel a neuronhálóban. Az ábrán az első, harmadik és negyedik réteg szerepel, ezért a réteg kontrol tartalma 1011. Minden réteghez neuron kontrol bitek tartoznak. A neuron kontrol bitek adják meg, hogy a rétegen belüli neuronok közül melyik szerepel a neuronhálóban. Az ábrán a harmadik rétegben az első és második neuron szerepel, a harmadik és negyedik nem, ezért a neuron kontrol blokk tartalma 1100.

A bitvektor hosszát könnyen ki tudjuk számítani: 4 + 4 * 4 = 20 bit. Tehát $C = \{0,1\}^{20}$.

Többdimenziós keresési tér

Ha a keresési tér többdimenziós akkor egyszerre több paramétert (gént) kell reprezentálnunk. Az egyes paraméterekhez külön-külön rendelhetünk hozzá bit- vagy stringvektort, majd ezeket összefűzve kapjuk meg a kromoszómát. Mivel nehéz egy olyan kromoszómát kezelni melynek egyes részei bit-, más részei stringvektorok, ezért vagy az összes paramétert bitvektorra, vagy az összeset stringvektorra képezzük le.

Ha megvannak az egyes paraméterekhez rendelt kromoszómadarabok, már csak össze kell őket fűzni. Kérdés, milyen sorrendben fűzzük őket össze? Könnyen látható,

hogy ha $n > 1$ darab génünk van, akkor a lehetséges sorrendek száma $\frac{n!}{2}$. (A kettővel való osztás azért szerepel, mert az egész kromoszómát megfordíthatjuk.) A génsorrend akkor jó, ha az összefüggő gének közel vannak egymáshoz a kromoszómában, a függetlenek távol [BBM93a]. A feltételeknek nehéz eleget tenni, hiszen az egyes gének közti kapcsolat erősségét sokszor nem ismerjük eléggé. Ilyen esetekben hasznos lehet, ha olyan reprezentációt választunk, melyben a gén funkciója független a lókuszától (2.2.1. rész). Ilyen reprezentációról a 3.1. részben olvashatunk.

2.7. Mutáció

A mutáció a genetikai anyag véletlen megváltozása. A természetben megismert mutációt (2.2.3. rész) érdemes alkalmazni a genetikus algoritmusokban is.

Már a természetben jelentkező mutáció során is említettük, hogy a mutáció lehet hasznos és káros is. A genetikus algoritmusok során nem kell túlzottan tartani a káros mutációktól, hiszen az így keletkező rossz egyedek nagyon hamar kihalnak, így nem veszik el a helyet a sikeresebb egyedektől. Túl sok mutáció azonban már káros lehet, mivel túl sok életképtelen egyed születne. A mutáció valószínűségét ezért elég alacsonyan szokták meghatározni, egy bit mutációjának valószínűsége többnyire 0.001–0.01.

A legegyszerűbb mutációnál a mutáció hatásának kitett bit értékét megváltoztatják (0-ból 1, 1-ből 0 lesz). Néha a bit új értékét véletlenszerűen választják, így az esetek felében nem változik a bit értéke. Ha a kromoszóma nem bit-, hanem stringvektor, akkor az új értéket véletlenül választják, de nem biztos, hogy az egyes választásoknak egyforma a valószínűsége.

Ha a gének értékét binárisan kódoljuk, felmerülhet még egy probléma: Képzeljük el, hogy 5 biten ábrázoljuk egy gén értékét. A gén értéke 0 és 31 között lehet. Tegyük fel, hogy az optimum a 16-os érték, és a kromoszómánkban a gén értéke 15, ami nagyon közel van az optimumhoz. Ha binárisan ábrázoljuk a két értéket (15: 01111, 16: 10000), észrevehetjük, hogy a bitek értéke páronként különböző, vagyis 5 egymás utáni mutációra lenne szükség, hogy a 15-ös értékből 16-ot kapjunk, aminek nagyon kicsi az esélye.

További probléma a mutációval, hogy mivel minden bitnél ugyanakkora a mutáció valószínűsége, ugyanakkora az esély arra, hogy a gén értékét 1-el változtatjuk (ha a 0. bitek történik a mutáció), mint arra, hogy 16-tal (ha a 4. biten történik a mutáció). Ez ellentmond annak a megfigyelésnek, hogy a kisebb mutációk valószínűbbek, a nagyobbak ritkábbak. Két megoldási javaslat van a problémákra:

2.7.1. +/- ε módszer

A *Messa* által javasolt +/- ε módszerben a gén értékéhez hozzáadunk, vagy kivonunk belőle egy ε számot. Az ε szám kettő hatványa 1 és 2^M között, tehát egy olyan szám,

2.6. táblázat. Gray kódok

	0	1	2	3	4	5	6	7
Bináris kódolás	000	001	010	011	100	101	110	111
Gray kódok	000	001	011	010	110	111	101	100

mely binárisan ábrázolva pontosan egy db 1-es bitet tartalmaz (az 1-es bit pozícióját jelöljük i-vel).

Ha a módosítandó gén i. bitje 0, akkor ε hozzáadása után a bit 1 lesz, hasonlóan a hagyományos mutációhoz. Ugyanazt az eredményt kapjuk akkor is, ha a bit értéke 1 volt, és kivontuk belőle ε-t. A különbség akkor jelentkezik, ha a bit értéke 1, és hozzáadjuk ε-t, vagy ha a bit értéke 0, és kivonjuk. Az előbbi példában ha a 15-höz (01111) epszilon=1-et (00001) hozzáadunk, egy lépésben megkapjuk a 16-ot (10000).

Létezik a módszernek egy *csökkenő* változata is, melyben kezdetben véletlenszerűen választjuk 1 és 2^M között ε értékét, később, az algoritmus előrehaladtával azonban fokozatosan kizárjuk a nagyobb értékeket, így megnövelve a kisebb mutáció valószínűségét.

2.7.2. Gray kód

A Gray[1] kód használata esetén nem kell változtatnunk a mutáció algoritmusán, csak a gének értékének ábrázolási módján.

A Gray kódolás esetén is a bináris kódolásnál kapott bitvektorokat használjuk, de más sorrendben rendeljük hozzá a számokhoz. A Gray kódolásnál két szomszédos szám mindig olyan bitvektorokat rendelünk, melyek között csak egy bitben van eltérés. (Tehát két szomszédos bitvektor Hamming távolsága mindig 1.) Ennek köszönhető, hogy elég egyetlenegy mutáció, hogy a gén értékét 1-el megváltoztassuk. A 0–7 értékek bináris és Gray kódolását láthatjuk a 2.6. táblázatban.

Bár a mutációk többsége a Gray kódoknál kis mutáció lesz, nagyon nagy mutációk is előfordulhatnak, olyan nagyok is, melyek a hagyományos bináris kódolásnál nem jelentkezhettek. Ha a gén értéke 0 (000), akkor a 2. bit megváltoztatásával 100-t kapunk, ami a Gray kódolásnál a 7-et kódolja. Vagyis a mutációval a lehető legnagyobb változást idéztük elő, a legkisebb elemből a legnagyobbat kaptuk.

[1] A módszer neve *Frank Gray* nevéből származik, nem az angol szürke (gray, grey) szóból, így Graynak kell írni

2.7.3. Fenotípusos mutációk

Az eddig áttekintett mutációk közös vonása, hogy nem törődnek azzal, hogy az egyes bitek, számok mit reprezentálnak, pusztán az egyed genotípusát veszik figyelembe. Ennek a szemléletnek tagadhatatlan előnye, hogy ezek a módszerek univerzálisak. Bármilyen problémát próbálunk megoldani GA-val, ezeket a módszereket használhatjuk.

Ez az univerzalitás kísértetiesen emlékeztet az MI őskorában használt GPS-ekre (General Problem Solver). A kutatók olyan általános módszert kerestek, mely segítségével a megoldandó problémák többségét meg lehet oldani, vagyis a módszer nem használ fel semmit a probléma-specifikus tudásról. Kiderült, hogy ilyen módszereket elméletben könnyű gyártani, a gyakorlatban azonban a kombinatorikus robbanás miatt csak a legegyszerűbb — más módszerekkel is könnyen megoldható — problémákat oldja meg. A GPS-ek sikertelensége hosszú távra visszavetette az MI kutatásokat.

Hasonló jelenség tapasztalható a GA-nál is. A kanonikus GA, mely nem használ fel probléma-specifikus információt, elméletileg minden olyan problémát képes megoldani, melyet le lehet írni a GA számára. A gyakorlatban azonban az ilyen GA nem hatékonyabb mind a hagyományos módszerek, és komolyabb feladat megoldására nem alkalmas. A hatékonyság növelésére a probléma-specifikus tudást be kell ágyazni a GA-ba. Speciális reprezentálást, speciális genetikai operátorokat kell használni. Ha például bináris fákat ábrázolunk, akkor elképzelhető, hogy érdemes egy olyan mutációt használni, mely két részfát megcserél. A kromoszóma szerkezetétől függően egy ilyen mutáció során esetleg sok bit értéke megváltozik. Egy ilyen mutációra a hagyományos mutációs módszereket használja nagyon kis esély van.

A későbbiekben amikor konkrét GA alkalmazásokat vizsgálunk meg, több ilyen mutációra látunk majd példát.

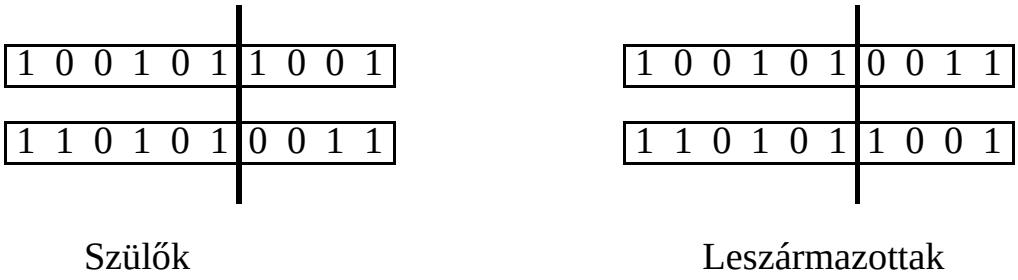
2.8. Keresztezés

A genetikus algoritmusok fontos eleme a keresztezés, ennek segítségével tudják egyes egyedek genetikai anyaguk egy részét kicserélni. A keresztezés azért fontos, mert a keresztezés során a szülőkben külön-külön jelenlévő tulajdonságok keveredhetnek az utódban. Az utódok generálásakor nem minden esetben történik keresztezés, lehetőséget adva arra, hogy a szülők génkészlete változatlan formában kerüljön a következő generációba. A keresztezés valószínűségét többnyire 60–70 %-ban határozzák meg.

A továbbiakban a különféle keresztezési módszereket nézzük át.

2.8.1. 1-pontos keresztezés

A legegyszerűbb keresztezési módszernél egy véletlen helyen (kereszteződési pontban) elvágjuk a kromoszómákat, és a kereszteződési pont utáni kromoszómadarabokat felcseréljük. A 2.4. ábrán látható egy egyszerű 1-pontos keresztezés. Kereszteződési



2.4. ábra. 1-pontos keresztezés

pontnak a 6. és 7. gén közötti helyet választottuk, ezt jelzi a függőleges vonal. Megfigyelhető, hogy a leszármazottak kromoszómája mindkét szülő kromoszómájából tartalmaz egy-egy darabot, és az is, hogy a létrejövő új kromoszómák mindkét szülő kromoszómájától különböznek.

2.8.2. Többpontos keresztezés

Az 1-pontos keresztezést könnyen módosíthatjuk úgy, hogy nem egy, hanem több kereszteződési pontot választunk, és az így feldarabolt kromoszómák megfelelő részeit cseréljük ki.

2-pontos keresztezés

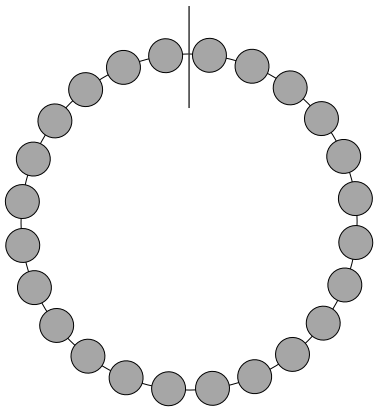
A többpontos keresztezés leggyakrabban használt változata a 2-pontos keresztezés.

Kétpontos keresztezésnél nem egy, hanem két keresztezési pontot választunk, és a két pont közötti részt cseréljük ki.

A kétpontos keresztezésnél a kromoszómát nemcsak lineáris sztringnek, hanem egy génekből képzett nyakláncnak is tekinthetjük. (Tehát a kromoszóma eleje és vége összekapcsolódik.) A 2.5. ábrán láthatunk egy ilyen kromoszómát, a függőleges vonal jelöli azt a pontot, ahol a kromoszóma eleje és vége összekapcsolódik. Ebből a szemszögből vizsgálva az 1-pontos keresztezést, az 1-pontos keresztezés felfogható egy olyan 2-pontos keresztezésnek, ahol az egyik kereszteződési pont mindig a kromoszóma elején található.

Minél több kereszteződési pontot választunk, annál jobban összekeveredik a szülők genetikai anyaga, ami egyrészt hasznos, hiszen így nő a genetikai változatosság, másrészt káros, hiszen ha az egyik szülőnél több jó gén található egymás mellett, nagy a valószínűsége, hogy egy kereszteződési pont elválasztja egymástól a géneket, és a leszármazottban már nem lesz megtalálható a jó génsorozat.

A többpontos keresztezések mellett szól, hogy amikor a populáció már nagyrészt konvergált, kevés kereszteződési pont esetén a leszármazottak az esetek nagy részében



2.5. ábra. A kromoszóma mint egy nyaklánc

teljesen megegyeznek a szülőkkel. Amennyiben úgy módosítjuk a keresztezés algoritmusát, hogy a leszármazottak szülőkkel való megegyezése esetén új kereszteződési pontokat választunk, akkor nagyrészt kiküszöböljük az előző problémát.

2.8.3. Uniform keresztezés

Ahogy a többpontos keresztezés során egyre több kereszteződési pontot választunk, egyre kisebb darabokra vágjuk szét a kromoszómát. Vágjuk most szét a kromoszómát a lehető legkisebb darabokra, azaz minden darab tartalmazzon egy-egy bitet.

A uniform keresztezés során először egy véletlen keresztezési maszkot generálunk. A keresztezési maszk 0-kból és 1-esekből áll, hossza a kromoszómában található bitek számával egyezik meg. A leszármazottak génkészletét a következőképpen kapjuk meg: Ahol a keresztezési maszkban 1-es áll, ott az első szülő génje kerül az első leszármazottba, a második szülő génje kerül a második leszármazottba. Ahol viszont a keresztezési maszkban 0-s áll, ott az első szülő génje kerül a második leszármazottba, és a második szülő génje kerül az első leszármazottba (2.6. ábra).

A uniform keresztezés felfogható egy olyan többpontos keresztezésnek, ahol a kereszteződési pontok száma nincs előre meghatározva, de a kereszteződési pontok várható értéke $L/2$ (ahol L a kromoszóma hosszát jelöli), így a uniform keresztezésre még inkább igazak a többpontos keresztezésnél említett előnyök és hátrányok.

2.8.4. A keresztezési módszerek összehasonlítása

Jelenleg is vitatott, melyik keresztezési módszer a leghatékonyabb. A különböző elemzések különböző eredményeket hoztak, ami azt jelenti, nem lehet egyértelműen kijelenteni melyik módszer a hatékonyabb, a hatékonyság függ a feladattól, és a genetikus algoritmus további paramétereitől.

Keresztezési maszk:

1	0	0	0	1	0	1	1	0	0	0	1	0	1	0	0	1	1

1. Szülő

0	1	1	1	0	1	0	1	0	1	1	1	0	0	1	1	0	0

2. Szülő

0	0	0	1	1	0	1	1	1	0	0	0	1	0	1	0	1	1

1. Utód

0	0	0	1	0	0	0	1	1	0	0	1	1	0	1	0	0	0

2. Utód

0	1	1	1	1	1	1	1	0	1	1	0	0	0	1	1	1	1

2.6. ábra. Uniform keresztezés

Széles körben elfogadott, hogy a 2-pontos keresztezési módszer előrelépés az 1-pontoshoz képest, a további kereszteződési pontok felvételével kapcsolatban nincs egyetértés.

Általánosan a következő mondható: 2-pontos keresztezést érdemes használni viszonylag nagy populációnál, uniform keresztezést kis populációnál. Rövid kromoszómánál kevés, hosszabb kromoszómánál több kereszteződési pontot érdemes választani. Ha a kromoszómában jó a gének sorrendje (2.6.1. rész), akkor a 2-pontos keresztezés ajánlott, a uniform keresztezést nem befolyásolja mennyire jó a génsorrend.

2.8.5. Egyéb keresztezési módszerek

Az előbb ismertetett módszereken kívül több viszonylag új, még nem eléggé elterjedt módszer létezik. További keresztezési módszereket ismerhetünk meg a 3.1.2. részben, az ott ismertetésre kerülő módszerek — ellentétben az itt leírtakkal — helyesen működnek abban az esetben is, ha megkülönböztetjük a gén funkcióját és lókuszát. A konkrét alkalmazásokról szóló részben fenotípusos kereszteződésekkel is találkozhatunk.

2.9. Fitneszfüggvény

A genetikus algoritmusoknál a fitneszfüggvény értéke alapján döntjük el, mely egyedeket választjuk ki, kinek a leszármazottai kerülnek a következő populációba. A fitneszfüggvény minden megoldandó problémánál más és más, ezért nagyon fontos a megfelelő fitneszfüggvény kiválasztása. Bár a fitneszfüggvény mindig a megoldandó problémától függ, érdemes megismerni a különböző típusú fitneszfüggvényeket.

Mint látni fogjuk, megfelelő fitneszfüggvény választása esetén is előfordulhat, hogy olyan problémák jelentkeznek, melyeket a fitneszfüggvény futás közbeni módosításával tudunk orvosolni. Meg fogjuk ismerni ezeket a problémákat, és azokat a módszereket, melyek segítségével a problémák megoldhatóak.

2.9.1. A fi tneszfüggvények fajtái

A feladatok egy részénél nagyon egyszerű a fitneszfüggvény kiválasztása, hiszen ha egy függvényt kell maximalizálnunk, akkor a maximalizálandó függvényt választhatjuk fitneszfüggvénynek. A továbbiakban olyan eseteket vizsgálunk, amikor nem ilyen egyszerű a fitneszfüggvény meghatározása.

Közelítő fitneszfüggvény

A feladatok egy részében a fitnesz függvény túl bonyolult, lassan számolható. Hiába tudjuk pontosan leírni, a nagy számításigény miatt az algoritmus lassan fog futni, ezért nem biztos, hogy a rendelkezésre álló időn belül megfelelő eredményt kapunk. Ebben az esetben segíthet az, ha az eredeti fitneszfüggvény helyett egy másik függvény értékét számoljuk ki, egy olyan függvényét, mely elég jól közelíti az eredeti függvényt, ugyanakkor sokkal gyorsabban kiszámolható. Ha jó függvényt választunk, akkor többet nyerünk azzal, hogy az algoritmus ugyanazon idő alatt több generációt tud kiértékelni, mint amit elvesztünk az által, hogy nem az eredeti függvényt használjuk.

Goldberg könyvében láthatunk példát [Gol89, 138. oldal] egy olyan alkalmazásra, ahol orvosi képeket vizsgáltak, és az összes képpont vizsgálata helyett (az lett volna az eredeti fitneszfüggvény), csak a pontok egy részét vizsgálták, és így lényegesen jobb eredményt kaptak.

Részcélokat kezelő fitneszfüggvény

Képzeljük el, hogy órarendet kell készítenünk. A megoldásoknak több feltételt kell kielégíteniük (például: egy tanár egyszerre legfeljebb egy órát tart, egy teremben egyszerre legfeljebb egy osztály tartózkodik). Könnyen látható, hogy ha a lehetséges megoldásokat vizsgáljuk, az esetek nagy részében valamelyik feltétel nem teljesül. Mivel a keresési tér legtöbb pontja illegális megoldást reprezentál, a fitneszfüggvény értéke majdnem mindenhol nulla.

Ha valóban 0-nak vesszük a fitneszértékeket az ilyen pontokban, akkor a GA a véletlen kereséshez hasonlóan működik, mivel minden hibás megoldást egyformán rossznak ítél. Az lenne jó, ha az ilyen pontokban a fitneszfüggvény azt fejezné ki, milyen közel van a ponthoz egy legális megoldás. Nyilván nem tudjuk ilyen egyszerűen megadni a fitneszfüggvényt, hiszen a legális megoldások ismeretlenek, pont azokat keressük.

Egyik lehetséges megoldás, ha a célt több kisebb részcéllé bontjuk, és egy pont fitneszértékét a teljesített részcélokból számítjuk ki. Az órarendkészítésnél a részcélok lehetnek az egyes osztályok, tanárok feltételnek megfelelő beosztásai.

Büntetőfüggvény segítségével generált fitneszfüggvény

Az előző probléma másik megoldása, ha azt írjuk le egy büntetőfüggvény segítségével, mennyire rossz az illegális megoldás, mennyi (és milyen fontos) feltételt sért meg.

A fitneszfüggvényt úgy kaphatjuk meg, hogy egy megfelelő konstansból kivonjuk a büntetőfüggvényt. *Richardson* és társai végeztek kutatást a témában, és azt találták ([BBM93a]), akkor jó a büntetőfüggvény, ha azt fejezi ki, milyen költséggel lehet az illegális kromoszómából legálisat készíteni. Büntetőfüggvényre példát a 4.5. részben találhatunk.

Ember, mint fitneszfüggvény

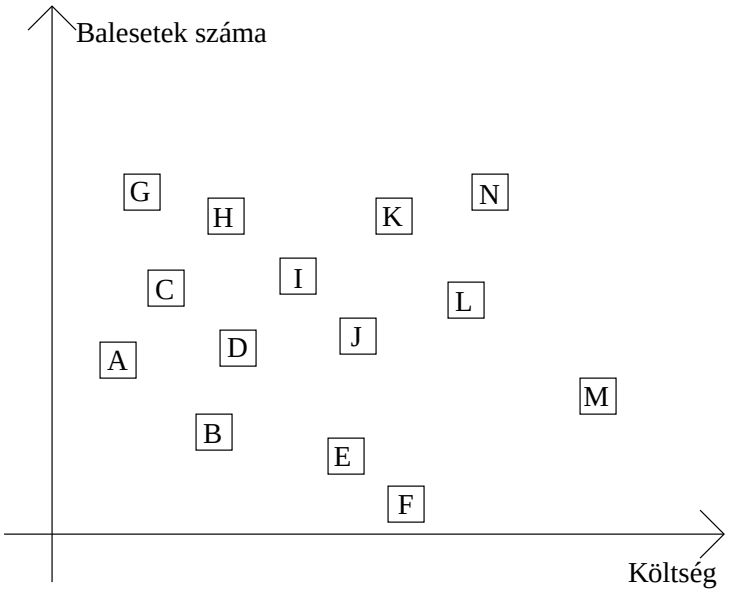
A [BBM93a] cikkben olvashatunk a *C. CaldWell* és *V. S. Johnston* által alkalmazott érdekes fitneszfüggvényről. A feladat rendőrségi fantom-képek készítése. Az emberi arc különböző részekre van bontva (szem, orr, áll, száj, fül . . .), minden alkotórészből számos található az adatbázisban, és a gyanúsítottra leginkább hasonlító képet keressük.

A működés során a GA véletlen arcokat generál, a tanúnak — aki látta a tettest — ki kell választania azt a két képet, amelyik legjobban hasonlít a gyanúsítottra. A következő képsorozatban a képek már jobban hasonlítanak a kiválasztott képekre. Ennél az alkalmazásnál a fitneszfüggvény szerepét a tanú vette át.

Többértékű fitneszfüggvény

Előfordulhat, hogy nem egy függvényt szeretnénk optimalizálni, hanem egyszerre többet, esetleg néhány függvényt optimalizálni, másokat minimalizálni szeretnénk. Azokban az esetekben, amikor a függvényekből össze tudunk állítani egy új fitneszfüggvényt, nincs probléma, hiszen az új fitneszfüggvényt kell maximalizálnunk. Néhány esetben nem tudunk új fitneszfüggvényt képezni. Gondoljunk arra, hogy egyszerre kell minimalizálni a termelési költséget és a munkahelyi balesetek számát. Míg a termelési költség forintban adott, egy munkahelyi baleset nehezen kifejezhető forintban.

Vizsgáljunk meg egy egyszerű példát, ahol két függvényt szeretnénk egyszerre minimalizálni. A 2.7. ábrán láthatunk több lehetséges megoldást. Bár nem tudjuk egyér-



2.7. ábra. Többértékű fitneszfüggvény

telműen eldönteni melyik a legjobb, több megoldást ki tudunk zárni. Észrevehetjük, hogy az A megoldás mindkét függvénynél kisebb értéket ad mint a G megoldás, tehát az A minden vizsgált szempontban jobb a G-nél. Ezt úgy mondjuk, hogy az A dominálja G-t. A domináltság alapján parciális rendezést($<_p$) tudunk definiálni a pontokon (feltételezzük, hogy minden függvény szerint minimalizálunk): $x <_p y \iff (\forall i : x_i \leq y_i) \wedge (\exists i : x_i < y_i)$ Az algoritmus feladata olyan pontok keresése, mely pontokat nem dominál semelyik másik pont sem. A példában négy ilyen pont van, az A, B, E és F. Ezeket a pontokat hívjuk *dominálatlan* pontoknak.

A kérdés az, hogyan találhatjuk meg ezeket a pontokat. *Schaffer* VEGA[2] programja (*Grefenstette* Genesis [Gre90] programjának továbbfejlesztése) a következő elven működött: A populációt több alpopulációra osztotta, és minden alpopulációban egy-egy függvény szerint rangsorolta az egyedeket. A kiválasztás független volt az alpopulációkban, míg a párosodás már különböző alpopulációban található egyedek között történt. A módszer hibája, hogy vannak olyan egyedek, melyek egyik függvényt vizsgálva sem érnek el kiemelkedő eredményt, ugyanakkor az összes függvényt együtt vizsgálva már sikeresek lehetnek. (Hasonlóan a tízpróbázókhoz, akik a tíz közül egyik számban sem tudják megközelíteni a szám specialistáit, ugyanakkor tízpróbában sikeresebbek a specialistáknál.)

A *Baker* által javasolt *dominálatlan rendező módszer* már egyenlő esélyt nyújt a dominálatlan pontoknak. A módszer leginkább a fitnesz rangsoroláshoz (2.9.2. rész) hasonlít.

[2]Vector Evaluated Genetic Algorithm

Először megkeressük a dominálatlan egyedeket, ezek mind egyes sorszámot kapnak. A kettes sorszámú egyedek megkereséséhez először eltávolítjuk a populációból az egyes sorszámú egyedeket, és az így kapott populációban keressük meg a dominálatlan egyedeket. Az így kapott egyedek lesznek a kettes sorszámúak. A kettes sorszámú egyedek eltávolítása után keletkező populáció dominálatlan egyedei a hármas sorszámúak. Az eljárást addig folytatjuk, míg el nem fogy az összes elem.

A sorszámok kiosztása után a kiválasztás már a sorszámoktól függ, minél kisebb egy egyed sorszáma, annál nagyobb eséllyel kerül kiválasztásra. Az ábrán (2.7.) látható példában egyes sorszámot kap az A, B, E, F, kettes sorszámot a G, C, D, M, hármas sorszámot a H, I, J, négyes sorszámot a K, L, végül ötös sorszámot az N.

Mivel a többértékű fitneszfüggvényeknél általában több lehetséges megoldás is van (a dominálatlan egyedek), hasznos, ha a GA nemcsak egy, hanem lehetőleg minél több lehetséges megoldást megad, ezért a módszert gyakran használják együtt az élettér felosztással, és fajokra tagolódással (3.4. rész).

2.9.2. Szülőválasztási technikák

Tudjuk, hogy a szülők kiválasztásánál előnyt élveznek a magasabb fitneszértékkel rendelkező egyedek. Az egyik legegyszerűbb módszernél, a rulettkerék módszernél (2.5. rész) a kiválasztás a fitneszértékkel arányos. Bizonyos esetekben ez problémához vezethet.

A kezdeti populációban többnyire találunk néhány — a többi egyedhez képest — kiemelkedő egyedet. Ekkor ez a néhány egyed nagyon hamar dominánssá válhat. Ezt nevezik *túl korai konvergenciának*. Ilyenkor a keresési tér csak kis részét vizsgálja az algoritmus, és valószínűleg csak lokális maximumot ad. Az esetet úgy tudjuk kiküszöbölni, ha a kiemelkedő egyedeknek arányosan csökkentjük az esélyeit.

A másik probléma bizonyos értelemben az előző probléma ellentetje. Ha az algoritmus már sok generációt vizsgált, akkor a populáció már nagy mértékben konvergált, kicsi az eltérés az egyedek között. Mivel kicsi az eltérés az egyedek fitneszértéke között is, ezért a legjobb egyedek kiválasztásának valószínűsége alig nagyobb a gyengébb egyedekénél. Ilyenkor a keresés alig jobb mint egy egyszerű véletlen keresés. Itt a megoldás az, ha a jobb egyedeknek növeljük az esélyeit.

Az említett problémák megoldására több módszer ismert. A módszerek egy részénél a fitneszfüggvény értékét módosítják, és a módosított értékek alapján történik a kiválasztás (ezek az explicit fitneszfüggvény leképezések), más részénél nem módosítják a fitneszfüggvények értékét, hanem másként érnek el hasonló hatást (ezek az implicit fitneszfüggvény leképezések).

Explicit fitneszfüggvény leképezések

Fitnesz skálázás. Elterjedt módszer a lineáris fitnesz skálázás. A módszer célja, hogy a fitneszértékek megváltoztatásával (egy lineáris függvény segítségével), elérje, hogy

a legjobb elem fitneszértéke az átlagos fitneszérték kontansszorosa legyen. Jelöljük a kontanst C_{mult}-tal. A konstans értékére többnyire 1.2 és 2.0 közötti értékeket választanak, leggyakrabban 2.0-t. A továbbiakban a [Gol89] könyvben bemutatott skálázást vizsgáljuk meg. A lineáris skálázásban az új fitneszfüggvényt (f'), a következőképpen kapjuk meg az eredeti (f) fitneszfüggvényből: $f' = a \cdot f + b$. A feladat a és b értékének helyes megválasztása. Jelöljük az átlagos fitneszértéket f_{avg}-vel, a maximálisat f_{max}-szal, a minimálisat f_{min}-nel. A skálázásra a következő egyenleteknek kell teljesülni:

$$f'_{max} = C_{mult} \cdot f'_{avg} \quad (2.1)$$
$$f'_{avg} = f_{avg} \quad (2.2)$$

További feltétel, hogy a fitneszfüggvény értékeinek nemnegatívnak kell maradniuk. Feltéve hogy $C_{mult} > 1$, ez teljesül, ha $f'_{min} \geq 0$. Amennyiben a és b értékeit a következőképp választjuk:

$$a = (C_{mult} - 1) \cdot \frac{f_{avg}}{f_{max} - f_{avg}}$$
$$b = f_{avg} \cdot \frac{f_{max} - C_{mult} \cdot f_{avg}}{f_{max} - f_{avg}}$$

könnyen belátható, hogy a 2.1. és a 2.2. egyenletek teljesülnek:

$$\begin{aligned} f'_{max} &= \frac{f_{max} \cdot (C_{mult} - 1) \cdot f_{avg}}{f_{max} - f_{avg}} + \frac{f_{avg}(f_{max} - C_{mult} \cdot f_{avg})}{f_{max} - f_{avg}} \\ &= \frac{f_{max} \cdot C_{mult} \cdot f_{avg} - f_{avg}^2 \cdot C_{mult}}{f_{max} - f_{avg}} = \frac{C_{mult} \cdot f_{avg}(f_{max} - f_{avg})}{f_{max} - f_{avg}} \\ &= C_{mult} \cdot f_{avg} \\ f'_{avg} &= \frac{f_{avg} \cdot (C_{mult} - 1) \cdot f_{avg}}{f_{max} - f_{avg}} + \frac{f_{avg}(f_{max} - C_{mult} \cdot f_{avg})}{f_{max} - f_{avg}} \\ &= \frac{f_{avg} \cdot f_{max} - f_{avg}^2}{f_{max} - f_{avg}} = \frac{f_{avg}(f_{max} - f_{avg})}{f_{max} - f_{avg}} = f_{avg} \end{aligned}$$

Az f'_{min}-re vonatkozó feltétel ($f'_{min} \geq 0$) sajnos nem teljesül mindig:

$$\begin{aligned} f'_{min} = \frac{f_{min} \cdot (C_{mult} - 1) \cdot f_{avg}}{f_{max} - f_{avg}} + \frac{f_{avg}(f_{max} - C_{mult} \cdot f_{avg})}{f_{max} - f_{avg}} &> 0 \iff \\ f_{min} \cdot (C_{mult} - 1) \cdot f_{avg} + f_{avg}(f_{max} - C_{mult} \cdot f_{avg}) &> 0 \iff \\ f_{min} \cdot (C_{mult} - 1) + (f_{max} - C_{mult} \cdot f_{avg}) &> 0 \end{aligned}$$

Amennyiben az egyenlőtlenség nem teljesül, akkor nem tudjuk végrehajtani a skálázást a megadott feltételekkel. Ekkor C_{mult} értékét annyira lecsökkentjük, hogy $f'_{min} = 0$ és

$f'_{avg} = f_{avg}$ teljesüljön. Ekkor a következő a, b és C_{mult} értéket kell választani:

$$\begin{aligned} a &= \frac{f_{avg}}{f_{avg} - f_{min}} \\ b &= \frac{-f_{min} \cdot f_{avg}}{f_{avg} - f_{min}} \\ C_{mult} &= \frac{f_{max} - f_{min}}{f_{avg} - f_{min}} \end{aligned}$$

Fitnesz ablakozás. A módszert *Grefenstette* Genesis [Gre90] programjából ismerhetjük meg. A módszer alapja egy, az előbb ismertetettnél egyszerűbb lineáris skálázás. A lineáris skálázás ezen változatánál minden fitneszértékből kivonnak egy számot, így növelve a legjobb és az átlagos fitneszérték közti arányt.

A fitnesz ablakozás során minden lépésben fel kell jegyezni a legkisebb fitneszértékkel rendelkező egyed fitneszértékét. A feljegyzett értékekből csak az utolsó n darabot vesszük figyelembe. Az n számot nevezzük az ablak méretének (n általában 10). A fitneszértékekből kivonandó szám az utolsó n minimumérték minimuma.

Fitnesz rangsorolás. A módszernél az egyedeket sorbarendezik fitneszértékük alapján, és a továbbiakban a sorrendben elfoglalt hely alapján történik a kiválasztás. A sorbarendezés miatt így teljesen mindegy, hogy mennyivel volt jobb az egyik egyed a másiknál, így a kiemelkedően jó egyedek sem tudnak túlzottan elszaporodni. A módszer hasznos akkor is, ha túl kicsi volt a különbség az egyedek fitneszértékei között.

A sorrend alapján történő leképezés történhet akár lineárisan, akár exponenciálisan is.

Implicit fitneszfüggvény leképezések

Bajnokság. Ha a szülőket bajnokság segítségével választjuk ki, nincs szükség a fitneszfüggvény értékének módosítására. A legegyszerűbb bajnokság esetén kiválasztunk véletlenszerűen két egyedet a populációból, majd a magasabb fitneszértékkel rendelkezőt helyezzük a szülők közé. Ezt addig ismételjük, míg a kívánt számú szülőt meg nem kapjuk. Lehetőségünk van nagyobb bajnokságra is, ha nem 2, hanem több egyed közül választjuk ki a legjobbat szülőnek.

További továbbfejlesztési lehetőség, ha a legnagyobb fitneszértékű egyed nem mindig, csak $p\ (0.5 < p < 1)$ valószínűséggel nyeri meg a bajnokságot. A bajnokság méretének, és a győzelmi esélynek változtatásával könnyen lehet növelni vagy csökkenteni a nagyobb fitneszértékű egyedek kiválasztásának esélyeit.

A módszer különösen hasznos azokban az esetekben, ahol az egyedekhez nehezen tudunk fitneszértéket rendelni, ugyanakkor könnyen el tudjuk dönteni, hogy két egyed közül melyik a jobb. Ha például az egyedek különböző stratégiájú amőba (sakk, go,

othello ...) játékosokat reprezentálnak, akkor két egyed összehasonlítása egyszerű, hiszen csak egymás ellen kell őket játszatni, miközben egy egyedhez nehezen tudnánk fitneszértéket rendelni.

2.10. A séma elmélet

A genetikus algoritmusok vizsgálata során eljutottunk arra a pontra, amikor már az olvasó tisztában van a genetikus algoritmusok működésével, képes lenne számítógépen implementálni az algoritmust. Tudjuk miként működik az algoritmus, de nem vizsgáltuk még azt, hogy miért működik.

Bár a genetikus algoritmusok alapja a biológiai evolúció, érdemes megvizsgálni a matematikai alapokat. A genetikus algoritmusok alapvető elméletét *Holland* dolgozta ki 1975-ben.

Először is a séma fogalmát kell megismernünk. Jelöljük a kromoszóma hosszát L-el. A kromoszómák a {0, 1} ábécé feletti L hosszúságú szavak. Bővítsük ki az ábécé-t a $*$ jellel. Az így kapott {0, 1, $*$} ábécé feletti L hosszúságú szavakat nevezzük sémáknak. Mivel az ábécé elemszáma 3, összesen 3^L séma létezik. Azt mondjuk, hogy egy string megfelel a sémának, ha a sémában lévő 1-esek helyén a stringben is 1-esek találhatók, a sémában lévő 0-k helyén 0-k találhatók. (A $*$ helyén tehát mind 0, mind 1 állhat.) Tehát a $H = **10**1*$ sémának megfelelnek a 00100010, 00100011, ... stringek. Ahogyan egy sémának több string is megfelelhet, úgy minden string több sémának felel meg. (Pontosan 2^L-nek, mivel minden helyen a string megfelelő eleme vagy $*$ állhat)

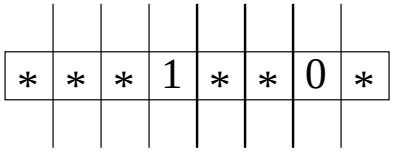
Látható, hogy minél több $*$ van a sémában, annál több string felel meg neki. A H sémában lévő 1-esek és 0-k számának összegét (azaz a fix pozíciók számát) nevezzük a séma rangjának és jelöljük $o(H)$-val. Tehát például $o(**10**1*) = 3$. A séma rangjának segítségével már le tudjuk írni az egyes sémáknak megfelelő stringek számát: Egy H sémának $2^{(L-o(H))}$ string felel meg.

A H sémában található legelső és legutolsó fix (0 vagy 1) elem indexének különbségét a séma meghatározó hosszának nevezzük, és $\delta(H)$-val jelöljük. Tehát például $\delta(**10**1*) = 4$, mivel az első fix elem indexe 3, az utolsóé 7, és $7 - 3 = 4$. Az egyetlen fix elemmel rendelkező sémáknál az első és utolsó fix elem megegyezik, így a meghatározó hossz 0. (például $\delta(***1*****) = 0$)

Azon stringek számát a $t.$ lépés utáni populációban, melyek megfelelnek a H sémának $m(H,t)$-vel jelöljük.

2.10.1. Kiválasztás

Akkor mondhatnánk, hogy az algoritmus jól működik, ha a jó sémáknak, az idő múlásával egyre több string felel meg. Vizsgáljuk meg azt az esetet először, ahol nincsen mutáció és keresztezés, csak kiválasztás. Minél nagyobb fitneszérték tartozik a



2.8. ábra. Lehetséges kereszteződési pontok

stringhez, annál nagyobb valószínűséggel választjuk ki. Ha a kiválasztásnál a rulettkerék módszert (2.5. rész) használjuk, akkor az A_i string kiválasztásának valószínűsége $p_i = f_i / \sum_{j=1}^{n} f_j$, ahol f_i az i. stringhez tartozó fitneszérték (n jelöli a populáció méretét). Mivel az új populáció is n elemből áll, a H sémának megfelelő stringek száma a következőképpen módosul a $t.$ és $t+1.$ időpillanat között:

$$m(H, t+1) = \frac{m(H,t) \cdot \ n \cdot \ f(H)}{\sum_{j=1}^{n} f_j} \tag{2.3}$$

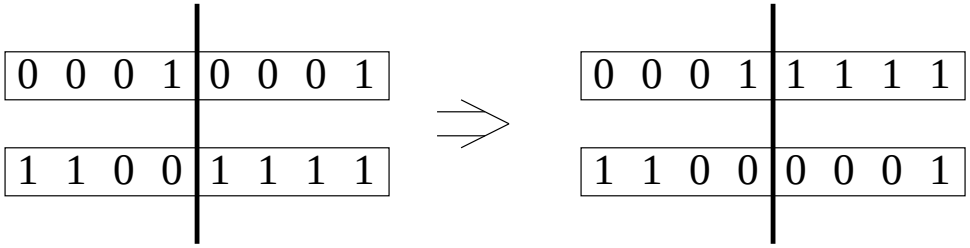
A H sémának megfelelő stringek átlagos fitneszértékét jelöltük $f(H)$-val. Észrevehetjük, hogy $\sum_{j=1}^{n} f_j$ / n a populáció átlagos fitneszértéke, jelöljük ezt $\overline{f}$-sal. Így kapjuk a következő képletet:

$$m(H, t+1) = m(H,t)\frac{f(H)}{\overline{f}} \tag{2.4}$$

A képletből látható, hogy egy adott sémának megfelelő stringek száma olyan arányban nő, mint a sémának megfelelő, és az egész populáció átlagos fitneszértékének aránya. Tehát, ha a sémának megfelelő stringek átlagos fitneszértéke magasabb a populáció átlagos fitneszértékénél, akkor elterjed a séma, míg ha kisebb, akkor a séma fokozatosan eltűnik.

2.10.2. Keresztezés

Vizsgáljuk most a kiválasztás mellett a keresztezést is. Az egyszerűség kedvéért az 1-pontos keresztezést elemezzük. Nyilvánvaló, hogy egy $*****1**$ sémának megfelelő egyeden hiába hajtunk végre keresztezést, az egyik utód meg fog felelni a sémának. Ha egy másik sémát vizsgálunk, például a $***1**0*$ sémát, akkor ha a 2.8. ábrán vastag vonallal jelölt helyeken lesz a kereszteződési pont, akkor lehetséges, hogy az egyik utód sem fog megfelelni a sémának (például a 2.9. ábrán az egyik utód sem felel meg a $***1**0*$ sémának), míg ha a vékony vonallal jelölt helyeken lesz a kereszteződési pont, akkor valamelyik utód biztosan megfelel a sémának. Tehát legalább 4/7 valószínűséggel fog az egyik utód megfelelni a sémának. Ha egy olyan sémát választunk, amelyben mindkét végpontnál fix érték van (pl. 1**0*100), akkor bárhol lesz a kereszteződési pont, nem garantált, hogy valamelyik utód megfelel a sémának. Látható, hogy



2.9. ábra. A keresztezés hatása a sémára

annak a valószínűsége, hogy egy adott séma túléli a keresztezést (jelöljül p_s-el), a séma meghatározó hosszától függ.

$$p_s \geq 1 - \frac{\delta(H)}{L-1} \tag{2.5}$$

Ha a kiválasztás után nem mindig következik keresztezés, csak az esetek p_c részében, akkor a 2.5. képlet a következőképp módosul:

$$p_s \geq 1 - p_c\frac{\delta(H)}{L-1} \tag{2.6}$$

Ha a 2.4. egyenletet módosítani szeretnénk úgy, hogy kezelje a keresztezés miatt elromló sémákat, akkor a 2.6. egyenlőtlenséggel kell kombinálnunk:

$$m(H,t+1) \geq m(H,t)\frac{f(H)}{\overline{f}}\left[1 - p_c\frac{\delta(H)}{L-1}\right] \tag{2.7}$$

A keresztezés hatására nemcsak elromolhatnak a sémák, hanem elképzelhető, hogy valamelyik utód olyan sémának is megfelel, amilyennek egyik szülője sem felelt meg (például a 2.9. ábrán, bár semelyik szülő sem felel meg a $***11111$ sémának, a felső utód megfelel). Így ha a 2.4. egyenlethez hasonlóan itt is egyenlőséget szeretnénk felírni, akkor még egy bonyolult pozitív tagra is szükségünk lenne. Ezt a tagot itt egyszerűen elhagyjuk, ezért kapunk egyenlőtlenséget.

2.10.3. Mutáció

A kiválasztás és a keresztezés után most vizsgáljuk meg a mutáció hatását. Egy adott bit mutációjának valószínűségét p_m jelöli. Egy séma akkor romlik el a mutáció hatására, ha a séma fix (0 vagy 1) pozícióján következik be mutáció. Egy fix pozíción $1 - p_m$ valószínűséggel nem következik be mutáció. Mivel egy sémában $o(H)$ fix pozíció van, annak a valószínűsége, hogy egyik fix pozíción sem következik be mutáció $(1 - p_m)^{o(H)}$. Mivel p_m értéke többnyire nagyon kicsi (pl. 0.001), ezért $(1 - p_m)^{o(H)}$

értékét közelíthetjük egy egyszerűbb kifejezéssel is: $1 - o(H)p_m$. A 2.7. egyenlőtlenséget újra módosítva kapjuk meg azt az egyenlőtlenséget (2.8.), amely már a mutáció hatását is tartalmazza.

$$m(H, t+1) \geq m(H, t)\frac{f(H)}{\overline{f}}\left[1 - p_c\frac{\delta(H)}{L-1} - o(H)p_m\right] \tag{2.8}$$

A mutáció figyelembevételénél is elhanyagoltuk azt a hatást, hogy mutáció hatására létrejöhet olyan string, mely megfelel a sémának, noha egyik szülője sem felelt meg. Elhanyagoltuk továbbá azt a tényt, hogy egy sémát elronthat egyszerre a keresztezés és a mutáció is. Az ilyen esetek valószínűségét levontuk mind a keresztezésnél, mind a mutációnál is, helyesen csak egyszer kellett volna levonnunk.

Az utolsó egyenlőtlenségből leolvasható, hogy a séma elmélet szerint milyen tulajdonságú sémák terjednek el legnagyobb valószínűséggel. Azt már megállapítottuk, hogy a sémának megfelelő stringek átlagos fitneszértéke nagy kell hogy legyen (az átlagos fitneszértéknél nagyobb), a 2.8. egyenlőtlenségből látszik az is, hogy azon sémák sikeresek, melyek rövidek (meghatározó hosszuk ($\delta(H)$) kicsi), és alacsony rangúak ($o(H)$ is kicsi).

A séma elmélet következményeként szokás felírni az *építőkocka hipotézist*. Eszerint a GA az építőkockának nevezett alacsony rangú, rövid, magas fitneszű sémák összeillesztésével találja meg a közel optimális megoldást.

2.11. Minimális csalfa probléma

Próbáljunk egy olyan egyszerű feladatot készíteni, mely becsapja a GA-t, vagyis egy olyan feladatot, ahol a GA nem képes megtalálni a globális optimumot. A feladat leírása a [Gol89] könyvben található, az ábrák is a könyvből származnak. Az építőkocka hipotézist felhasználva olyan problémát próbálunk készíteni, ahol rövid, alacsony-rangú blokkok hibás (szuboptimális), hosszú, magas-rangú blokkokhoz vezetnek.

A vizsgálandó sémák rangja 2, vagyis két bit kivételével minden pozícióban * található. A két bit pozíciója nem lényeges, de minden sémában azonos. A két bit pozíciójának távolsága $\delta(H)$.

$$\begin{array}{ll} ****0****0**** & f_{00} \\ ****0****1**** & f_{01} \\ ****1****0**** & f_{10} \\ ****1****1**** & f_{11} \end{array}$$

A fitnesz értékek a sémához tartozó átlagos értékek. Tegyük fel, hogy a négy érték közül f_{11} a legnagyobb:

$$f_{11} \quad > \quad f_{00} \tag{2.9}$$

$$f_{11} > f_{01} \tag{2.10}$$
$$f_{11} > f_{10} \tag{2.11}$$

Ahhoz, hogy a rövid sémák félrevezessék a GA-t, arra van szükség, hogy két rövid sémát összehasonlítva, az optimális sémához tartozó rövid séma fitnesze legyen alacsonyabb. Vagyis azt szeretnénk, ha a következő két egyenlőtlenség igaz lenne:

$$f_{0*} > f_{1*} \tag{2.12}$$
$$f_{*0} > f_{*1} \tag{2.13}$$

A két egyenlőtlenség nem teljesülhet, hiszen

$$f_{0*} > f_{1*} \Leftrightarrow f_{00} + f_{01} > f_{10} + f_{11} \tag{2.14}$$
$$f_{*0} > f_{*1} \Leftrightarrow f_{00} + f_{10} > f_{01} + f_{11} \tag{2.15}$$

A két egyenlőtlenséget összeadva $f_{00} > f_{11}$ adódna, ami ellentmond 2.11. egyenlőtlenségnek.

Mivel a két egyenlőtlenség egyszerre nem teljesülhet, feltételezzük, hogy csak 2.14. igaz.

Az egyszerűség kedvéért normalizáljuk a fitneszértékeket, és így kapjuk a következő értékeket:

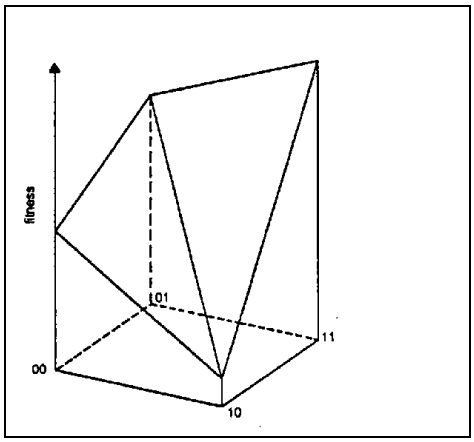
$$r = \frac{f_{11}}{f_{00}} \tag{2.16}$$
$$c = \frac{f_{01}}{f_{00}} \tag{2.17}$$
$$c' = \frac{f_{10}}{f_{00}} \tag{2.18}$$

Ezeket felhasználva a korábbi egyenlőtlenségeket egyszerűbb formában felírhatjuk. A globális maximumra vonatkozó egyenlőtlenségek:

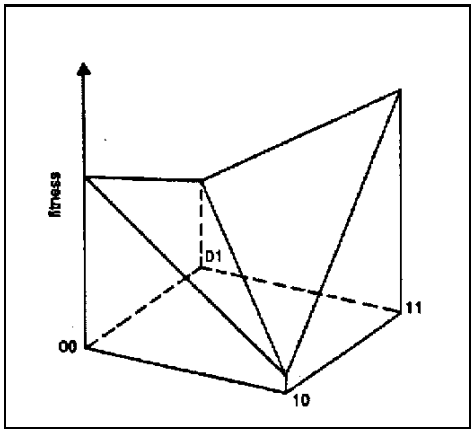
$$r > c \tag{2.19}$$
$$r > 1 \tag{2.20}$$
$$r > c' \tag{2.21}$$

A 2.14. egyenlőtlenség:

$$r < 1 + c - c' \tag{2.22}$$



2.10. ábra. I. típus



2.11. ábra. II. típus

2.19. és 2.22. egyenlőtlenségekből, illetve 2.20. és 2.22. egyenlőtlenségekből következik:

$$c' < 1 \tag{2.23}$$

$$c' < c \tag{2.24}$$

Észrevehető, hogy a probléma két alesetre bontható:

- I. $f_{01} > f_{00} (c > 1)$
- II. $f_{01} \leq f_{00} (c \leq 1)$

Az első típusra a 2.10., a második típusra a 2.11. ábrán láthatunk példát.

A probléma vizsgálatához a séma elméletet is felhasználhatjuk. Ha a mutációt elhanyagoljuk, akkor 2.6. egyenlőtlenség alapján a H_{11} séma előfordulásának gyakorisága a $t+1$-edik generációban:

$$P_{11}^{t+1} \geq P_{11}^{t} \frac{f_{11}}{\bar{f}} \cdot \left[1 - p_c \frac{\delta(H_{11})}{L-1} \right] \tag{2.25}$$

Sajnos mivel nem egyenlőségünk, csak egyenlőtlenségünk van, nem tudjuk mindig megállapítani, hogy hová tart P_{11}. A sémaelmélet szerint azért nem lehet egyenlőséget írni, mert a keresztezés során időnként létrejöhetnek olyan sztringek, melyeknél az egyik szülő sem felelt meg a sémának, de a sztring megfelel. Például egy H_{00}-ás és egy H_{11}-es sztring keresztezéséből létrejöhet egy H_{01}-es, és egy H_{10}-ás sztring. Az esetek nagy részében azonban ettől nem kell tartani, ha H_{11}-es párosodik H_{10}-ával, akkor a leszármazottak is csak H_{11}-esek, vagy H_{10}-ások lehetnek.

A 2.25. egyenlőtlenség másik pontatlansága, hogy nem kellene minden esetben csökkenteni az P_{11} értékét keresztezésnél, csak akkor, ha a másik szülő H_{00}-es volt, hiszen ellenkező esetben az egyik gyerek H_{11}-es sémájú lesz.

A következő táblázat összefoglalja, hogy mely esetekben milyen leszármazottak jöhetnek létre. Csak azokat a leszármazottakat tünteti fel a táblázat, ahol a leszármazott nem egyezik meg valamely szülővel.

	00	01	10	11
00	-	-	-	01,10
01	-	-	00,11	-
10	-	00,11	-	-
11	10,01	-	-	-

A táblázatot felhasználva már nemcsak egyenlőtlenséget, hanem egyenlőségeket is fel lehet írni.

$$P_{11}^{t+1} = P_{11}^t \cdot \frac{f_{11}}{\bar{f}} \left[1 - p_c' \frac{f_{00}}{\bar{f}} P_{00}^t\right] + p_c' \frac{f_{01} f_{10}}{\bar{f}^2} P_{01}^t P_{10}^t \tag{2.26}$$

$$P_{10}^{t+1} = P_{10}^t \cdot \frac{f_{10}}{\bar{f}} \left[1 - p_c' \frac{f_{01}}{\bar{f}} P_{01}^t\right] + p_c' \frac{f_{00} f_{11}}{\bar{f}^2} P_{00}^t P_{11}^t \tag{2.27}$$

$$P_{01}^{t+1} = P_{01}^t \cdot \frac{f_{01}}{\bar{f}} \left[1 - p_c' \frac{f_{10}}{\bar{f}} P_{10}^t\right] + p_c' \frac{f_{00} f_{11}}{\bar{f}^2} P_{00}^t P_{11}^t \tag{2.28}$$

$$P_{00}^{t+1} = P_{00}^t \cdot \frac{f_{00}}{\bar{f}} \left[1 - p_c' \frac{f_{11}}{\bar{f}} P_{11}^t\right] + p_c' \frac{f_{01} f_{10}}{\bar{f}^2} P_{01}^t P_{10}^t \tag{2.29}$$

A fitneszátlagot $\bar{f}$ jelöli, ez esetben ezt explicit módon ki tudjuk fejezni, hasonlóan p_c'-höz:

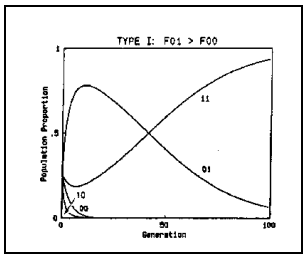
$$\bar{f} = P_{00}^t f_{00} + P_{01}^t f_{01} + P_{10}^t f_{10} + P_{11}^t f_{11} \tag{2.30}$$

$$p_c' = p_c \cdot \frac{\delta(H)}{l-1} \tag{2.31}$$

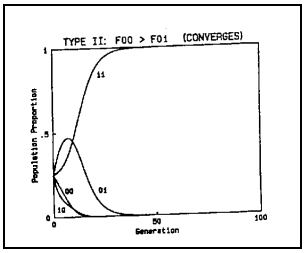
Az előző egyenletek segítségével szimulálható a négy séma arányának változása a populációban. Abban az esetben mondhatjuk, hogy a GA az optimális megoldáshoz konvergál, ha a következő feltétel teljesül:

$$\lim_{t \to \infty} P_{11}^t = 1 \tag{2.32}$$

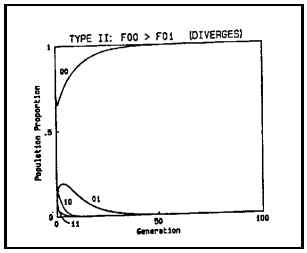
A számítógépes szimulációk azt mutatják, hogy az első típusú eset, ha a kezdő populációban egyik séma aránya sem nulla, teljesíti a feltételt, vagyis a GA megtalálja az optimális megoldást (pl. 2.12. ábra).



2.12. ábra. I. típus konvergál



2.13. ábra. II. típus konvergál



2.14. ábra. II. típus divergál

A második típusnál az esetek nagy részében (pl. 2.13. ábra) a konvergencia biztosított, de ha H_{00} aránya nagyon magas a kezdeti populációban, akkor a GA H_{00}-hoz konvergál (pl. 2.14. ábra), vagyis nem találja meg az optimális megoldást. Goldberg szerint viszonylag egyszerű szabályokkal meghatározható mikor konvergál a GA.

2.12. Implicit párhuzamosság

Már a GA kutatásának elején bebizonyították, hogy egy lépés alatt, ha a populáció mérete n, akkor $O(n^3)$ sémát vizsgál meg a Genetikus Algoritmus. Ennek a fontos eredménynek *Holland* külön nevet adott, ezt hívjuk implicit párhuzamosságnak. Ez egyike azon kevés esetnek, amikor a kombinatorikus robbanás a segítségünkre van.

Az itt leírt bizonyítás a [Gol89] könyvön alapszik, de több esetben annál bővebb.

Vegyünk egy n elemű populációt, melyben egy egyedet l bittel ábrázolunk. Csak azokat a sémákat vesszük figyelembe, mely túlélési valószínűsége nagyobb egy előre meghatározott p_s konstansnál,vagyis $\varepsilon < 1-p_s$ valószínűséggel veszik el. Ha egyszerű keresztezést, és alacsony mutációs rátát használunk, akkor belátható, hogy csak az $l_s < \varepsilon(l-1)+1$ hosszúságú sémákat kell figyelembe vennünk. ($\varepsilon = 0$ és $\varepsilon = 1$ értékére könnyen bizonyítható, e kettő között közel lineáris).

Tegyük fel hogy $l = 10$ hosszú sztringben próbáljuk a legfeljebb $l_s = 5$ hosszú sémákat megszámolni.

$$\boxed{1\,0\,1\,0\,1}\,1\,1\,0\,1\,0$$

Először megszámoljuk azokat a sémákat, ahol az első 5 bitet leszámítva $*$ van a sémában. További megszorításként csak azokat számoljuk, ahol az 5. helyen fix érték van.

$$\boxed{?\,?\,?\,?\,1}\,*\,*\,*\,*\,*$$

A ?-ek helyére írhatjuk a sztringben szereplő bitet, vagy $*$-ot. Mivel kérdőjelből $l_s - 1$ darab van, így összesen 2^{l_s-1} sémát számoltunk meg eddig.

1 0 1 0 1 1 1 0 1 0

Az $l_s = 5$ méretű ablakot el tudjuk tolni eggyel jobbra, és itt újra 2^{l_s-1} sémát tudunk megszámolni. Mivel az ablak legjobboldalibb pozíciójában mindig fix érték van, egyik sémát sem számoljuk egynél többször. Az ablak eltolását összesen $l-l_s+1$ alkalommal tudjuk megtenni, így egy darab sztringben $(l - l_s + 1)2^{l_s-1}$ sémát számoltunk meg.

Ha a populációban lévő n darab egyedet vesszük, akkor nem szorozhatjuk meg az előbbi számot egyszerűen a populáció méretével, hiszen egy rövidebb sémát több egyednél is számolhatnánk. Mivel a különböző hosszú sémák száma binomiális eloszlást követ, a sémák fele hosszabb, fele pedig rövidebb mint $l_s/2$. Mi a továbbiakban csak az $l_s/2$-nél hosszabb sémákat számoljuk. Ha a populáció méretét $n = 2^{l_s/2}$-nek választjuk, akkor egy legalább $l_s/2$ hosszú séma várhatóan legfeljebb egyszer szerepel a populációban. Vagyis a populációban lévő sémák száma $n(l - l_s + 1)2^{l_s-1}/2 = n(l - l_s + 1)2^{l_s-2}$. Felhasználva, hogy $n = 2^{l_s/2}$, azt kapjuk, hogy a sémák száma $\frac{(l-l_s+1)}{4}n^3$

3. fejezet

Fejlett technikák

3.1. A gén lókuszának kezelése

A génekről szóló részben (2.2.1.) már megemlítettük, hogy a gén funkcióját meg kell különböztetni a helyétől, vagyis *lókuszától*. A genetikus algoritmusok egyszerűbb megvalósításaiban a gén funkciója mégis megegyezik a kromoszómában elfoglalt helyével, vagyis a gének sorrendje állandó. Lehetőség van azonban arra, hogy a gén funkcióját ne a gén kromoszómában elfoglalt helye határozza meg.

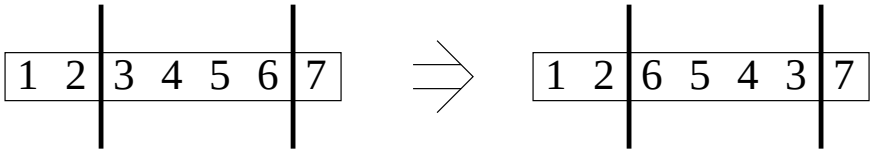
Felmerülhet a kérdés, szükség van-e egyáltalán a gének funkciójának és lókuszának megkülönböztetésére? Miért lesz ettől jobb az algoritmus? A megkülönböztetés csupán azt jelenti, hogy a gének sorrendje nem állandó, hanem változhat az algoritmus során. Minden egyes génfunkciót is kezelő kromoszómának megfelel egy génfunkciót nem kezelő kromoszóma, sőt minden egyes génfunkciót nem kezelő kromoszómának $L!$ génfunkciót kezelő kromoszóma felel meg. (L a kromoszóma hossza.) Látszólag tehát csak feleslegesen bonyolítottuk a problémát. A génfunkció kezelésének azért van mégis haszna, mert fontos a gének sorrendje. Láttuk (2.6.1. rész), hogy jó génsorrendnél hatékonyabban működik a keresztezés (feltéve, hogy nem uniform keresztezést használunk (2.8.3. rész)), és ezáltal az algoritmus is. A génfunkció kezelésével az algoritmus kipróbálhatja a lehetséges génsorrendeket, s mivel a sikeresebb génsorrenddel rendelkező egyedek sikeresebbek, jobban elterjednek, így az algoritmus saját maga találja meg a megfelelő génsorrendet.

A génfunkció kezeléséhez szükséges, hogy ne csak a gén értékét, hanem a gén funkcióját is eltároljuk. Az ábrázolás során legtöbbször a gének értékei fölé írt számokkal jelezzük a funkciót. A 3.1. ábrán láthatunk egy olyan esetet, ahol a gének funkcióját a gén kromoszómában elfoglalt helye határozza meg. Az algoritmus elején többnyire ilyen kromoszómákkal kezd az algoritmus dolgozni, és később a mutációk, kereszteződések hatására keverednek össze a gének.

A továbbiakban csak a funkció kezelésére összpontosítunk, ezért nem jelenítjük meg a gének értékét, csak a funkció sorszámát. A gének értékét állandónak tekintjük.

1	2	3	4	5	6	7
1	0	0	1	1	0	1

3.1. ábra. Gének funkciói és értékei



3.2. ábra. Inverzió

Az eddig megismert — gének értékét módosító — mutációkat a most ismertetendőekkel egyszerre is használhatjuk. Az eddig megismert keresztezések sajnos hibásan működnek, így helyettük új keresztezési módszereket kell használnunk.

A gén funkciójának eltárolásával lehetőség nyílik arra is, hogy a fitneszérték kiszámításakor ne csak a gének értékeit vegyük figyelembe, hanem a gének sorrendjét is.

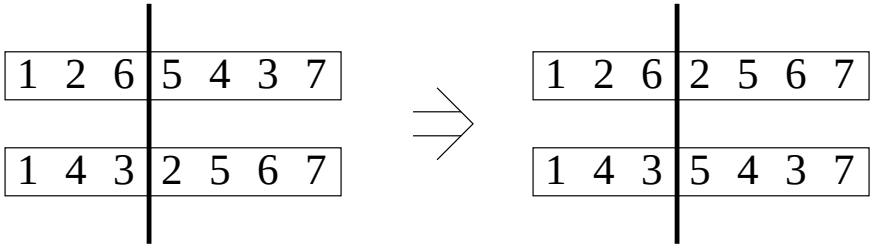
3.1.1. Mutáció

A gének sorrendjének mutációjára legtöbbször az *inverziót* használják. Az inverzió lényege, hogy a kromoszóma két véletlenül kiválasztott pontja között lévő kromoszómadarab megfordul (3.2. ábra). A jelenség ismert a természetben is.

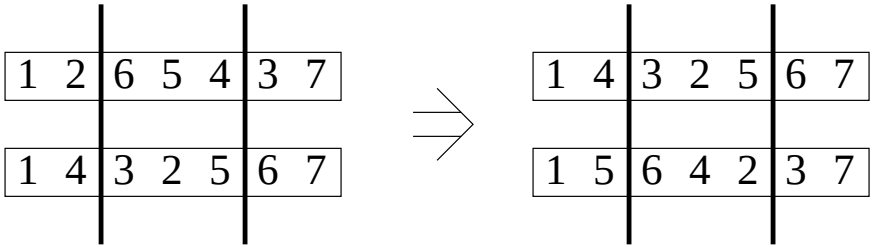
Nézzük meg, mi a valószínűsége annak, hogy egy gén elmozdul a helyéről az inverzió hatására. Ez csak abban az esetben következhet be, ha a két véletlenül kiválasztott hely egyike a gén előtt, másika a gén után található. Ha egy a kromoszóma közepén lévő gént vizsgálunk, ez a valószínűség közel 1/2, míg ha a kromoszóma szélén lévő gént vizsgálunk, akkor $\frac{2}{L} - \frac{1}{L^2}$. Látható, hogy a kromoszóma szélén lévő géneknél kisebb az elmozdulás valószínűsége, mint a kromoszóma közepén lévő géneknél.

Ha ki szeretnénk küszöbölni ezt a különbséget, azt legegyszerűbben úgy tehetjük meg, hogy a kromoszómát nem mint egy lineáris stringet képzeljük el, hanem mint egy génekből készült nyakláncot (hasonlóan a 2-pontos keresztezéshez (2.8.2. rész)). Mivel ebben az esetben nincs eleje és vége a kromoszómának, az elmozdulás valószínűsége megegyezik minden génnél.

Egy másik javaslat szerint nem kell módosítani az inverziót, mivel hasznos lehet az elmozdulások valószínűségének különbsége. E szemlélet szerint a már jó sorrendbe kerül génsorozatok kikerülnek a kromoszóma széléhez közel eső területre, így kis valószínűséggel fognak elszakadni egymástól, míg azok a gének, melyek nem találták meg helyüket, a kromoszóma közepén jobban ki vannak téve az inverzió hatásának.



3.3. ábra. Hibás 1-pontos keresztezés



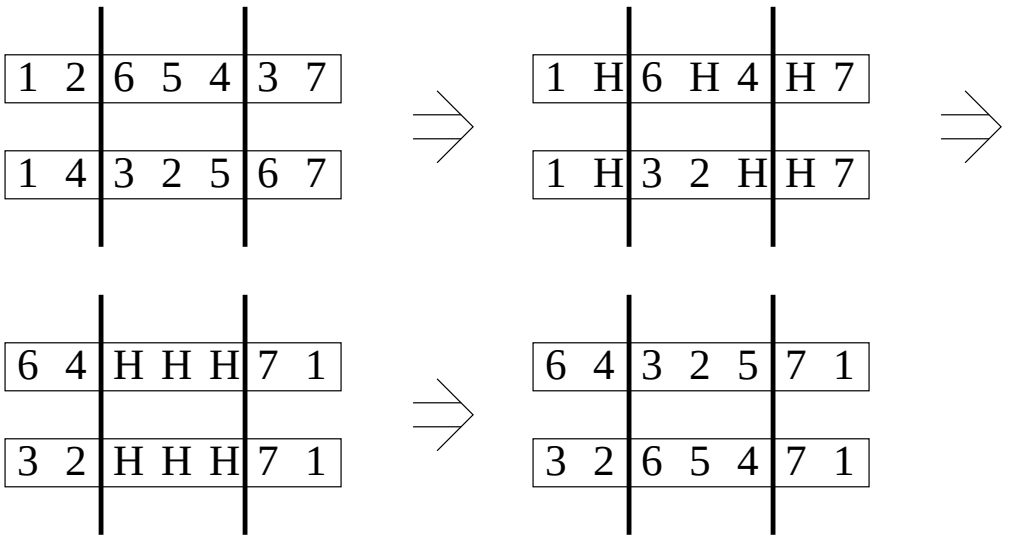
3.4. ábra. PMX keresztezés

3.1.2. Keresztezés

Az eddig megismert keresztezési módszerek (2.8. rész), ha kezeljük a funkciót is, nem működnek helyesen. Figyeljük meg a 3.3. ábrát, ahol két kromoszómát keresztezünk össze az 1-pontos keresztezéssel (2.8.1. rész). A számok a gének (funkciójának) sorszámát jelölik, a gének értékét nem jelenítjük meg. Látható, hogy a szülőknél teljes génkészlet található, vagyis minden gén szerepel a kromoszómában (és pontosan egyszer). A leszármazottakban azonban nem található meg a teljes génkészlet, az egyikből a 3. és 4., a másikból a 2. és 6. gén hiányzik. Az ilyen hiányos kromoszómákkal nem tudnánk dolgozni, hiszen nem határozhatnánk meg a megoldás fenotípusát. A hagyományos keresztezési módszerekkel szemben tehát olyan keresztezési módszerekre van szükségünk, melyek helyesen működnek akkor is, ha kezeljük a funkciót is. Három ilyen módszert vizsgálunk meg.

PMX

A PMX keresztezés (Partially Matched Crossover) során először két véletlen helyet választunk ki a kromoszómán, a két hely közti kromoszómadarabot nevezzük *illeszkedési szakasz*nak. Az itt lévő gének sorszámait párbaállítva számpárokat kapunk. A 3.4. ábrán látható példán ezek a számpárok a (6,3), (5,2) és a (4,5). A leszármazottak generálásához először másolatot készítünk a szülőkről. Ezután a leszármazottakban kicseréljük a számpárok által meghatározott géneket. Tehát először az első leszármazottban a hatost és a hármast (Ekkor kapjuk az 1235467 génsorrendű kromoszómát), majd az



3.5. ábra. OX keresztezés

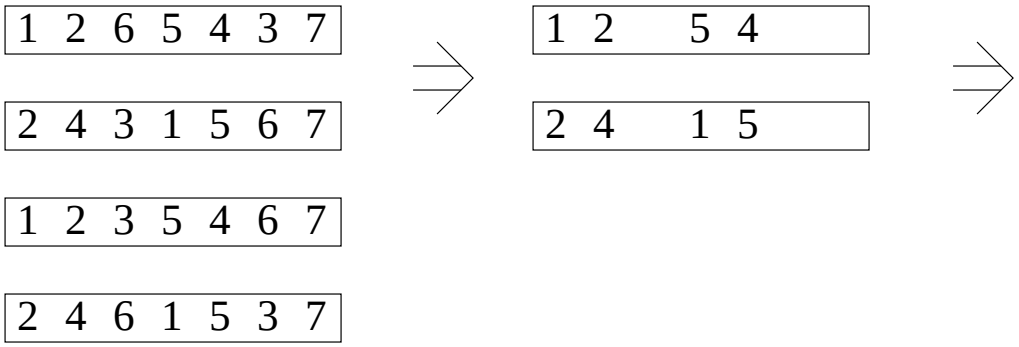
ötöst és a kettest (1532467), végül a négyest és az ötöst (így kapjuk meg az ábrán is látható 1432567 génsorrendű kromoszómát). Ugyanezeket a cseréket elvégezve a másik leszármazotton, megkapjuk az 1564237 génsorrendet.

OX

Az OX keresztezés (Order Crossover) során is először két véletlen helyet (kereszteződési pontot) kell választani a kromoszómán. A 3.5. ábrán látható példában ugyanazokat a helyeket választottuk, mint az előző (3.4.), PMX keresztezést bemutató ábrán. A keresztezés három lépésre bontható:

1. A kromoszómákon kihagyjuk azoknak a géneket (H betűvel jelöljük a helyüket), melyek a másik kromoszóma középső (a két kereszteződési pont közé eső) részén megtalálhatóak. Az ábrán az első kromoszóma középső részén három gén található: 6, 5, 4. Ezért a második kromoszómában elhagyjuk a hatos, ötös és négyes gént. Ugyanilyen okokból elhagyjuk az első kromoszómából a hármas, kettes és ötös gént.

2. Az így kapott lyukakat (H betűket) egymás mellé toljuk. Az összes H betűt a középső kromoszómadarabba juttatjuk. A H betűk száma megegyezik a kromoszómadarab hosszával, így a H betűk pont kitöltik a középső kromoszómadarabot. A többi gén elhelyezése úgy történik, hogy ciklikus sorrendjük ne változzon. A második kereszteződési pont utáni első nem H gén lesz az új kromoszóma második kereszteződési pontja utáni helyen (az ábrán ez a hetes gén). Mivel a ciklikus sorrend nem változik, a többi gént már a helyére tudjuk illeszteni. Mindig



3.6. ábra. CX keresztezés

vesszük a következő nem H gént (a következőt ciklikusan értve), és átmásoljuk a következő szabad helyre.

3. A középső részen lévő H betűk helyére a másik szülő-kromoszóma középső részén található géneket másoljuk. A példában tehát az első kromoszómába a hármas, kettes, ötös, a második kromoszómába a hatos, ötös, négyes gént másoljuk.

CX

A CX keresztezés (Cycle Crossover) különbözik az előző két keresztezéstől, a uniform keresztezésre (2.8.3. rész) emlékeztet. A kromoszóma minden pozíciójára igaz, hogy a leszármazott a gént valamelyik szülőjétől kapja. A két szülő azonos helyen lévő génjei közül az egyiket az egyik utód, a másikat a másik utód kapja meg. Míg az utód kiválasztása a uniform keresztezésnél tetszőleges lehetett, itt a génkészlet teljességének megtartása érdekében más módszert kell választani.

Vizsgáljuk meg a 3.6. ábrán látható példát. A legelső génpárnál (1,2) az 1. szülő génje kerül az 1. utódba, a 2. szülő génje a 2. utódba. Mivel a génkészletnek minden utódban teljesnek kell lennie, a 2. utódban is kell hogy legyen egyes gén. Az egyes gén a szülőkben kétszer fordul elő (mindkét szülőben egyszer). Az egyik darab — az 1. génpár utódokba másolásával — az 1. utódnak jutott, tehát a másik darabot a 2. utódnak kell megkapnia. Tehát a 4. génpár (5,1) esetén az egyes gént a 2. utód, az ötös gént az 1. utód kapja. Hasonló okokból az 5. génpár esetén az ötös gént a 2., a négyes gént az 1. utód kapja. Ezt a lépés még egyszer tudjuk megtenni, a 2. génpárnál a négyes gént a 2., a kettes gént az 1. utód kapja. A kettes génből már kapott egy darabot a 2. utód (az 1. génpárnál), így nincs már lépéskényszer. A többi gén elosztása már egyszerű, az 1. utód kapja a 2. szülő génjeit, a 2. utód kapja az 1. szülő génjeit.

3.1.3. Permutáció optimalizálása

Az előbbi példákban az egyszerűség kedvéért a gének értékeit rögzítettnek tekintettük. Van a feladatoknak egy olyan része, ahol a gének értékei valóban rögzítettek. Ha a gének értékeit rögzítjük, akkor az algoritmus a gének helyes sorrendjét keresi. Ebben az esetben az algoritmus az 1 és n közötti számok optimális *permutációját* keresi. Egy jól ismert probléma ahol permutációt keresünk, az utazóügynök probléma (TSP=Traveling Salesperson Problem).

3.2. Az utazóügynök probléma

Adottak városok, a városok közti távolsággal. (Feltételezzük, hogy bármely két város között van út.) Egy utazó ügynök szeretné az összes várost érinteni úgy, hogy a lehető legkevesebbet kelljen utaznia. Az út végén az ügynöknek vissza kell érnie a kiindulási pontba. (Tehát egy teljes gráf legrövidebb Hamilton körét keressük.) Egy lehetséges megoldáson a városok sorrendjét értjük.

A feladat NP-nehéz, vagyis arra nincs is reális esélyünk, hogy az optimális megoldást megtaláljuk. A cél, hogy minél jobb megoldást találjunk. Létezik olyan hagyományos megoldás, mely, ha a feladat megfelel bizonyos további feltételeknek (pl. háromszög-egyenlőtlenség), akkor garantálja, hogy az optimális megoldástól csak egy előre megadott arányban marad el.

A feladatot már sokszor, sokféleképpen megoldották GA segítségével. A megoldások leginkább ábrázolási módjukban térnek el egymástól. A továbbiakban a legelterjedtebb ábrázolási módokat mutatjuk be. További megoldásokról is olvashatunk a [Mic92] könyvben.

3.2.1. Szomszédsági vektor

Ebben az ábrázolási módban egy vektor segítségével írjuk le az utakat. Ha a j. város az i. pozícióban van, akkor (és csak akkor) vezet az út az i. városból a j. városba. Például a

(2 4 8 3 9 7 1 5 6)

vektor az 1 - 2 - 4 - 3 - 8 - 5 - 9 - 6 - 7 utat reprezentálja. Az első pozíción a 2. város van, a 2. pozícióban a 4. város, a 4. pozícióban a 3. város, a 3. pozícióban a 8. város, a 8. pozícióban az 5. város, ... Előfordulhat, hogy a vektor illegális utat reprezentál. A

(2 4 8 1 9 3 5 7 6)

vektor a 1 - 2 - 4 - 1 illegális úthoz vezet.

Ennél az ábrázolási módnál nem használhatóak a hagyományos keresztezési módszerek. Az itt használható keresztezési módszerek közül a *váltakozó él* keresztezést mutatjuk itt be.

Keresztezzük például a következő kromoszómákat:

p_1 = (2 3 8 7 9 1 4 5 6)
p_2 = (7 5 1 6 9 2 8 4 3)

Véletlenszerűen választunk egy élet p_1 kromoszómából (1 - 2). Ezután a másik kromoszómából választjuk az ehhez az élhez kapcsolódó élet. A második kromoszóma 2. pozíciójában az 5 van, vagyis a 2 - 5 élet választjuk. Ezután az első kromoszómából választjuk az 2 - 5 élhez kapcsolódó élet. Az első kromoszómában az 5. pozícióban a 9. város van, így az 5 - 9 élet választjuk, vagyis a leszármazottban az 5. pozícióban a 9. város lesz. Ezután p_2-ből a 9 - 3 élet, p_1-ből 3 - 8 élet, p_2-ből 8 - 4, p_1-ből a 4 - 7 élet választjuk. Ezután p_2-ből a 7 - 8 élet kellene választani, de a 8. város már szerepel az útban, így ezt az élet nem választhatjuk. Ilyen esetekben a megmaradt éleket megvizsgáljuk, és véletlenszerűen választunk azok közül, melyek eddig nem érintett városba vezetnek. A példában a 7 - 6 élet választottuk (az él egyik szülőnél sem szerepelt!). A p_1-ből választjuk az utolsó élet (1 - 6), így a leszármazott kromoszómája a következő:

(2 5 8 7 9 1 6 4 3).

Az ábrázolási mód legnagyobb előnye, hogy a séma elmélet alkalmazható ennél az ábrázolási módnál.

3.2.2. Sorrendi reprezentáció

Ennél az ábrázolási módnál is egy vektor segítségével írjuk le az utat. A vektor i. eleme egy szám 1 és $n - i + 1$ között. A városok egy előre meghatározott rendezett listáját használjuk referenciapontként. Az egyszerűség kedvéért a referenciapont legyen a következő sorrend:

C = (1 2 3 4 5 6 7 8 9)

Az l = (1 1 2 1 4 1 3 1 1) lista által reprezentált utat a következőképp kaphatjuk meg:

A lista első eleme 1, vagyis C első eleme lesz az út első városa. Egyidejűleg töröljük az első várost C-ből. A részút tehát:

1

A lista következő eleme is 1, vagyis C első eleme lesz az út második városa. Mivel C-ből töröltük az 1. várost, C első eleme a 2. város (melyet rögtön törlünk C-ből), vagyis a részút:

1 - 2

A lista következő eleme 2, vagyis C második városa lesz az út következő városa. A részút:

1 - 2 - 4

A lista következő eleme 1, vagyis C első városa lesz az út következő városa. A részút tehát:

1 - 2 - 4 - 3

Hasonló módon megkaphatjuk az egész utat (1 - 2 - 4 - 3 - 8 - 5 - 9 - 6 - 7).

Az ábrázolás legnagyobb előnye, hogy a hagyományos keresztezési módszerek használhatóak, hiszen minden esetben legális utódok keletkeznek.

p_1 = (1 1 2 1 | 4 1 3 1 1) és
p_2 = (5 1 5 5 | 5 3 3 2 1)

kromoszómák a következő két útvonalat reprezentálják:

1 - 2 - 4 - 3 - 8 - 5 - 9 - 6 - 7
5 - 1 - 7 - 8 - 9 - 4 - 6 - 3 - 2

Egypontos keresztezésnél (2.8.1. rész) a keletkező utódok

(1 1 2 1 | 5 3 3 2 1) és
(5 1 5 5 | 4 1 3 1 1)

a következő két utat reprezentálják:

1 - 2 - 4 - 3 - 9 - 7 - 8 - 6 - 5
5 - 1 - 7 - 8 - 6 - 2 - 9 - 3 - 4.

Látható, hogy az utak eleje változatlan, a végük viszont elég véletlenszerűen változik meg.

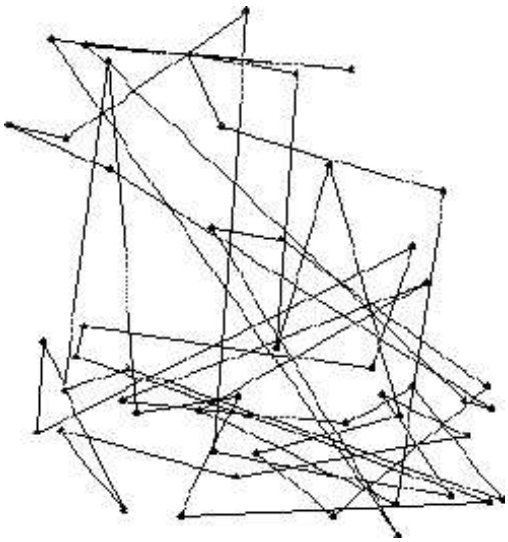
3.2.3. Útvonal-vektor

Az egyik legtermészetesebb reprezentálási mód, ha egy vektorban tároljuk a városokat, és ha a vektor i. eleme akkor és csak akkor a j. város, ha az út i. eleme a j. város. Ebben az esetben a gének értékei állandóak, és csak a lókusz változik, a 3.1.3. részben leírtak szerint. A lókuszkezelésnél leírt mutációt és keresztezéseket használhatjuk ebben az esetben. Többnyire speciális mutációkat és keresztezési módszereket is használnak ebben az esetben.

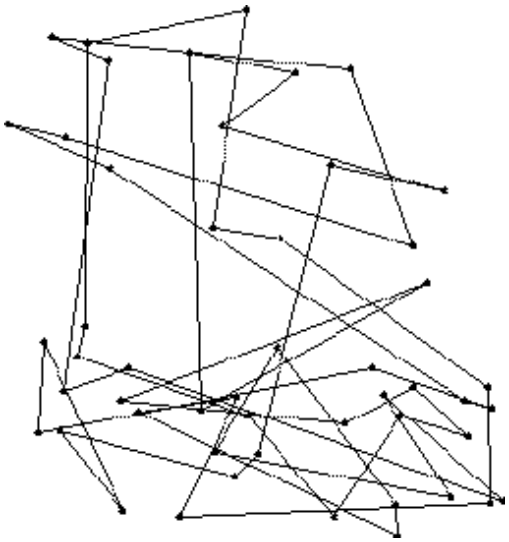
A 3.7., 3.8., 3.9., 3.10. ábrák egy útonal-vektor ábrázolási módot használó GA által a 0., 100., 300. és 1000. generációban talált legjobb útvonalát mutatják. A feladat 50 várost tartalmazott, a populáció mérete 30 volt, inverziót (3.2. rész) és PMX keresztezést (3.1.2. rész) használt a GA. A megoldás csak a GA lehetőségeinek demonstrálására szolgált, nem volt TSP-re optimalizálva a GA.

ER operátor

Az eddig használt operátorok a városokat tekintették elsődlegesnek (pl. a városok pozícióit), és nem az éleket. A feladat jellegéből adódóan érdemes megvizsgálni egy olyan



3.7. ábra. A legrövidebb útvonal (0. lépés)



3.8. ábra. A legrövidebb útvonal (100. lépés)

operátort, mely a gráf éleit fontosabbnak tartja mint a csúcsait. Az ER (*Edge recombination*) operátor az élek kb. 95%-át a szülőkből veszi.

Először az operátor az út éleit keresi meg. A

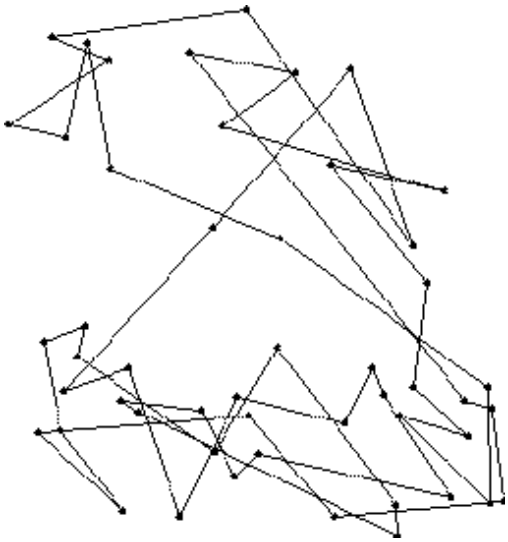
(4 1 2 8 7 6 9 3 5)

útban a következő élek találhatóak: (4 1), (1 2), (2 8), (8 7), (7 6), (6 9), (9 3), (3 5), (5 4). Az élek irányítottságával nem törődik az operátor, a (4 1) és az (1 4) él számára egyforma. Az operátor a leszármazottat kizárólag a szülők éleiből szeretné felépíteni, ezért először megnézi, hogy az egyes csúcsokból mely csúcsokba megy él valamelyik szülőben. Ha a két szülő

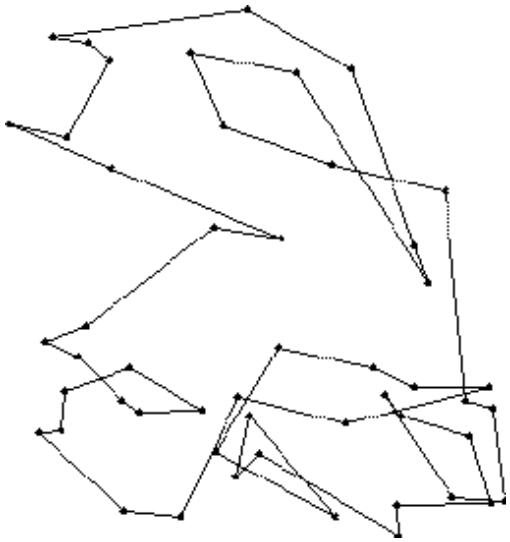
(1 2 3 4 5 6 7 8 9) és
(4 1 2 8 7 6 9 3 5)

akkor az első városból vezet és a 2. városba (mindkét szülőnél), a 9. városba (1. szülő) és a 4. városba (2. szülő). A teljes éllista a következő:

1. város: 9, 2, 4
2. város: 1, 3, 8
3. város: 2, 4, 9, 5
4. város: 3, 5, 1
5. város: 4, 6, 3
6. város: 5, 7, 9
7. város: 6, 8
8. város: 7, 9, 2
9. város: 8, 1, 6, 3



3.9. ábra. A legrövidebb útvonal (300. lépés)



3.10. ábra. A legrövidebb útvonal (1000. lépés)

Először véletlenül választunk egy várost, legyen ez az első város. Ez lesz az út első városa. Az 1. városból vezet út a 9., 2. és 4. városba. A 9. városnak 4, a másik két városnak 3 szomszédja van. Mindig a legkevesebb szomszéddal rendelkező városok közül választunk véletlenül, ez növeli az esélyét, hogy az operátor legális utat ad eredményül. Tegyük fel, hogy az algoritmus a 4. várost választotta. A 4. városból vezet út a 3., 5. és 1. városba, de az 1. város már szerepel az útban. Az 5. városnak kevesebb éle van, tehát ezt választjuk. Hasonló módon választható ki a többi város, és végül megkaphatjuk a

(1 4 5 6 7 8 2 3 9)

utat. Ebben az útban az összes él valamelyik szülőtől származik. A tesztek azt mutatják, hogy csak az esetek 1 – 1.5 százalékában nem lehet ilyen módon összeállítani az utódot.

3.2.4. Evolúciós stratégiában alkalmazott reprezentálás

Bár az evolúciós stratégiák (1.4.2. rész) nem tartozik a GA tárgykörébe, érdemes megnézni, hogy ES esetén miként lehet ábrázolni a TSP egy lehetséges útvonalát. ES esetén a kromoszóma valós számok sorozata. Vegyük például a következő vektort:
v = (2.34, -1.09, 1.91, 0.87, -0.12, 0.99, 2.13, 1.23, 0.55)
A vektorban lévő legalacsonyabb szám a 2. szám (-1.09), ezért az út első városa a 2. város. A következő legalacsonyabb szám a vektor 5. eleme (-0.12), ezért a következő város az 5. Az elvet követve könnyen megállapíthatjuk, hogy a vektor a

2 - 5 - 9 - 4 - 6 - 8 - 3 - 7 - 1

útvonalat reprezentálja.

3.2.5. Mátrixalapú reprezentáció 1.

Bár a kanonikus GA nem engedélyezi, az utóbbi időkben több olyan GA változatot fejlesztettek ki a TSP megoldására, melyek mátrixot használnak kromoszómaként. Ezek közül először a *Fox* és *MacMahon* által kidolgozott megoldást vizsgáljuk meg.

A kromoszóma egy n x n -es bináris mátrix (n a gráf csúcsainak száma). M mátrix m_{ij} eleme akkor és csak akkor 1, ha az i. város az úton a j. város előtt van. Az útvonalat reprezentáló mátrixok a következő tulajdonsággal rendelkeznek:

1. Az 1-esek száma pontosan $\frac{n(n-1)}{2}$
2. $m_{ii} = 0 \; \forall \; 1 \leq i \leq n$
3. Ha $m_{ij} = 1$ és $m_{jk} = 1$, akkor $m_{ik} = 1$

Ha az 1-esek száma kisebb mint $\frac{n(n-1)}{2}$, és a másik két feltétel teljesül, akkor a városok parciálisan rendezettek, és a mátrix kiegészíthető úgy, hogy mindhárom feltétel teljesüljön.

Két új operátort vezettek be, a *metszetet* és a *uniót*, melyek a keresztezésre hasonlítanak.

Metszet

A metszetnél első lépésben egy olyan leszármazottat készítenek, mely csak azokban a pontokban tartalmaz 1-et, ahol mindkét szülő 1-et tartalmazott. Belátható, hogy az ilyen mátrixban az 1-esek száma kisebb mint $\frac{n(n-1)}{2}$, és a másik két feltétel teljesül.

A következő lépésben az egyik szülőből 1-eseket másolnak a leszármazottba, és egy speciális módszerrel kiegészítik a mátrixot, hogy teljesítse az első feltételt is.

Például a következő két utat:

p_1 = (1 2 3 4 5 6 7 8 9)
p_2 = (4 1 2 8 7 6 9 3 5)

a 3.11. ábrán látható két mátrix reprezentálja.

A metszet első fázisa utáni leszármazottat a 3.12. ábrán láthatjuk. Ez a mátrix a csúcsok egy parciális rendezését határozza meg. Az első városnak meg kell előznie a 2., 3., 5., 6., 7., 8. és 9. várost, de nem tudunk semmit az 1. és a 4. város kapcsolatáról.

A mátrixba a következő lépésben 1-eseket kell beszúrni, hogy teljes rendezést kapjunk. Egy lehetséges mátrixot mutat a 3.13. ábra, mely a következő utat reprezentálja:

(1 2 4 8 7 6 3 5 9)

	1	2	3	4	5	6	7	8	9
1	0	1	1	1	1	1	1	1	1
2	0	0	1	1	1	1	1	1	1
3	0	0	0	1	1	1	1	1	1
4	0	0	0	0	1	1	1	1	1
5	0	0	0	0	0	1	1	1	1
6	0	0	0	0	0	0	1	1	1
7	0	0	0	0	0	0	0	1	1
8	0	0	0	0	0	0	0	0	1
9	0	0	0	0	0	0	0	0	0

	1	2	3	4	5	6	7	8	9
1	0	1	1	0	1	1	1	1	1
2	0	0	1	0	1	1	1	1	1
3	0	0	0	0	1	0	0	0	0
4	1	1	1	0	1	1	1	1	1
5	0	0	0	0	0	0	0	0	0
6	0	0	1	0	1	0	0	0	1
7	0	0	1	0	1	1	0	0	1
8	0	0	1	0	1	1	1	0	1
9	0	0	1	0	1	0	0	0	0

3.11. ábra. A két szülő

	1	2	3	4	5	6	7	8	9
1	0	1	1	0	1	1	1	1	1
2	0	0	1	0	1	1	1	1	1
3	0	0	0	0	1	0	0	0	0
4	0	0	0	0	1	1	1	1	1
5	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	1
7	0	0	0	0	0	0	0	0	1
8	0	0	0	0	0	0	0	0	1
9	0	0	0	0	0	0	0	0	0

3.12. ábra. A leszármazott az első fázis után

	1	2	3	4	5	6	7	8	9
1	0	1	1	1	1	1	1	1	1
2	0	0	1	1	1	1	1	1	1
3	0	0	0	0	1	0	0	0	1
4	0	0	1	0	1	1	1	1	1
5	0	0	0	0	0	0	0	0	1
6	0	0	1	0	1	0	0	0	1
7	0	0	1	0	1	1	0	0	1
8	0	0	1	0	1	1	1	0	1
9	0	0	0	0	0	0	0	0	0

3.13. ábra. A leszármazott a végső fázis után

Únió

Az únió operátornál első lépésben két diszjunkt halmazra bontjuk a városokat. Az első halmazhoz tartozó biteket az első, a második halmazhoz tartozó biteket a második mátrixból másoljuk át a leszármazottba. Itt még a leszármazott bitjeinek egy része nem definiált.

A nemdefiniált biteket a metszetnél alkalmazott módszerhez hasonlóan egészíti ki az algoritmus.

3.2.6. Mátrixalapú reprezentáció 2.

Egy másik mátrix alapú megközelítés dolgozott ki *Seniw*. A bináris mátrix m_{ij} eleme akkor és csak akkor 1, ha az út az i. városból a j. városba vezet. Egy legális utat megtestesítő mátrixban minden sorban és oszlopban pontosan egy darab 1-es van. Az ilyen

mátrixok közül azonban nem mindegyik reprezentál legális utat, elképzelhető, hogy a mátrix több kört reprezentál a gráfban. Ezeket a kisebb köröket egy determinisztikus algoritmus kapcsolja össze egy teljes hosszúságú körré. A megengedett körök minimális hossza három volt.

A mutáció során véletlenszerűen sorokat és oszlopokat választunk ki, és a metszéspontokban lévő biteket által alkotott mátrixot vizsgáljuk a továbbiakban. A kis mátrixban tetszőlegesen módosíthatóak a bitek, ha a sorokban, oszlopokban lévő bitek számát változatlanul hagyjuk. Vagyis a

0	1	0
0	0	1

mátrix helyettesíthető a

0	0	1
0	1	0

mátrixxal.

A keresztezés először egy csak 0-kat tartalmazó mátrixból indul ki, majd azokban a pontokban ahol mindkét szülőben 1-es van, 1-est ír a leszármazottba is. Ezután a szülőkből felváltva másolódnak a bitek a leszármazottba, amíg a leszármazott megfelel a feltételeknek. Végül azokat a sorokat és oszlopokat ahol nincs 1-es, véletlenszerűen egészíti ki az algoritmus. Mivel így csak egy leszármazott keletkezne, az algoritmus még egyszer lefut, transzponált szülőmátrixokat használva.

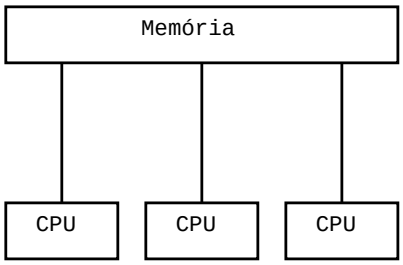
3.3. Párhuzamos Genetikus Algoritmusok

Több oka lehet annak, hogy párhuzamos algoritmusokat használjunk. A lehetséges okok közül néhány:

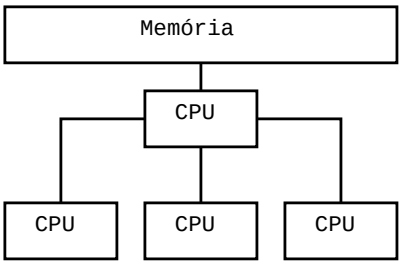
- Az algoritmus eredendően párhuzamos
- Teljesítménynövelés
- Hangzatos
- Az algoritmus párhuzamos változata jobb eredményt ad

A párhuzamos genetikus algoritmusoknál (PGA) eredetileg a teljesítménynövelés volt a cél, de időközben kiderült, hogy a párhuzamos algoritmusok gyakran jobb eredményt adnak, mint a szekvenciális változat. A párhuzamos genetikus algoritmusokról bővebben a [CP99] cikkben olvashatunk.

Az eredeti GA algoritmusról bizonyított, hogy $O(nlogn)$ lépésben konvergál, ahol n a populáció mérete. A konvergencia azt jelenti, hogy a populáció egyedei lényegében azonosak, és további fejlődés már csak mutációval segítségével történhet. A konvergencia nem jelenti azt, hogy az algoritmus megtalálta az optimális megoldást, de a



3.14. ábra. Globális populáció: osztott memória



3.15. ábra. Globális populáció: nincs osztott memória

populáció méretének növelésével ennek valószínűsége nő. Ezzel természetesen a konvergenciához szükséges idő is nő. Vagyis választhatunk a jó de lassú, és a gyors, de rossz eredmény között. PGA-val lehetőségünk van jó eredményt viszonylag gyorsan megkapni.

A PGA algoritmusok könnyen kategorizálhatóak. **Globális populációt** használva az egyedek kiértékelése, és/vagy a genetikai operátorok explicit módon párhuzamosítottak. Az operátorok szemantikája ebben az esetben nem változik, a módszer könnyen implementálható, és nem igényel speciális hardware-t.

A **durva szemcsés** módszernél a populációt több kisebb viszonylag izolált alpopulációra osztjuk. Az alpopulációk között a migráció nevű új operátort használjuk az egyedek mozgatására. Az ilyen PGA-kat többnyire több processzorral rendelkező MIMD gépeken futtatják.

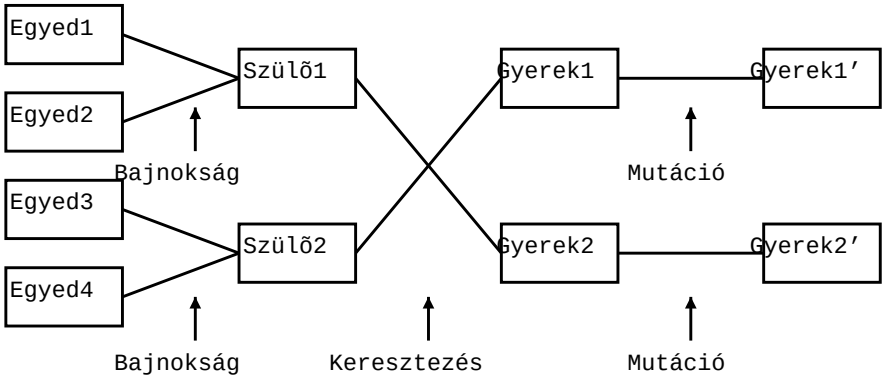
A harmadik megközelítés a **finom szemcsés** PGA ahol a populációt rendkívül kicsi alpopulációra osztjuk. A rendkívül kicsi gyakran 1 egyedből álló populációt jelent. Az esetek nagy részében speciális számítógépeken futnak ezek a PGA-k.

Az utolsó két kategóriánál a kiválasztás és a párosodás csak az alpopuláción belül történik (leszámítva a migrációt), így — a migrációt nem véve figyelembe — azt várjuk, hogy az egyes alpopulációk sokkal hamarabb fognak konvergálni. Ha a migrációt is figyelembe vesszük, akkor ez nem ilyen nyilvánvaló.

3.3.1. Globális párhuzamosítás

Az egyedek kiértékelése könnyen párhuzamosítható, minden processzor bizonyos számú egyedet értékel ki. A kiértékelés során nem kell a processzoroknak kommunikálni. Ha van olyan memóriaterület amit minden processzor lát, akkor a populációt ebben a memóriaterületben tárolhatjuk (3.14. ábra). A generációváltás során szinkronizáció szükséges.

Ha nincs közös memóriaterület, akkor többnyire az egyik processzor felel a populációért (3.15. ábra). Ez a populáció adja oda az egyedeket a többi processzornak, és ez felel az új generáció létrehozásáért. Könnyen lehet, hogy ennek a processzornak a teljesítménye lesz a szűk keresztmetszet.



3.16. ábra. Globális populáció: az operátorok párhuzamosítása

További lépés a párhuzamosításban, ha a genetikus operátorokat is párhuzamosítjuk. A kiválasztás párhuzamosítását nehezíti, hogy néhány változata globális információt igényel (pl. a rulettkerék módszer az átlagos fitneszértéket). A bajnokság kiválasztás viszont könnyen párhuzamosítható. A keresztezés és a mutáció párhuzamosítása nem okoz problémát. A 3.16. ábra egy olyan esetet mutat be, ahol négy egyedből először bajnokság segítségével kiválasztunk két szülőt, majd elvégezzük a keresztezést, végül az utódokon a mutációt. Kérdéses hogy érdemes-e az operátorok párhuzamosításával foglalkozni, mivel az esetek nagy részében az operátorok párhuzamosítás nélkül is kellően gyorsak, és a megnövekedett kommunikáció miatt elképzelhető az algoritmus összességében lassabb lesz.

A globális párhuzamosításra látható példa a 4.5. részben.

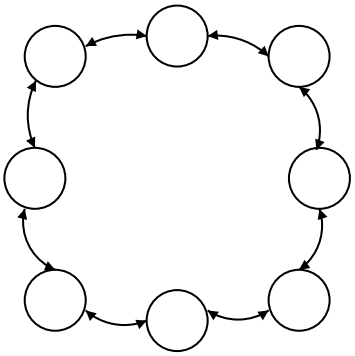
3.3.2. Durva szemcsés párhuzamosítás

A durva szemcsés párhuzamosításnál néhány, viszonylag nagyobb alpopulációnk van, ahol izoláltan fejlődnek az egyedek (Hasonlóan a szigetvilágban élő emberekhez). Időnként bizonyos egyedek átkerülnek az egyik alpopulációból egy másikba, ezt hívjuk *migrációnak*. Ez a 80-as évek közepén létrejött irányzat a legnépszerűbb PGA jelenleg.

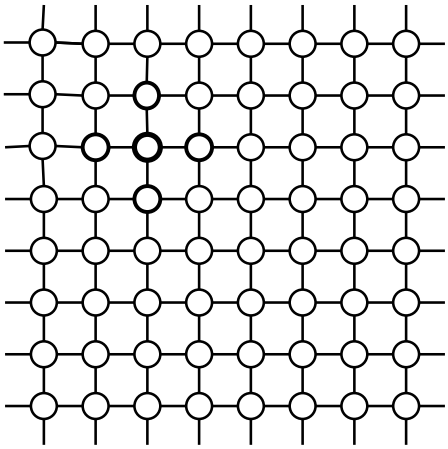
A durva szemcsés párhuzamosítás során az egyes alpopulációk hamarabb konvergálnak mint a nagy globális populáció. A különböző alpopulációk az esetek többségében különböző részmegoldásokat találnak meg, és ezekből a részmegoldásokból a migráció segítségével állhat össze egy még jobb megoldás. A feladattól függ, hogy a megoldást mennyire lehet részmegoldásokká bontani. Ha a feladat jól felbontható, akkor a PGA teljesítménye többnyire felülmúlhatja a szekvenciális GA teljesítményét, ellenkező esetben a szekvenciális GA-nak van előnye.

A migrációt több paraméter segítségével írhatjuk le:

- A **topológia** definiálja, hogy mely alpopulációk között következhet be a migráció.



3.17. ábra. Durva szemcsés párhuzamosítás (gyűrű topológia)



3.18. ábra. Finom szemcsés párhuzamosítás (2-D topológia)

- A **migrációs ráta** határozza meg, hogy a migráció során hány egyed kerül át az eredeti alpopulációból a cél-alpopulációba.
- A **migrációs intervallum** határozza meg, hogy milyen gyakran következik be a migráció.

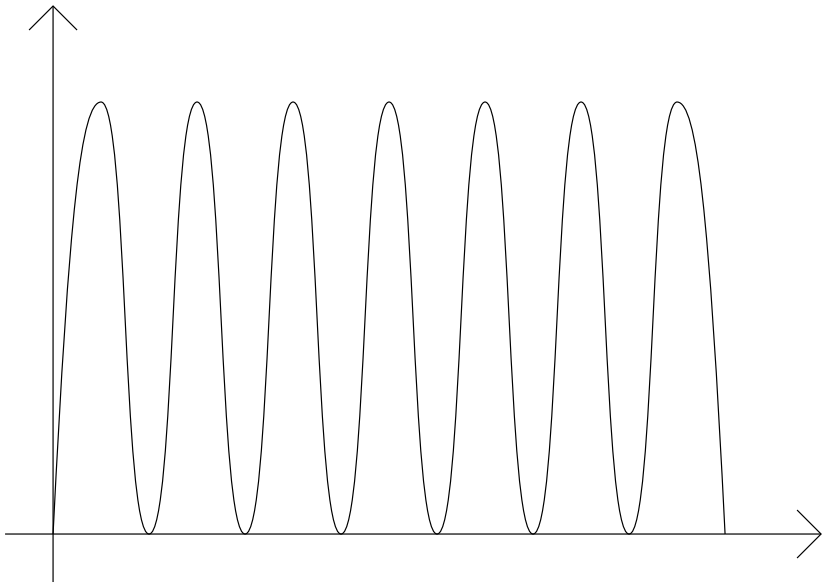
A topológiának fontos szerepe van abban, hogy milyen gyorsan terjednek el a sikeres gének a populációban. Ha az egyes alpopulációk „közel” vannak, akkor gyorsabban, ha „távol”, akkor lassabban terjednek el a sikeres gének. A gyorsaság nem feltétlenül előny, hiszen lehet hogy egy sikeres gén egy alacsonyabb fitneszértékű lokális maximumhoz vezet. A kutatók többnyire a számítógép által használt topológiát használják. Legelterjedtebb a hiperkocka és a gyűrű (3.17. ábra).

A migráció akkor a legsikeresebb, ha akkor hajtjuk végre amikor az egyes alpopulációk már konvergáltak. Ha ezután még várunk, akkor felesleges időt vesztünk el, hiszen a konvergencia után már nem fejlődnek az alpopulációk. Ha nem várjuk meg a konvergenciát, akkor könnyen lehet, hogy még az alpopulációban sem terjedtek el a sikeres gének, így elveszítjük azokat.

Mivel a migráció során a legsikeresebb egyedek kerülnek át egy másik alpopulációba, a migrációs rátát úgy kell megválasztani, hogy csak a legjobbak kerüljenek át (a ráta ne legyen túl nagy), viszont ők mind átkerüljenek (a ráta ne legyen túl kicsi).

3.3.3. Finom szemcsés párhuzamosítás

A finom szemcsés párhuzamosításnál sok, nagyon kis alpopulációnk van. Ennél a PGA-nal legtöbbször speciális számítógépet használnak, így az alpopulációk közti kapcsolatot a gép topológiája határozza meg. Leggyakrabb a 2-D rács topológia (3.18.



3.19. ábra. Több azonos magasságú csúccsal rendelkező függvény

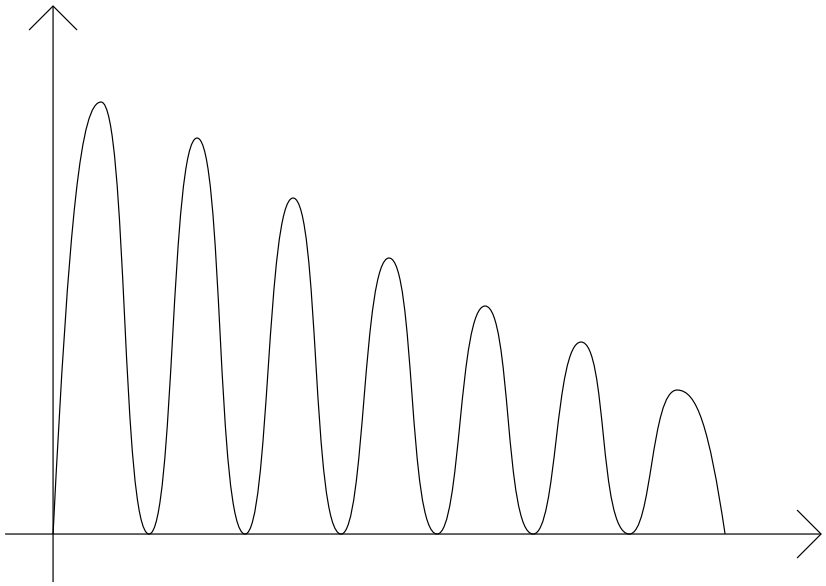
ábra) , de előfordul más is, például gyűrű, tórusz, hiperkocka. Nem lehet meghatározni a „legjobb” topológiát, ez függ a problémától is.

Ha 2-D topológiát használunk, akkor a legegyszerűbb esetben csak a szomszédos egyedek párosodhatnak, így az alpopulációkat az egyedek helye határozza meg. Párosodásnál figyelembe vehetjük a négy legközelebbi szomszédot, vagy a nyolc legközelebbi szomszédot is. Lehetőség van arra is, hogy egy r sugarat határozunk meg, és azok az egyedek párosodhatnak, melyek távolsága ennél kisebb. A tesztek alapján ez gyengébb eredményt adott mint a négy vagy nyolc legközelebbi szomszéd módszere.

3.4. Élettér megosztás

Ha egy olyan függvényt szeretnénk maximalizálni, melynek több azonos magasságú csúcsa van (3.19. ábra), a hagyományos GA programok csak egyetlen csúcsot találnak meg. Bizonyos feladatoknál hasznos, ha nemcsak egy, hanem több csúcsot is megtalál az algoritmus. Vegyünk egy példát a természetből: Ha több közel egyforma minőségű legelő van egy adott területen, az állatok nem zsúfolódnak össze egyetlen legelőn, hanem minden legelőn találhatunk állatokat. Ennek oka az, hogy így nagyobb élettér jut egy állatnak.

Ha egy olyan függvényt maximalizálunk, melynek több különböző magasságú csúcsa van (3.20. ábra), a hagyományos GA programok csak a legnagyobb csúcsot találják meg. Bár a legnagyobb csúcs a legfontosabb, néha fontos lehet a többi csúcs ismerete is. A természetben is megfigyelhető, hogy több különböző minőségű legelő esetén több



3.20. ábra. Több különböző magasságú csúccsal rendelkező függvény

legelőn találhatunk állatokat, a jobb minőségű legelőkön többet, a rosszabb minőségűeken kevesebbet.

A cél az, hogy az algoritmus során az egyedek ne egy csúcs köré gyűljenek össze, hanem minden csúcs körül egy-egy — a csúcs magasságával arányos létszámú — részpopuláció alakuljon ki.

Az egyik legelső módszer a probléma megoldására a *preszelekció*, melynek során a gyengébbik szülőt helyettesítik leszármazottjával, amennyiben az jobb nála. Mivel a leszármazott hasonlít a szülőjére, nem tűnnek el a populáció egyedei között lévő különbségek, tehát az algoritmus nemcsak egy csúcsot talál meg.

A *preszelekció* továbbfejlesztése a *migráció*. Itt a leszármazottat egy véletlenül választott alpopulációval hasonlítják össze, és a hozzá legjobban hasonlító egyed helyére teszik. Az alapelv itt is ugyanaz mint a *preszelekciónál*, egy egyedet mindig egy hozzá hasonló vált fel. A szubpopuláció méretét szabadon választhatjuk meg, a módszert kidolgozó *De Jong* kísérleteiben a méret 2 és 3 volt.

A legismertebb módszer a *Goldberg* és *Richardson* által kidolgozott *fitnesz megosztó módszer*. Az egyedek fitneszét lerontja a hozzájuk közel álló másik egyed. Két egyed távolságát jelöljük d-vel. $d(x_i, x_j)$ az i. és j. egyed távolsága, legegyszerűbb esetben a kromoszómákban lévő eltérő bitek számának és a kromoszóma hosszának hányadosa (Hamming távolság), bár az összehasonlítás nemcsak a genotípusok, hanem a fenotípusok alapján is történhet. Adott egy s megosztó függvény, mely a szomszédság (d) alapján meghatározza a megosztás fokát a populáció minden egyedére. A megosztó függvény nagyon különböző egyedek esetén alacsony értéket vesz fel (közel 0-t), míg hasonló egyedek esetén magasabb értéket (közel 1-et). Két teljesen megegyező egyed

esetén a függvény értéke 1. A legegyszerűbb a háromszög-alakú megosztó függvény, ahol $s = 1 - d$. Természetesen más megosztó függvényt is alkalmaznak.

Az eredeti fitneszértékekből a megosztó függvény segítségével számítja ki az eljárás az új fitneszértékeket, melyek alapján a kiválasztás megtörténik. Az új fitneszérték a következő:

$$f_s(x_i) = \frac{f(x_i)}{\sum_{j=1}^{n} s(d(x_i, x_j))} \tag{3.1}$$

Látható, hogy a nevező annál nagyobb, minél több egyed van közel a vizsgált egyedhez. Ez megakadályozza, hogy egy csúcsnál túl sok egyed összegyűljön, hiszen lerontják egymás fitneszértékét.

Ha a módszerek helyesen működnek, egy új probléma jelentkezik. Ha már kialakultak az alpopulációk, akkor a különböző alpopulációba tartozó egyedek közti párosodás többnyire gyenge utódokat eredményez. A jelenséget megakadályozandó, különböző párosodás-megszorító technikákat is alkalmaznak.

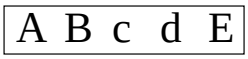
Ha egy több csúccsal rendelkező függvény több csúcsát szeretnénk megtalálni, a lehető legegyszerűbb megoldás, ha többször egymás után elindítjuk a keresést, abban bízva, hogy nem ugyanazt a csúcsot találja meg mindig. Ennek a módszernek a továbbfejlesztése található meg a [BBM93c] cikkben. Az alapgondolat, hogy egy csúcs megtalálása után a megtalált csúcsot megpróbálják „levágni”, így a következő keresés már biztosan egy új csúcsot talál. A „levágást” a fitneszfüggvény módosításával érik el. Az $n.$ lépésben használt fitneszfüggvényt M_n jelöli. Kezdetben $M_0(x) \equiv F(x)$. ($F(x)$ az eredeti fitneszfüggvény). Az $n.$ keresés során (a számozást nullától kezdve) kapott legjobb elemet jelöljük s_n-nel. Az $n+1.$ keresésnél a fitneszfüggvény a következő lesz: $M_{n+1}(x) \equiv M_n(x) \cdot G(x, s_n)$. A G függvény vágja le a megtalált csúcsot. A szerzők kétféle G függvényt használtak, melyeket G_p-vel és G_e-vel jelöltek:

$$G_p(x, s) = \begin{cases} (d_{xs}/r)^{\alpha}, & ha\ d_{xs} < r \\ 1, & \text{különben} \end{cases} \tag{3.2}$$

$$G_e(x, s) = \begin{cases} exp(ln\, m \cdot (r - d_{xs})/r), & ha\ d_{xs} < r \\ 1, & \text{különben} \end{cases} \tag{3.3}$$

Mindkét függvénynél r jelöli az élettér méretét, vagyis azt a távolságot, amekkora távolságon belül megváltozik a fitneszfüggvény. m és α a függvények paraméterei, a tesztek alapján α értékét 2-nek, 4-nek,míg m értékét 0.01-nek célszerű választani. A — szerzők által készített — tesztek alapján a módszer legalább olyan jónak bizonyult mint a *fitnesz megosztó módszer*. A módszer kritikus pontja r értékének meghatározása. Túl nagy érték esetén egy másik csúcs is eltűnhet, túl kicsi érték esetén a csúcs levágásával több magas csúcs keletkezik. (Új csúcsok mindenképpen keletkeznek, de helyesen választott r érték esetén elég alacsonyak, így nem zavarják a további kereséseket.)

Érdemes még a módszerrel kapcsolatban megemlíteni, hogy a módszer nem használja ki, hogy genetikus algoritmust használunk, vagyis másfajta keresési módszerrel



3.21. ábra. Egy diploid kromoszóma

3.22. ábra. A fenotípust meghatározó allélok

együtt is alkalmazható.

3.5. Diploiditás és dominancia

A genetikáról szóló (2.2.) részben olvashattuk, hogy az élő szervezetek közül csak az egyszerűbbek haploidok, a bonyolultabbak diploidok. A diploiditás elsőre pazarlásnak tűnik, hiszen minden génhez két példányt kell eltárolni. A biológusok szerint a diploiditás előnye, hogy az élőlény több olyan allélt tud eltárolni, melyek a múltban hasznosak voltak, és egy esetleges változás (pl. éghajlat) után újra hasznosak lehetnek.

A legismertebb példa egy angliai molylepkefajta sorsa az ipari forradalom idejében. A molylepke kétfajta színben létezik, az egyik világos, a másik sötét színű. Az ipari forradalom előtt a fák kérgén egy világos szín zuzmó élt, így a világos színű lepke könnyen el tudott rejtőzni, ezért ez az allél lényegesen sikeresebb volt mint a sötét színt meghatározó allél. Ekkor a világos színt meghatározó allél volt többségben a sötét színt meghatározó alléllal szemben. Bár a lepkék többsége világos színű volt a sötét szín allélja megtalálható volt a lepkék egy részében.

Az ipari forradalom során a légszennyezés miatt elpusztultak a zuzmók, így a fák eredeti sötét színű kérge vált láthatóvá. Ekkor a sötét színű lepke jobban el tudott rejtőzni, mint a világos színű lepke, így hamarosan a sötét színű lepkék kerültek túlsúlyba.

A történetben mindenképpen fontos, hogy a sötét szín nem egy hirtelen mutáció eredménye, hanem egy már korábban létező lepkeszín, mely a megváltozott környezet hatására előtérbe került. A diploiditás nélkül a sötét szín alléljét hordozó lepkék mind sötét színűek lettek volna, így az allél már az ipari forradalom előtt elveszett volna, ami az ipari forradalom során könnyen a lepkék kipusztulásához vezetett volna.

Felmerül a kérdés, érdemes-e a genetikus algoritmusokban foglalkozni a diploiditással? A diploid kromoszómák kezelése azt jelenti, hogy minden génhez két — nem feltétlenül különböző — allél tartozik. Szemléletesen a diploid kromoszóma két összetartozó haploid kromoszómának tűnik.

Ha a génhez tartozó két allél megegyezik akkor homozigóta, ha különbözik akkor heterozigóta génről beszélünk. Homozigóta génnél a fenotípus meghatározása egyszerű, a heterozigóta esetben azonban a két allél különböző fenotípust határoz meg. A valódi fenotípus meghatározása legtöbbször a dominancia segítségével történik. Dominanciáról akkor beszélünk, ha az egyik allél (a domináns) elnyomja a másikat (a

	0	1_0	1
0	0	0	1
1_0	0	1	1
1	1	1	1

3.23. ábra. A triallélikus séma

recesszívet), vagyis ha mind a két allél jelen van a kromoszómában, akkor a domináns allél határozza meg a fenotípust.

Nézzünk egy egyszerű példát (3.21. ábra). A kromoszómánk álljon 5 génből, mindegyik génnek két allélja legyen. Az első gén alléljait jelöle 'a' és 'A', a második gén alléljait 'b' és 'B', és így tovább egészen 'E'-ig. Ha a példában a nagybetűs allélokat tekintjük dominánsnak, akkor a 3.22. ábrán látható allélok határozzák meg a fenotípust.

A dominancia könnyen megvalósítható, ha fix dominanciát alkalmazunk, vagyis előre elhatározzuk hogy melyik allél domináns a másikkal szemben. A megvalósítás nehezebb ha azt szeretnénk, hogy a dominancia is változhasson. A módszerek közül legsikeresebb a Hollstein-Holland féle triallélikus séma. (A többi módszert a [Gol89] könyvből lehet megismerni.)

Tegyük fel, hogy két allélunk van, melyeket 0-val és 1-el jelölünk. A triallélikus sémában bevezetünk egy új allélt, melyet 1_0-al jelölünk. Ez az allél — a dominanciát nem tekintve — megegyezik az 1-es alléllal. A különbség az 1-es és az 1_0-s allél között, hogy az 1-es domináns a 0-val szemben, míg az 1_0-s recesszív. A 3.23. ábrán látható, hogy milyen allélpároknál milyen lesz a fenotípus.

Érdemes még a diploid kromoszómák kapcsán megemlíteni, hogy a diploid kromoszómák használatának csak akkor van számottevő előnye, ha a fitneszfüggvény nem állandó, hanem az időben változik.

A tesztek azt mutatják, hogy ilyen esetben a haploid struktúránál sikeresebb a fix dominanciával rendelkező diploid struktúrák, azonban ennél is sikeresebbek a triallélikus sémát használó diploid struktúrák.

3.6. A GA paramétereinek meghatározása

Ha egy problémát GA segítségével szeretnénk megoldani, és már döntöttünk az alapvető kérdésekben (milyen GA-t használunk, miként reprezentáljuk a megoldásokat, . . .), még mindig nagyon sok paramétert kell beállítanunk. A mutáció(k) és a keresztezés(ek) valószínűségének kismértékű változása is nagyban befolyásolja a GA eredményességét.

A legegyszerűbb megoldás, ha véletlenszerűen választjuk ki a paraméterek néhány lehetséges értékét, és mindegyikkel lefuttatunk egy tesztet, és végül a legjobb eredményt választjuk ki. A véletlenszerű választás helyett természetesen szisztematikusan

is kereshetjük a paraméterek értékeit, például megvizsgálunk 10 féle mutációs rátát 0.1 és 1 százalék között, és 10 féle keresztezési rátát 30 és 90 százalék között. Észrevehetjük, hogy az optimális paraméterek keresése egy optimalizálási feladat (1.3. rész). Az ilyen feladatokat többféle módszerrel is megoldhatjuk, például hegymászó módszerrel (1.3.2. rész), de akár genetikus algoritmus segítségével is.

3.6.1. Meta-GA

A Meta-GA során keresési módszernek is GA-t választunk. Ebben az esetben tehát a GA optimális paramétereit egy másik GA (meta-GA) segítségével keressük. A meta-GA futása során a fitneszértéket úgy tudjuk kiszámítani, hogy az eredeti GA-t futtatjuk. Mivel a GA egy lassú optimalizálási módszer, így egy fitneszérték számítása is lassú lesz, ami azt eredményezi, hogy a meta-GA még az eredeti GA-nál is sokkal lassabb lesz.

A fitnesz kiszámítása során az eredeti GA-t többnyire csak egy előre meghatározott, viszonylag alacsony, generációszámig futtatjuk, és az így kapott eredményből következtetünk a paraméterek sikerességére. Ez csak egy közelítő fitneszfüggvény (2.9.1. rész), hiszen nem biztos, hogy az a GA a legsikeresebb, mely alacsony generációszámnál is sikeresebb volt.

A meta-GA lassúsága miatt elképzelhető, hogy ha meta-GA helyett a rendelkezésre álló időben az eredeti GA-t futtattuk volna (valamilyen más módon meghatározott paraméterekkel), akkor a nagyobb generációszám miatt jobb eredményt kaptunk volna.

Mivel a meta-GA is GA, ez is rendelkezik optimalizálandó paraméterekkel, melyeket szintén meghatározhatnánk GA-val. Meta-meta-GA-t tudomásom szerint még nem használtak sikerrel.

3.6.2. Adaptív GA

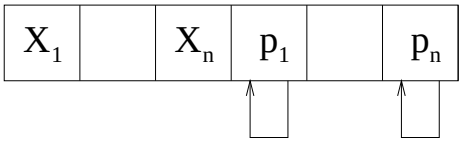
Adaptív GA használatánál a genetikus algoritmus a futás során önmaga próbálja a paraméterek optimális értékét megtalálni. Az adaptív GA egy részénél a (bitvektorrá konvertált) paramétereket az eredeti kromoszómához kapcsoljuk, egy másik csoportjánál nincs szükség a kromoszóma módosítására.

Adaptív GA kromoszómamódosítással

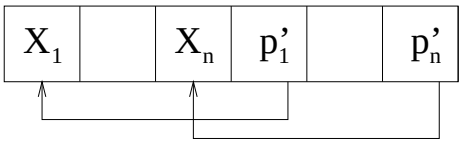
Az adaptív GA-k ezen csoportjánál az eredeti kromoszómához illesztjük azokat a biteket, melyek az adaptív viselkedést befolyásolják. A [Bäc91] cikkben a mutáció valószínűségének meghatározására használta a szerző az adaptív GA-t.

Az eredeti kromoszóma olyan bitvektor, mely n darab valós számot $(x_1, \ldots, x_n)$ reprezentál. Az adaptív GA során külön mutációs rátája van minden elkódolt számnak, az x_i mutációs rátáját p_i jelöli. Egy mutációs rátát $\hat{l}$ bitből álló bitvektor reprezentál, n darab ilyen bitvektort illesztenek a kromoszóma végéhez.

1. Mutációs ráták mutációja



2. Eredeti változók mutációja



3.24. ábra. Az adaptáció működésének alapelve

A mutáció során először a mutációs ráták esnek át mutáción, minden mutáció valószínűségét a a mutációs ráta saját maga határozza meg. A módosult mutációs ráták határozzák meg annak a valószínűségét, hogy az eredeti számok mutáción esnek át. A működés vázlatát mutatja a 3.24. ábra.

A cikk ismerteti a módszer egy egyszerűbb változatát is. Az egyszerűbb esetben ugyanaz a mutációs ráta érvényes egy kromoszóma összes változójára, ekkor csak egy mutációs rátát illesztenek a kromoszómához.

A cikk tesztjei azt mutatják, hogy ha az optimalizálandó függvény egy globális optimummal rendelkezik akkor az egyszerűbb, ha több optimummal rendelkezik akkor a bonyolultabb változatot érdemes használni.

Adaptív GA kromoszómamódosítás nélkül

Az itt leírt módszerről például a [ESKB97] cikkben olvashatunk bővebben. A cikkben több keresztezési módszer közül kellett megtalálni a feladatoknak legmegfelelőbb módszert.

A populációt M alpopulációra osztják, és minden alpopuláció különböző keresztezést használ. A kezdetben egyforma méretű alpopulációk mérete az algoritmus során folyamatosan változik, a sikeresebb alpopulációk mérete nagyobb lesz mint a sikertelen populációké. A populációk méretét két mechanizmus változtatja: a *migráció*, mely növeli a sikeres populációk méretét, és a ritkábban használt *újrafelosztás*, mely növeli a sikertelen alpopulációk méretét, egy új esélyt adva nekik.

Migráció. A migrációt többféleképpen is meg lehet valósítani. A cikk írói minden alpopulációból véletlen számú egyedet helyeztek át a migrációs halmazba (MP = Migration pool). A sikertelenebb alpopulációk több egyedet helyeznek el a migrációs halmazba, a sikeresek kevesebbet. A következő lépésben véletlenszerűen osztjuk el az

egyedeket az alpopulációk között, úgy, hogy mindegyik körülbelül azonos számú egyedet kapjon.

Jelölje F_i az $i.$ alpopuláció átlagos fitneszét, F_{min} és F_{max} a legrosszabb és legjobb alpopuláció átlagos fitneszét. Legyen $\Delta F_i = F_{max} - F_i$. Ha $F_{max} = F_{min}$, akkor nincs migráció. Ellenkező esetben ΔF_i értékeit a következő módon normalizáljuk:

$$\Delta F_i^* = \frac{\Delta F_i}{F_{max} - F_{min}} \tag{3.4}$$

Az $i.$ alpopuláció $c\Delta F_i^*|P_i|$ egyedet küld a migrációs halmazba, ahol $0 \leq c \leq 1$ a mechanizmus paramétere (a tesztek során 0.75 volt). A migrációs halmaz (MP) egyedeit egyenlő arányban osztják el egymás között az alpopulációk, minden alpopuláció $\lfloor \frac{|MP|}{M} \rfloor$ egyedet kap. Ha a kerekítések miatt maradnak még egyedek a migrációs halmazban, akkor a legjobb alpopulációk még kapnak egy-egy egyedet.

Újrafelosztás. Az újrafelosztás növeli a sikertelen alpopulációk méretét, új esélyt adva így a sikertelen keresztezéseknek. Ez megóvja azokat a keresztezéseket az eltűnéstől, melyek eleinte sikertelenek és csak később válnak sikeressé.

Az újrafelosztás során minden alpopuláció egyedeinek egy adott százalékát helyezi az újrafelosztásos halmazba (RP=redivision pool), melyet egyenlően osztanak el az alpopulációk között. Mivel a kevesebb egyedet tartalmazó alpopulációk kevesebb egyedet küldenek a halmazba, ez a mechanizmus az alpopulációk méretét egymáshoz közelíti.

Az egyes egyedek $d|P_i|$ egyedet küldenek (d a mechanizmus paramétere, mely a tesztek során 0.25 volt) , és $\lfloor \frac{|RP|}{M} \rfloor$ egyedet kapnak az újrafelosztásos halmazból. Ha a kerekítések miatt marad még egyed a halmazban, akkor a legjobb alpopulációk kapnak egy-egy egyedet.

A cikk tesztjei azt mutatják, hogy az adaptív GA eredménye lényegesen jobb, mintha egy nem-optimális keresztezést használtunk volna, és csak minimális mértékben rosszabb mintha csak az optimális keresztezést használtuk volna. Vagyis ha több keresztezés közül kell választanunk, és nem tudjuk, hogy melyik az optimális, akkor az adaptív GA használatával a döntést a GA-ra bízhatjuk. Mivel az optimális keresztezés előre nem ismert, a hosszas tesztek helyett, melyek segítségével megtalálnánk az optimális keresztezést, érdemesebb adaptív GA-t használni.

Meglepő módon az algoritmus futása során az egyes alpopulációk mérete között nem alakult ki lényeges különbség, vagyis a sikeres alpopulációk mérete nem volt lényegesen magasabb mint a sikertelen alpopulációk mérete.

Összehasonlítás

Kromoszómamódosítással az adaptív viselkedést egyedenként, míg kromoszómamódosítás nélkül alpopulációnként befolyásolhatjuk. Ha a befolyásolni kívánt jellemző

az egyedre lokális (pl. a mutáció), akkor az egyedenkénti viselkedés hasznosabb lehet, ha globális (pl. problémafüggő paraméterek), akkor az alpopulációt használó adaptáció tűnik hasznosabbnak.

Megvalósítás szempontjából kényelmesebb és elegánsabb, ha nem módosítjuk a kromoszómát. Ha durva szemcsés párhuzamos GA-t (3.3.2. rész) használunk, akkor az ott alkalmazott alpopulációkat az adaptív GA számára is felhasználhatjuk.

4. fejezet

Alkalmazások

Ebben a fejezetben olyan alkalmazásokat ismertetünk, melyek a korábban leírt elméletekre épülnek. Az ismertetett problémák túlnyomó részénél nem cél az áttekintő bemutatás, a lehetséges megoldások közül csak egyet mutatunk be. Az alkalmazások kiválasztásánál a fő cél a változatosság volt, és az, hogy a korábban leírt elméletek közül minél több gyakorlati alkalmazása bemutatható legyen.

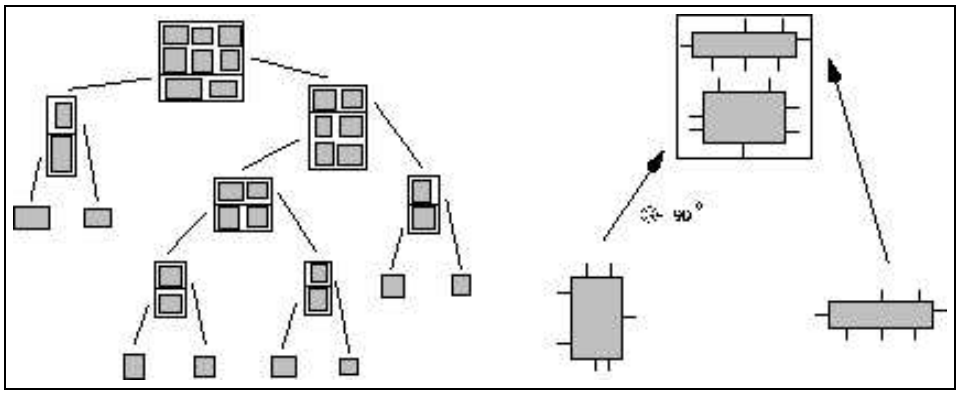
4.1. VLSI tervezés

A VLSI tervezés egyrészt önmagában is egy érdekes feladat, másrészt egy példa a korábban (3.6.2. rész) ismertetett adaptációra.

4.1.1. A feladat

A feladat VLSI (Very Large Scale Integrated) mikrochip elrendezésének tervezése. A mikrochip komponensekből, és a komponenseket összekötő vezetékekből áll. A komponensek nem fedhetik át egymást, a vezetékek között egy előre megadott minimális távolságnak kell lennie, és néhány vezeték maximális hossza előre meg van határozva. A cél az hogy egy előre megadott szerkezetű mikrochipet minél kisebb alapterületen megvalósítsunk.

A feladat komplexitása miatt többnyire két lépésben történik az optimalizáció. Az első lépésben a komponensek, a második lépésben a vezetékek elhelyezése történik. Ha a túl kis hely miatt nem lehet a vezetékeket elhelyezni, akkor az algoritmus a komponenseknek új helyet keres, ha túl sok hely marad akkor az optimalizáció után összébb lehet nyomni a komponenseket.



4.1. ábra. A genotípus

4.1.2. Megoldás GA-val

Az itt leírt megoldást, mely összevonja az optimalizáció két lépését, a [SV96] cikk írja le. A fejezet ábrái is megtalálhatóak a cikkben.

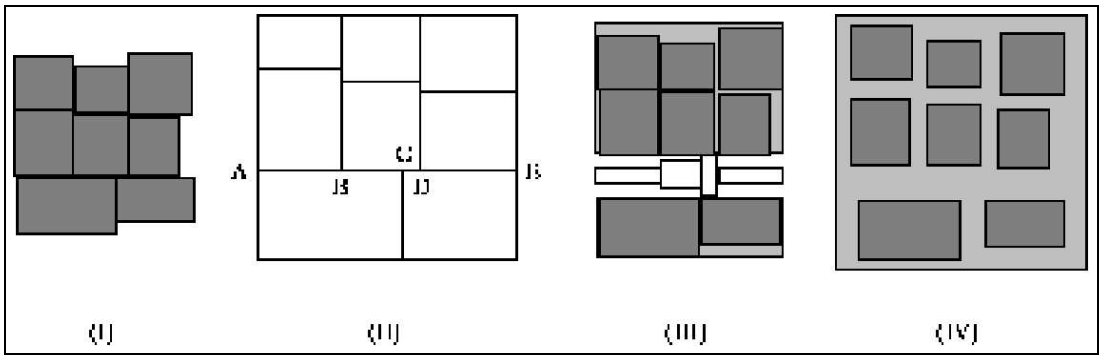
Reprezentáció

A genotípus egy bináris fa, melyet nem kódolunk el bitvektorrá. A bináris fa leveleiben a komponensek találhatóak. A belső csúcs két blokkot (pl. komponenst) illeszt össze egy ún. meta-blokká. A meta-blokkban a komponensek iránya meghatározott. A bináris fa tehát meghatározza a komponensek egymáshoz viszonyított helyzetét. A genotípusra mutat egy példát a 4.1. ábra. Bár a komponensek egymáshoz viszonyított helyzetét meghatározza, a genotípus nem adja meg pontosan a komponensek helyét, és egyáltalán nem foglalkozik a vezetékekkel.

A bináris fából egy elhelyezési gráf és egy útvonal gráf készíthető. Az elhelyezési gráf határozza meg a komponensek egymáshoz viszonyított helyzetét (pl. a harmadik komponens az első komponenstől jobbra, 90 fokkal elforgatva található), az útvonal gráf élei a csatornáknak felelnek meg, ahol a vezetékek futnak.

Az elhelyezési gráf alapján minden vezetékhez meghatározható a legrövidebb út (legrövidebb út keresése gráfban: pl. Dijkstra), az utak alapján ismert, hogy az egyes csatornákban hány vezetéknek kell hely. Így a csatornák minimális szélessége is ismert. Ebből az információból már meghatározható a komponensek helye. A 4.2. ábrán látható egy példa a komponensek helyének meghatározására. Az első ábra mutatja az elhelyezési gráfot, a második az útvonal gráfot. A harmadik ábrán látható az egyik csatorna szélességének meghatározása (a csatornában futó vezetékek számát felhasználva), a negyedik ábrán a komponensek végső elhelyezkedése.

Az algoritmus tehát csak a komponensek elhelyezését próbálja optimalizálni, és egy determinisztikus — ám nem feltétlenül optimális — algoritmus alapján határozza meg a vezetékek helyét.



4.2. ábra. Az elhelyezési (I), az útvonal (II) gráf, a vezetékeknek szükséges csatornák vastagsága (III), és a komponensek ez alapján meghatározott helye (IV)

Kezdeti populáció

A kezdeti populációt nem véletlenszerűen töltik fel, hanem egy VLSI-tervezésben ismert heurisztika alapján, mely az összefüggő komponenseket egymáshoz közel helyezi el.

Genetikus operátorok

A keresztezés során a két szülő egy-egy diszjunkt részfáját kapcsolják össze. Mivel így nem biztosított, hogy minden komponens szerepel a leszármazottakban, a hiányzó komponenseket a kezdeti populáció létrehozásakor használt heurisztika segítségével helyezik el.

A használt mutációk közül az egyik egy blokk (vagy meta-blokk) irányát módosítja (elforgatja), egy másik a fa két részfáját cseréli meg. A harmadik mutáció a fa egy részfáját a fa egy másik pontjába helyezi át.

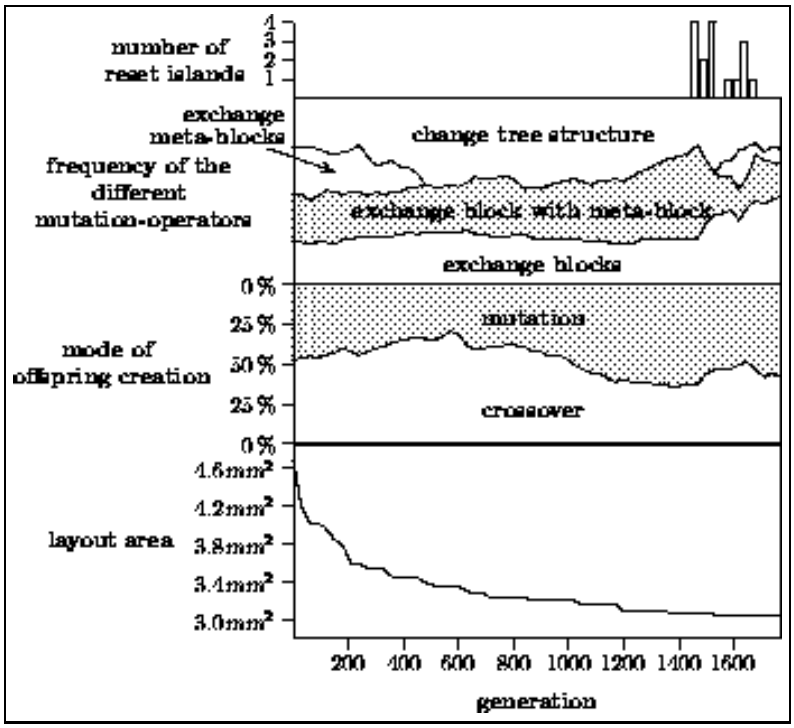
Mint a tesztek során kiderült, ebben az alkalmazásban a mutációk fontosabbak mint a keresztezés.

Adaptáció

A GA sok paramétere közül a cikk szerzői a következőket próbálták optimalizálni: mutációs ráták, a keresztezési ráta és vágási szelekció (melyről nem ír a cikk) küszöbértéke. Az algoritmus később a mutációs rátákat rögzíti, és csak a többi paramétert optimalizálja.

A populáció fix méretű alpopulációkra van osztva, és mindegyik alpopuláció különböző stratégiát követ. A stratégiát az optimalizálandó paraméterek értékei határozzák meg.

Előre meghatározott időpontonként a stratégiákat rangsorolják, és minden stratégia a rangsorban eggyel jobb stratégiához alkalmazkodik. A legjobb stratégia meghatározó



4.3. ábra. Az algoritmus egy futtatásának eredménye

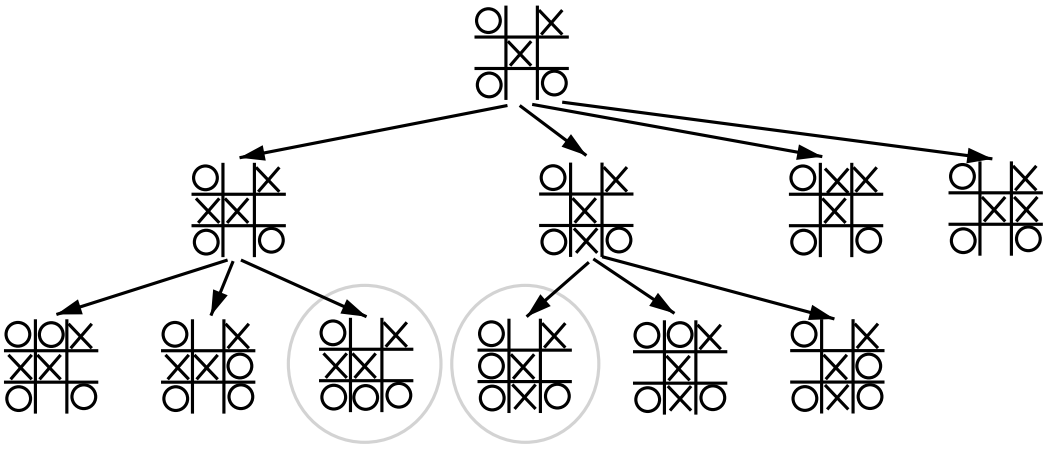
paramétereit megerősítik. Ha például a többi stratégiához képest alacsony a keresztezési ráta, akkor ezt tovább csökkentik.

Ha az optimalizálás elakad egy alpopulációban, vagyis a legutolsó adaptáció óta nincs előrelépés, akkor a stratégiát törlik, és az alpopuláció az eredeti stratégiák közül választ egyet.

A stratégiák rangsorolása nem a legjobb egyed fitnesze szerint történik, hanem az átlagos fitnesz legutolsó adaptáció óta történt megváltozásának mértéke szerint.

A szerzők által készített tesztek szerint legrosszabb eredményt akkor kapjuk, ha minden szigeten ugyanazt a stratégiát használjuk. Ha szigetenként különböző stratégiát használunk, lényegesen jobb eredményt kapunk. Az eredményt tovább javíthatjuk, ha használjuk az adaptációt is.

A 4.3. ábra mutatja az algoritmus egy futtatásának eredményét. Látható rajta, hogy stratégiatörlés csak az algoritmus futásának végén fordult elő. Az ábra mutatja a keresztezés/mutáció előfordulásának valószínűségét, és az egyes mutációk valószínűségét is. Végül az ábrán látható a chip méretének csökkenése is.



4.4. ábra. Játékgráf

4.2. Kétszemélyes játékok

A legtöbb, emberek által játszott táblás játékoknak több közös jellemzője van. A játékot két játékos játsza, akik felváltva lépnek. Mindkét játékos birtokában van a játékkal kapcsolatos összes információnak, a játékban nincs szerepe a véletlennek. Az aktuális állásban véges számú lépés közül választhat a játékos, és a lépés után egy új — a szabályok ismeretében előre kiszámítható — állásba jut.

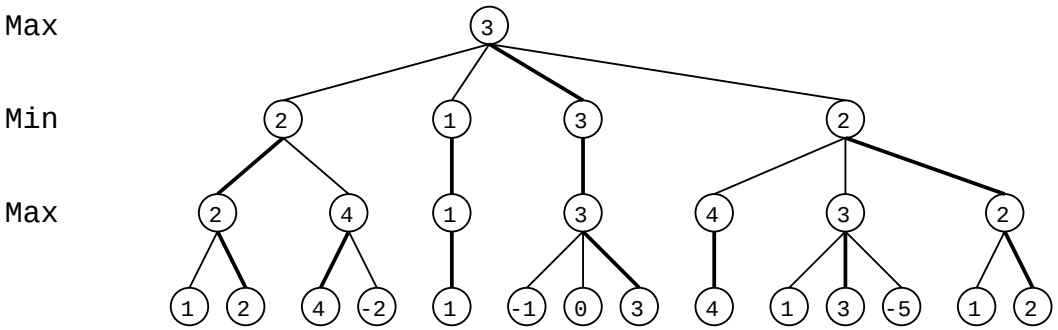
A játék egy kezdőállásból indul, és az állások egy része végállás, vagyis olyan állás, ahol vége van a játszmának. A végállásokban eldönthető a játék kimenetele, ami győzelem, vereség és döntetlen lehet. A véges lépésből álló játék kimenetele alapján a játékosok jutalmat kapnak, melynek összege 0. Az ilyen játékokat kétszemélyes, teljes információjú, nulla összegű játékoknak hívjuk. Az ilyen játékokról bővebben a [Fea99] könyvben olvashatunk.

Ha számítógép játszik kétszemélyes játékot, akkor egy adott állásban a lehetséges lépésekből kell a legjobb a lépést megtalálnia. A lehetséges lépések a játékszabályok alapján könnyen meghatározhatóak. Legjobb lépésnek azt a lépést tekintjük, ahol a várható jutalom a legnagyobb. Ha az egyes állásokról meg tudjuk mondani, hogy mennyire jó állások (mi a várható jutalom az állásokból indulva), akkor a legjobb lépés kiválasztása egyszerű maximumkereséssel kiválasztható.

4.2.1. Minimax algoritmus

Ha az állásokat egy gráf csúcsainak, a lépéseket a gráf éleinek tekintjük, akkor egy adott állásból indulva fel lehet rajzolni azt a gráfot (játékgráf), ami az állásból elérhető állásokat tartalmazza. A 4.4. ábra a 3x3-as amőba játékgráfjának egy részletét mutatja. A bekarikázott csúcsok végállásokat reprezentálnak.

Mivel a játékban korlátos a játszmák hossza, a gráf véges. A végállásokban könnyen



4.5. ábra. Minimax

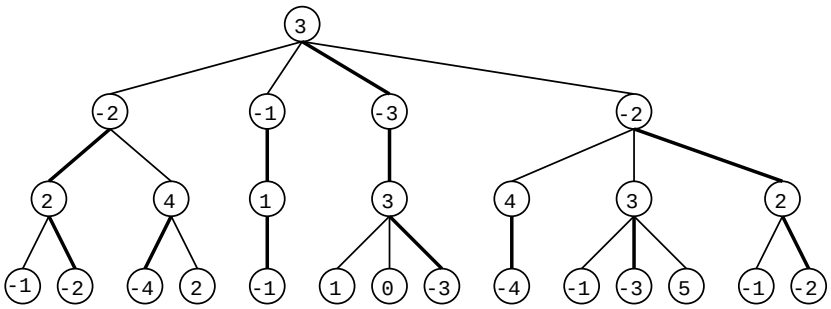
eldönthetjük, hogy ki nyer, így elméletileg az optimális lépés meghatározása könnyű, azonban a kombinatorikus robbanás miatt az optimális lépés megtalálásához szükséges idő túl nagy (pl. a világegyetem élettartamának többszöröse).

A minimax algoritmus során a gráfot fának tekintjük (a csúcsok többszörözésével ez könnyen elérhető), és csak egy előre meghatározott mélységig vizsgáljuk a fát. Az egyszerűség kedvéért a két játékost MAX és MIN játékosnak nevezzük.

A gráf leveleiben lévő (nem feltétlen vég) állásokról egy statikus kiértékelő függvény (melyet f-el jelölünk) segítségével határozzuk meg az állás jóságát. A függvény értéke annál nagyobb, minél inkább a MAX játékos áll jobban. A többi csúcsnál azt az egyszerű (ám nem mindig igaz) elvet követjük, hogy feltételezhető módon mindkét játékos mindig a számára legjobb lépést követi. Vagyis ha az ellenfél lép, akkor feltételezhetően azt a lépést választja, mely a legkisebb jósággal rendelkező állásba vezet (számunkra a legrosszabb állás), míg ha mi lépünk akkor feltételezhetően a legjobb állásba vezető lépést választjuk. Tehát ha az n csúcsban lévő értéket $v(n)$, n csúcs leszármazottait $n_1, n_2, \ldots, n_k$ jelöli, akkor a következő módon számíthatjuk ki $v(n)$ értékét:

$$v(n) = \begin{cases} f(n) & \text{ha n levélcsúcs} \\ max(v(n_1), \ldots, v(n_k)) & \text{ha n páros (MAX) szinten van} \\ min(v(n_1), \ldots, v(n_k)) & \text{ha n páratlan (MIN) szinten van} \end{cases} \quad (4.1)$$

A 4.5. ábrán egy játékgráf látható. A gráf szintjeit felváltva MAX (páros) és MIN (páratlan) szintnek nevezzük, a gyökér a MAX szinten található. A levelekben lévő számok a statikus kiértékelő függvény által adott értékek, melyek az állás jóságát fejezik ki, mindig ugyanazon játékos szempontjából. A gráf többi csúcsában megvizsgáljuk a csúcs leszármazottait, és a leszármazottakban lévő számok közül maximum (ha a csúcs MAX szinten van), vagy minimum (ha a csúcs MIN szinten van) kereséssel kapjuk meg a csúcs értékét. A módszert folytatva megkapjuk az egyre feljebbi szinten lévő csúcsok értékeit, végül a gyökérhez tartozó értéket, mely 3. Egy csúcsban az optimális lépés mindig egy olyan gyerekcsúcsba vezető lépés, melynek értéke mege-



4.6. ábra. Negamax

gyezik a csúcs értékével. Az ábrán a 3. lépés az optimális lépés. Az egyes csúcsokban az optimális lépés vastag vonallal jelölt.

Negamax algoritmus

A gyakorlatban a minimax algoritmus *negamax* változatát használják. A váltakozó minimum és maximum keresés helyett az algoritmus az egy szinttel lejjebb lévő állások pontszámait negálja, így elég maximumkeresést használnia az algoritmusnak. Az algoritmus pontos leírása megtalálható a [Fea99] könyvben.

A 4.1. képlet helyett a következő képletet használhatjuk:

$$v(n) = \begin{cases} f(n) & \text{ha n ps (MAX) szinten lévő levélcsúcs} \\ -f(n) & \text{ha n ptln (MIN) szinten lévő levélcsúcs} \\ max(-v(n_1), \ldots, -v(n_k)) & \text{ha n nem levélcsúcs} \end{cases} \tag{4.2}$$

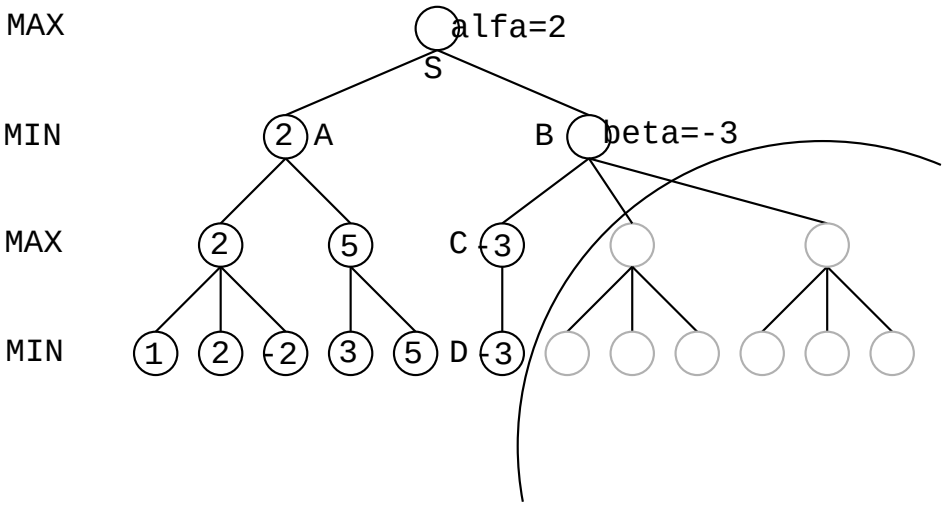
Alfa-béta vágás

A minimax algoritmus során a csúcsok egy részének kiértékelése felesleges. Az alfa-béta vágás segítségével a felesleges kiértékeléseket tudjuk elkerülni.

Ha egy MAX szinten lévő csúcsnak az első leszármazottját kiértékeltük, akkor a csúcs értékére egy alsó becslést kapunk (hiszen a kapott értéket felhasználjuk a maximumkeresés során). Ezt a MAX csúcshoz rendelt, kiértékelés során folyamatosan növekedő alsó becslést α-nak nevezzük.

Hasonló módon definiálhatunk minden egyes MIN csúcshoz egy β értéket, mely a csúcs értékére egy folyamatosan csökkenő felső becslés.

Ha egy MIN csúcshoz tartozó β érték kisebb, mint a csúcs valamely MAX őséhez tartozó α érték, akkor a MIN csúcs további leszármazottait nem kell kiértékelni, levághatjuk őket. Hasonló módon, ha egy MAX csúcshoz tartozó α érték nagyobb mint a csúcs egy MIN őséhez tartozó β érték, akkor a MAX csúcs további leszármazottait



4.7. ábra. Alfa-béta

nem kell kiértékelni, levághatjuk őket. Ezeket a vágásokat hívjuk alfa és béta vágásnak. A vágások segítségével az algoritmus ugyanazt az eredményt adja, de lényegesen gyorsabban.

A 4.7. ábrán egy egyszerű példa látható az alfa-béta vágásra. Az „A” csúcs leszármazottainak kiértékelése után megkapjuk az „A” csúcs értékét, 2-t. Ez egyben alsó becslés „S” csúcs értékére, vagyis ez az „S” csúcshoz tartozó α érték.

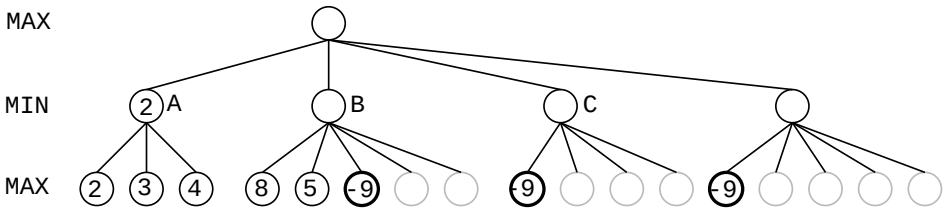
„D” csúcs kiértékelése után megkapjuk „C” csúcs értékét is, -3-at. Ez az érték egyben felső becslés „B” csúcs értékére, vagyis a „B” csúcs β értéke. „S” csúcs értéke „A” és „B” csúcs értékének maximuma ($v(S) = max(v(A), v(B))$) , „A” csúcs értéke 2 ($v(A) = 2$), „B” csúcs értéke legfeljebb -3 ($v(B) \leq -3$). Az előzőekből következik, hogy „S” csúcs értéke 2, függetlenül a még ki nem értékelt (az ábrán szürkével rajzolt) csúcsoktól. „B” csúcs többi leszármazottját tehát nem kell kiértékelnünk, az ágakat levághatjuk, és „B” csúcsnak értékül adhatjuk a -3-at.

Könnyen észrevehető, hogy a vágások száma akkor a legnagyobb, ha a jó lépéseket értékeljük ki először. Természetesen nem tudhatjuk melyek a jó lépések, hiszen ha tudnánk, az egész minimax algoritmusra nem lenne szükség.

A lépések jóságát meg tudjuk becsülni, ha minden leszármazottra meghívjuk a statikus kiértékelő függvényt, mely közelítőleg megmondja mennyire jó a lépés. A plusz kiértékelések költségét remélhetően kompenzálja a több vágási lehetőség.

A lépések sorrendjét heurisztika segítségével is meghatározhatjuk. Sakknál ismert az a módszer, ami a sáncolást (ami többnyire jó lépés) értékeli ki legelőször.

Nagyon gyakran alkalmazzák a *cáfoló lépés* (gyilkos lépés) módszerét. Képzeljük el (4.8. ábra), hogy a sakkpartiban az ellenfél megtámadja védtelen vezérünket, ami csak egy helyre tud menekülni. A lehetséges válaszlépéseinkből először kiértékeljük ezt a lépést, és azt kapjuk, hogy az állás körül-belül döntetlen, a csúcs (az ábrán A)



4.8. ábra. Cáfoló lépés

értéke 2.

A következő lépés (melyet az ábrán B jelöl) nem védi meg a vezért, így az ellenfél harmadik lehetséges lépésének (vezérünk ütése) kiértékelési után azt kapjuk, hogy vesztésre állunk (a csúcs értéke -9). Az ellenfél többi lépését nem is kell kiértékelnünk, hiszen B csúcs értéke legfeljebb -9 lehet, így a gyökérben lévő csúcs kiértékelésekor ez nem módosíthatja a maximumot (ami az A csúcs miatt legalább 2 lesz). A vezér ütését mint az ellenfél legsikeresebb lépését cáfoló (gyilkos) lépésnek nevezzük.

Az ellenfél következő lépéseinek kiértékeléseker (C csúcs), el kell dönteni, milyen sorrendben értékeljük ki a lehetséges lépéseket. A cáfoló lépés módszere azt ajánlja, vizsgáljuk meg először a cáfoló lépést (feltéve, hogy szerepel a lehetséges lépések közt)

4.2.2. Kétszemélyes játékok és a GA

Ha a fenti algoritmusokat használjuk, akkor a számítógépes játékos képességeit a statikus kiértékelőfüggvény határozza meg. Minél pontosabban (és gyorsabban) határozza meg a függvény, hogy egy állás mennyire jó, annál jobban játszik a program.

A statikus kiértékelőfüggvény meghatározásában segíthetnek az evolúciós algoritmusok is. Két fő módszert használnak a gyakorlatban. Az első esetben meghatározzák a statikus kiértékelőfüggvény szerkezetét, és bizonyos paraméterek optimalizálását bízzák GA-ra. Amennyiben minden állásban kiszámolható az állásra jellemző A, B, C és D értékek, akkor a statikus kiértékelőfüggvény alakja lehet például $\alpha A+\beta B+\gamma C+\delta D$. A kromoszóma $\alpha, \beta, \gamma, \delta$ elkódolt értékeit tartalmazza.

A másik módszerben genetikus programozást (1.4.5. rész) használnak. Genetikus programozás során a kiértékelőfüggvény szerkezete sincs meghatározva, a függvényt egy kiértékelőfával reprezentáljuk. A lehetséges operátorokat (pl. +, -, *, /) kell felsorolni, és a GP az ezekből összeállítható összes függvény közül tud választani.

Vagyis evolúciós algoritmusok használata esetén is a jól ismert minimax, negamax algoritmusokat használjuk, csupán a statikus kiértékelőfüggvény meghatározásához használjuk az evolúciós algoritmusokat.

4.2.3. Sakk GA-val

Sakkprogramnál a statikus kiértékelő függvény egyik legfontosabb része a táblán lévő figurák anyagerőssége. A gyalog értékét 1-nek véve, meg kell határozni a többi figura értékét a gyaloghoz viszonyítva. Többnyire a futó és a huszár értékét 3-nak, a bástya értékét 5-nek, a vezér értékét 8-10-nek veszik. Az értékeket felhasználva könnyen összeszámolhatjuk a világos és sötét játékos figuráinak összértékét.

Egy egyszerű sakkprogram szerzője próbált GA-t használni a figurák értékeinek meghatározásához. A már korábban elkészült sakkprogram statikus kiértékelőfüggvényét módosította úgy, hogy a figurák értékét (és még egy-két konstanst) paraméternek vett. A paraméterekből rakta össze a kromoszómát, vagyis a GA egy egyede a statikus kiértékelő függvény néhány alapvető paraméterét határozta meg. A statikus kiértékelőfüggvény szerkezete, és az algoritmus további részei közösek voltak minden egyedben.

A GA-t használva a figurák értékeit meghatározó paraméterek hamar konvergáltak, a figurák értékei lényegében megegyeztek a korábban behuzalozott értékekkel. A többi paraméter vagy nem konvergált, vagy nem az optimális értékhez.

Az így kapott statikus kiértékelő függvényt használva az algoritmus kismértékben rosszabb lett, mint az eredeti változat. Ez nem feltétlenül jelenti azt, hogy nem érdemes GA-t használni, hiszen arról nincs információnk, hogy az eredeti program paramétereinek meghatározása milyen nehézségű volt.

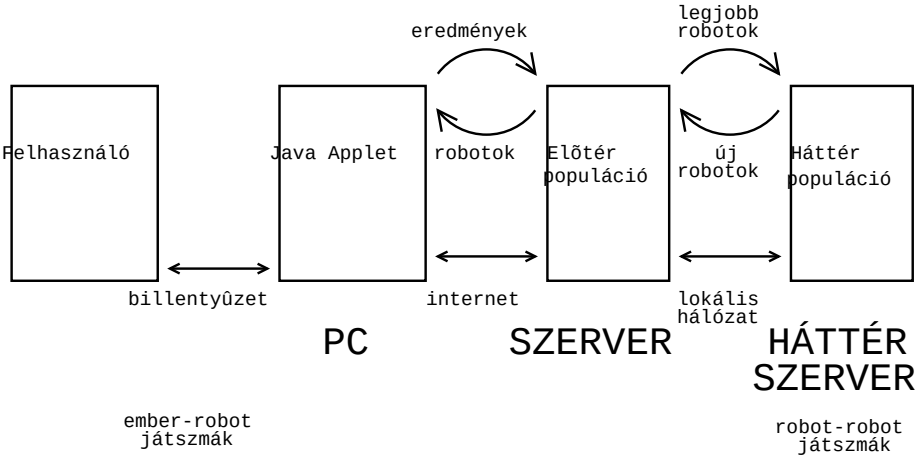
4.2.4. Tron GP-vel

Egy 1982-es Walt Disney film egy olyan virtuális világot mutatott be, ahol két motorbiciklis haladt konstans sebességgel. Csak derékszögben tudtak fordulni, maguk mögött falat húztak. A játéktér egyre inkább megtelt falakkal, míg végül valamelyik motoros falnak ütközött. A győztes az életben maradt motoros lett.

A játéknak sok számítógépes implementációja létezik, a szabályok időnként kismértékben eltérnek. A [FSJP97] cikk szerzőinél a pálya szélén nincs fal, a pályán kelet-nyugati és észak-déli irányban körbe lehet menni.

A játékot játszó robotok 8 irányban figyelik a világot. A szenzorok azt adják meg, van-e a közelben akadály (1: akadály közvetlen közel, 0: nincs akadály látótávolságon belül). A genetikus programozás (1.4.5. rész) során a nyolc szenzor értékeit (_A, _B, $\cdots$, _H), egy véletlen számot 0 és 1 között (R), alapműveleteket (+, -, *, % (biztonságos osztás)), elágazást (IFLTE - if a $\leq$ b then-else), és a fordulásért felelős szabályokat (LEFT, RIGHT) használhatunk. A fa maximális mélysége 7, maximális mérete 512 lehet.

A robotok játszhatnak egymás ellen, és emberek ellen is. Mindkét módszernek léteznek előnyei és hátrányai. Egymás ellen lényegesen több játszmát tudnak a robotok játszani, ugyanakkor túlzottan specializálódnak a robotok. A cikk írói egy olyan módszert választottak, ahol mind emberek, mind egymás ellen játszanak a robotok.



4.9. ábra. Tron

Egy Java applet segítségével az interneten keresztül bárki játszhat a robotok egy része ellen, miközben a számítógépen a robotok egymás ellen játszanak. Az emberek által elérhető robotok populációja időnként változik, hogy újabb robotok ellen lehessen játszani, és újabb robotok próbálhassák ki képességeiket az emberek ellen.A cikkből származó 4.9. ábrán látható a rendszer felépítése.

Az eredmények alapján az vehető észre, hogy az egymást követő generációkban egyre sikeresebbek robotok. Megfigyelték többek között azt is, hogy a robotok ellen játszó emberek átlagteljesítménye nem változik, de az egyes emberek teljesítménye gyorsan nő.

4.3. Gráfrajzolás

Gráfokat a számítástechnika sok területén használunk, például a fejlett vizuális fejlesztőeszközökben. Egy gráfot mindig többféleképpen ábrázolhatunk, és többnyire nem tudjuk eldönteni melyik ábrázolás a legjobb, hiszen nehéz definiálni mikor tekintünk egy ábrázolást jónak.

Ha adott egy G=(V,E) gráf, akkor a gráf rajzát (két vagy három dimenzióban) a következőképpen definiálhatjuk: Minden csúcshoz egy-egy pozíciót, minden élhez egy-egy görbét rendelünk hozzá, oly módon, hogy a görbék végpontjai az élhez tartozó csúcsokhoz rendelt pozíciókban legyenek.

Vannak bizonyos konvenciók, melyet a gráfrajzoló algoritmusok nagy része betart:

- Az algoritmusok egy része a csúcsokat egy rács csúcspontjain helyezi el, vagyis a csúcsokhoz rendelt pozíciók koordinátái egész számok.

- Az algoritmusok egy részében az éleket egyenes szakaszokból kapcsolhatjuk össze, vagyis egy él az egy törtvonal (*polyline*). Ha az él csak egyetlen szakaszból áll, akkor egyenes-vonalú rajzról beszélünk.

- Az előző konvenció speciális esete, ha a szakaszok mindig párhuzamosak valamelyik tengellyel. Ezeket a rajzokat hívjuk ortogonális rajzoknak.

Bár nehéz definiálni a jó gráfrajzokat, megfogalmazhatunk pár kritériumot, melyek alapján el lehet dönteni, hogy mennyire jó egy rajz:

- Az élkeresztezések elkerülése igen sok alkalmazásban fontos. Ha egy gráfot síkba lehet rajzolni úgy, hogy az élek nem keresztezik egymást, akkor a gráfot síkbarajzolhatónak nevezzük. Ez esetben a rajzolásnál törekednünk kell az ilyen ábrázolásra.

  Egy gráfot gyakran akkor is le kell síkban rajzolnunk, ha a gráf nem síkbarajzolható. Ebben ez esetben az élkeresztezések számát próbáljuk minimalizálni.

- A gráf csúcsfelbontása a két legközelebbi csúcs távolsága. Adott gráfméretre ezt a távolságot próbáljuk maximalizálni. Ha a csúcsok egy rács csúcspontjain helyezkednek el, akkor a csúcsfelbontás legalább 1, ebben az esetben a gráf rajzának méretét próbáljuk minimalizálni.

- Ha az élek törtvonalak, akkor a törések számát próbáljuk minimalizálni. Ha a törések száma 0, akkor minden él egyenes vonal.

A hagyományos gráfrajzoló algoritmusok többnyire egy kritériumot tartanak fontosnak, így többnyire a keletkező gráfrajzok a többi kritétiumnak nem felelnek meg. Nehezíti a feladat megoldását, hogy a legtöbb gráfrajzolással kapcsolatos probléma NP-nehéz.

A [Mic92] könyvben (e könyvből származnak a rész ábrái) leírt, genetikus algoritmusokon alapuló gráfrajzoló algoritmusok három kritériumot tartanak fontosnak, a jelenleg elkészült verziók azonban még nem foglalkoznak a második kritériummal.

- C1: Felfelé irányuló éleket el kell kerülni.

- C2: A csúcsokat egyenletesen kell elhelyezni.

- C3: Élkeresztezések számának minimálisnak kell lennie.

A kritériumok egy hagyományos gráfrajzoló algoritmusból származnak, mely algoritmus három fázisból állt. Minden fázis csak egy kritériummal foglalkozott, így a feladatot három részfeladattá bontotta. A feladat bonyolultságát jellemzi, hogy mind a három részfeladat NP-nehéz.

A két ismertetendő rendszer közül az egyik (GRAPH-1) a két szempontot egyszerre próbálja optimalizálni, a másik (GRAPH-2) az első fázisban az első , a második fázisban a harmadik kritérium alapján optimalizál. Ez az algoritmus tehát jobban hasonlít arra a hagyományos gráfrajzoló algoritmusra, melyből a kritériumok származnak.

Mindkét algoritmus felosztja a munkalapot négyzetekre. A gráf csúcsai egy-egy négyzetbe kerülnek, a gráf élei a négyzetek között húzott egyenes vonalak lesznek. A munkalap H négyzet magas és W négyzet széles.

Az, hogy az algoritmus csak egyenes élek használatát engedi meg, jelentősen leegyszerűsíti a problémát, hiszen így már a csúcsok elhelyezkedése is egyértelműen meghatározza a gráf rajzát.

A kiértékelőfüggvények elkészítéséhez figyelembe kell venni a korábban felsorolt kritériumokat. Négy függvényt definiálhatunk, melyeket majd felhasználhatunk a kiértékelőfüggvényben:

- $n_u(M)$: Felfelé irányuló élek száma.
- $n_h(M)$: Vízszintes élek száma.
- $n_c(M)$: Élkeresztezések száma.
- $n_o(M)$: Azon csúcsok száma, melyek azonos négyzetben vannak.

4.3.1. GRAPH-1

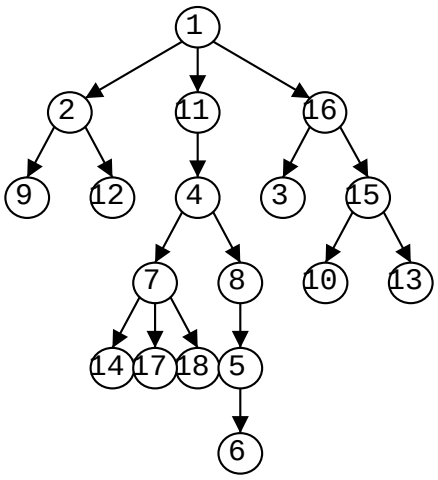
Ez a rendszer a talán legtermészetesebb reprezentációt választja, a kromoszóma egy $H \times W$-s mátrix, melynek elemei megfelelnek a munkamező négyzeteinek. A mátrixban lévő számok határozzák meg, hogy az adott négyzetben melyik (hányas számú) csúcs található.

A 4.10. ábrán látható gráfrajznak 6×10-es munkalapot használva a következő mátrix felel meg:

0	0	0	0	1	0	0	0	0	0
0	2	0	0	11	0	0	16	0	0
9	0	12	0	4	0	3	0	15	0
0	0	0	7	0	8	0	10	0	13
0	0	14	17	18	5	0	0	0	0
0	0	0	0	0	6	0	0	0	0

A mátrix nem tartalmazza a gráf éleit, mivel az az eredeti gráffal adott, és az algoritmus nem módosíthatja a gráf szerkezetét.

A kezdeti populáció egyedeinek olyan mátrixai vannak, melyekben az 1 és N (ahol N a csúcsok száma) közötti egész számok pontosan egyszer szerepelnek, a többi helyen pedig 0 található. Ilyen mátrixok könnyen generálhatóak.



4.10. ábra. G1 gráf

Operátorok

A rendszerben használt összes operátor úgy működik, hogy a leszármazottak is megfeleljenek a megszorításnak, vagyis az 1 és N közötti egész számok pontosan egyszer szerepeljenek a leszármazottakban.

- **Sorcsere**

  Két véletlenszerűen választott sor cseréje.

- **Oszlopcsere**

  Két véletlenszerűen választott oszlop cseréje.

- **Soron belüli csere**

  Egy sor két nemnulla elemének cseréje.

- **Oszlopon belüli csere**

  Egy oszlop két nemnulla elemének cseréje.

- **Egyszeres mutáció**

  Egy csúcs áthelyezése a mátrixban.

- **Kis mutáció**

  Két csúcs cseréje a mátrixban.

- **Nagy összefüggő mutáció**

  A mátrix két diszjunkt részének — melyek összefüggő sorokat és oszlopokat tartalmaznak — cseréje.

- **Sorinverzió**

  Egy sor elemei sorrendjének megfordítása.

- **Oszlopinverzió**

  Egy oszlop elemei sorrendjének megfordítása.

- **Sorrész inverzió**

  Egy sor egy részében az elemek sorrendjének megfordítása.

- **Oszloprész inverzió**

  Egy oszlop egy részében az elemek sorrendjének megfordítása.

- **Keresztezés**

  Az operátor először néhány oszlopot és sort választ ki. A két szülő mátrixának elemeit ezekben a sorokban, oszlopokban kicseréli. Ha a leszármazottakban nem szerepel minden szám 1 és N között pontosan egyszer, akkor a hibákat kijavítja.

- **Összefüggő keresztezés**

  Mint az előző, de összefüggő sorokat és oszlopokat választva.

A GRAPH-1 rendszer a következő kiértékelőfüggvényt alkalmazza:

$$Eval(M) = a_u n_u(M) + a_h n_h(M) + a_c n_c(M) + 1 \tag{4.3}$$

Az a_u, a_h, a_c együtthatók határozzák meg, mennyire fontosak az egyes kritériumok. A tesztek során $a_u = 2, a_h = a_c = 1$ értékeket használták a szerzők. A korábban felsorolt négy kritériumból a negyediknek ($n_o(M)$) a használatára nincs szükség, hiszen a reprezentáció nem engedi meg, hogy két csúcs azonos helyre kerüljön.

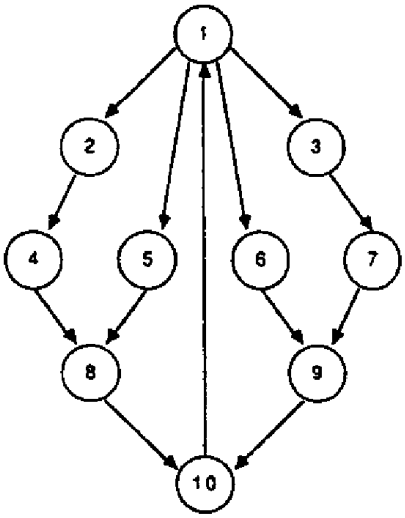
Bár a könyv nem említi, vélhetően az $f = \frac{1}{Eval(M)}$ függvény volt a fitneszfüggvény.

4.3.2. GRAPH-2

A GRAPH-2 rendszer kromoszómája egy $2 \times N$-es mátrix, ahol az i. oszlop az i. csúcs x és y koordinátáját tartalmazza. A 4.11. ábrán látható gráfrajz genotípusa 5×9-es munkalapon a következő mátrix (A (0,0) koordináta a bal alsó sarokban található):

N	1	2	3	4	5	6	7	8	9	10
x	4	2	6	0	3	5	8	2	6	4
y	4	3	3	2	2	2	2	1	1	0

Látszik, hogy a reprezentáció megengedi, hogy egy négyzetbe egynél több csúcs kerüljön. Ezek kiküszöbölését a kiértékelőfüggvény segítségével kell elérni. A mátrix méretét ebben az esetben a csúcsok száma határozza meg, szemben a GRAPH-1 rendszerrel, ahol a munkalap mérete.



4.11. ábra. G2 gráf

Első fázis

Az első fázisban a felfelé irányuló és vízszintesen haladó élek számának minimalizálása történik. Könnyen észrevehető, hogy a fázis során csak a csúcsok *y* koordinátájával kell foglalkozni.

Az első fázis kiértékelőfüggvénye a következő:

$$Eval(M) = a_u n_u(M) + a_h n_h(M) + 1 \tag{4.4}$$

Az a_u, a_h együtthatók értékei a tesztek során $a_u = 2, a_h = 1$ voltak.
Az első fázisban a következő operátorokat használjuk:

- **Y-csere**

  Két véletlenszerűen választott csúcs *y* koordinátájának cseréje.

- **Y-keresztezés**

  Két mátrix *y* koordinátáinak keresztezése.

- **Y-mutáció**

  Egy véletlenszerűen választott csúcs *y* koordinátájának véletlenszerű megváltoztatása.

Az első fázist különböző kezdeti populációból indulva többször lefuttatjuk, majd a kapott legjobb egyedek *y* koordinátáját felhasználva készítjük el a második fázis kezdeti populációját.

Második fázis

A második fázisban az élkeresztezések számát, és az egy négyzetben lévő csúcsok számát kell minimalizálni.

Az második fázis kiértékelőfüggvénye a következő:

$$Eval(M) = a_c n_c(M) + a_o n_o(M) + 1 \tag{4.5}$$

Az a_c, a_o együtthatók értékei a tesztek során $a_c = 1, a_o = 2$ voltak.

Mivel a második fázis során nem szabad az első fázis eredményét elrontani, az operátorok csak az x koordináta értékeit módosítják:

- **X-csere**

  Két véletlenszerűen választott csúcs x koordinátájának cseréje.

- **X-keresztezés**

  Két mátrix x koordinátáinak keresztezése.

- **X-mutáció**

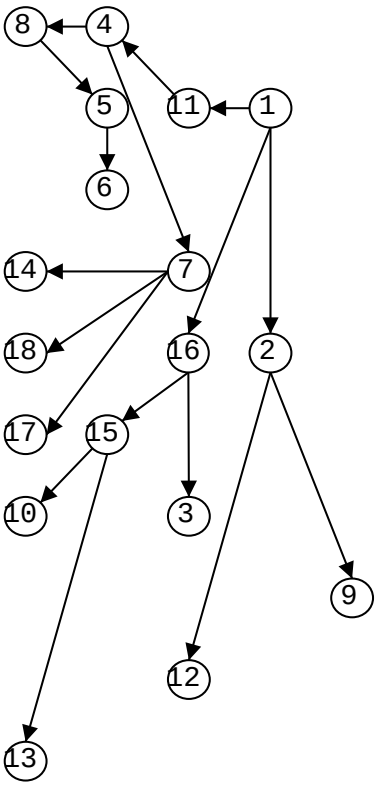
  Egy véletlenszerűen választott csúcs x koordinátájának véletlenszerű megváltoztatása.

HA megvizsgáljuk a két fázist, észrevehető, hogy a két fázis sorrendjét nem fordíthatjuk meg, mivel az első fázis elrontaná a második fázis eredményét.

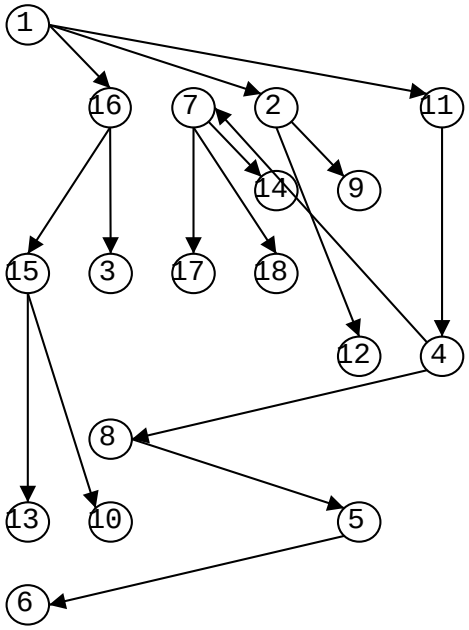
4.3.3. Eredmények

A tesztek során mindkét algoritmusnál 10x6-os munkalapot használtak. A genetikus algoritmusok a 200. generációig futottak, ez GRAPH-2 esetében 200-200 generációt jelent az első fázis összes futtatására, és a második fázis futtatására is.

Két gráf rajzát optimalizálták. Az eredmények G1 gráfra a 4.12. és 4.13. ábrán, G2 gráfra a 4.14. és 4.15. ábrán láthatóak. Bár a szerzők óvakodnak két gráf alapján messzemenő következtetéseket levonni, megállapítható, hogy G1 ügyesebben kerüli el az élkereszteződéseket, míg G2 a felfelé és vízszintes irányba irányuló éleket. Ezeket azonban okozhatják az együtthatók értékei is. A további következtetésekhez több gráf tesztje lenne szükséges, a szerzők azonban előbb a C2 kritérium figyelembevételét szeretnék integrálni az algoritmusokba.



4.12. ábra. GRAPH-1 megoldása G1-re



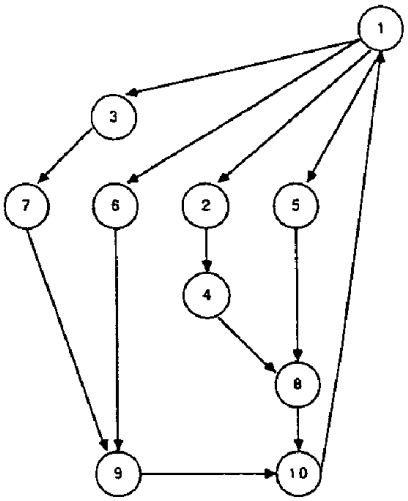
4.13. ábra. GRAPH-2 megoldása G1-re

4.4. Gépi tanulás

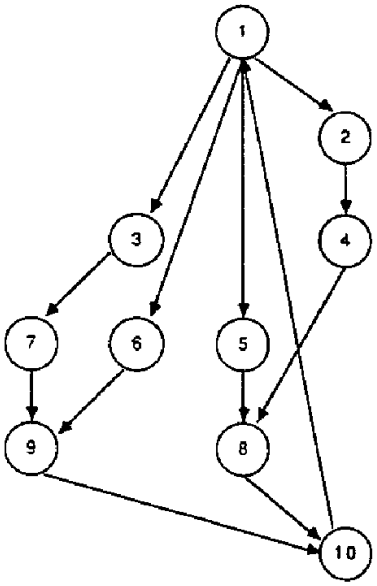
A gépi tanulás során olyan számítógépes programokat próbálnak készíteni, melyek automatikusan fejlődnek tapasztalataik által. A gépi tanulást a [Mit] könyvből lehet jobban megismerni, a könyv egy fejezete foglalkozik a gépi tanulás és a genetikus algoritmusok kapcsolatával. Ugyanerről szól a genetikus algoritmusokat ismertető [Mic92] könyv egy fejezete is. Gépi tanulásra példa egy olyan sakkprogram, mely ismeri a játék szabályait, és az általa játszott (eleinte túlnyomó részben elveszített), vagy sakkadatbázisokból letöltött játszmákat vizsgálva tanul sakkozni, és minél több játszmát vizsgál, annál jobban sakkozik.

A fejlődéshez szükséges tapasztalat tanulópéldákkal adott, melyek lehetnek *direkt*, vagy *indirekt* példák. Ha a tanulópélda sakkállásokból, és az állásokban lehetséges legjobb lépésekből áll, akkor direkt módon vannak meghatározva a legjobb lépések (ezeket szeretné a program megtanulni), ha a tanulópélda sakkjátszmákból áll, akkor a játszma kimeneteléből indirekt módon lehet következtetni a lépések jóságára.

A koncepció (fogalom, eszme) tanulás (*concept learning*) során is tanulópéldákból kell általános következtetéseket levonni. Egy fogalom általános definícióját kell megtalálni, a fogalomhoz tartozó, és fogalomhoz nem tartozó tanulópéldák segítségével.



4.14. ábra. GRAPH-1 megoldása G2-re



4.15. ábra. GRAPH-2 megoldása G2-re

Egy tanulópélda attribútumok halmazával írható le, a tanulandó fogalmat egy speciális attribútumnak tekintjük.

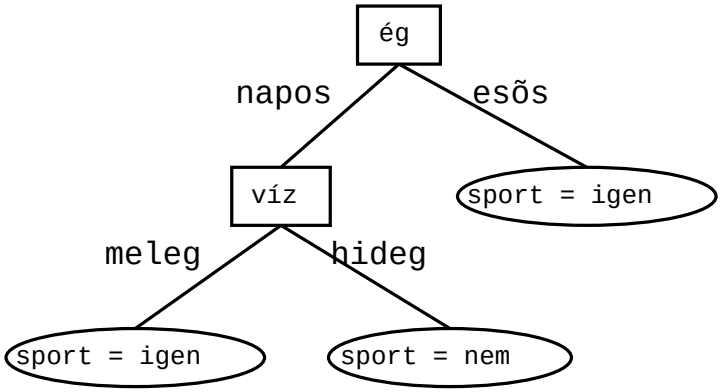
Ha különböző állatok alapvető jellemzői (tud-e repülni, tud-e úszni, vannak-e tollai, milyen súlyú, . . .) adottak, akkor a concept learning feladata lehet a ’madár’, vagy a ’hal’ fogalmának megtanulása.

A [Mit] könyv egy gyakran használ példáját idézve: Adott négy tanulópélda, nyolc attribútummal. A fogalom amit meg kell tanulni: „Napok, mikor barátom Aldo élvezi kedvenc vizisportját”. Az attribútumok közül a sport írja le a fogalmat, ennek értékét kell előrejelezni ha a másik hét attribútum értéke ismert.

példa	ég	hőmérséklet	páratartalom	szél	víz	előrejelzés	sport
1	napos	meleg	normális	erős	meleg	ua.	igen
2	napos	meleg	magas	erős	meleg	ua.	igen
3	esős	hideg	magas	erős	meleg	változás	nem
4	napos	meleg	magas	erős	hideg	változás	igen

Azokat a tanulópéldákat amelyek beletartoznak a fogalomba pozitív, melyek nem azokat negatív példáknak hívják. A fogalom tanulása során a tanulóalgoritmusok hipotéziseket állítanak fel, melyeket ellenőriznek, és segítségükkel újabb hipotézist állítanak fel.

A lehetséges hipotéziseket az úgynevezett hipotézistérben keresik az algoritmusok. A hipotézistér határozza meg milyen hipotéziseket kell az algoritmusnak megvizsgál-



4.16. ábra. Egy döntési fa

nia. A hipotézisek gyakran IF-THEN szabályok, például az előző példában egy lehetséges hipotézis az 'IF ég = napos és hőmérséklet = meleg THEN sport=igen'. Egy bonyolult fogalom gyakran nem írható le egyetlen szabály segítségével, ezért egy hipotézis szabályok diszjunkt halmaza. A szabályok diszjunkt halmaza legegyszerűbben döntési fa segítségével írható le, ezért a gépi tanulás egyik legismertebb ága a döntési fa generálás. A 4.16. ábrán látható egy egyszerű döntési fa, mely a következő szabályokat tartalmazza:

IF ég = napos $\wedge$ víz = meleg THEN sport=igen
IF ég = napos $\wedge$ víz = hideg THEN sport=nem
IF ég = esős THEN sport=igen

4.4.1. GABIL

A *DeJong* és társai által kifejlesztett GABIL rendszer GA segítségével valósítja meg a gépi tanulást.

Reprezentáció

Egy hipotézis több (előre nem meghatározható számú) diszjunkt IF-THEN szabályból áll. Minden egyes szabály fix hosszúságú bitvektorral leírható, de a változó számú szabályok miatt a kromoszóma hossza különbözik az egyes egyedeknél.

Egy szabály elkódolásához először meg kell határozni, milyen alakú szabályokat lehet leírni a rendszer segítségével. A leírható szabályok alakja a következő:
IF
$(attr_1 =$érték$_{11} \vee \ldots \vee$érték$_{1_{k_1}}) \wedge$
$(attr_2 =$érték$_{21} \vee \ldots \vee$érték$_{2_{k_2}}) \wedge$
$\vdots$

$(attr_l =$érték$_{l1} \vee \ldots \vee$érték$_{l_{k_l}}) \wedge$
$THEN$
$class = [yes|no]$

, ahol $attr_i$ attribútumnév, $ertek_{i_j}$ az i. attribútum egy lehetséges értéke, $class$ az a speciális attribútum, melynek értékét meg szeretnénk határozni.

Az előbb használt példahalmazban egy szabály lehet például:

IF (ég = napos $\vee$ ég = esős) $\wedge$ hőmérséklet = meleg THEN sport=igen

Az egyes attribútumokra vonatkozó részeket külön-külön kódoljuk el bitvektorrá, és ezeket a bitvektorokat összekapcsolva kapjuk meg a szabály bitvektorát. Ha például az ég attribútumnak három lehetséges értéke van (napos, esős, felhős), akkor a bitvektor három bitet tartalmaz, minden bit egy-egy lehetséges attribútumértékhez van hozzárendelve.

bitvektor	jelentés
100	ég = napos
010	ég = esős
001	ég = felhős
110	ég = napos $\vee$ ég = esős
101	ég = napos $\vee$ ég = felhős
011	ég = esős $\vee$ ég = felhős
111	ég = napos $\vee$ ég = esős $\vee$ ég = felhős
000	hamis

Genetikai operátorok

A GABIL rendszer a hagyományos bitváltó mutációt (2.7. rész) használja. A hagyományos keresztezési módszerek a változó kromoszómahossz miatt nem használhatóak. A rendszer ezek helyett a kétpontos keresztezés (2.8.2. rész) módosított változatát használja.

Az egyik kromoszómában hagyományos módon választjuk meg a kereszteződési pontokat. Amilyen arányban elmetszik a kereszteződési pontok az egyik kicsi fix hosszúságú részt, olyan arányban kell elmetszenie a kereszteződési pontoknak a másik kromoszóma valamely kicsi fix hosszúságú részét. Ha például a fix hossz 30 bit, és az első kromoszóma 120, a második 90 bit hosszú, akkor ha az első kromoszómát a 73. bitnél metszi a kereszteződési pont (vagyis a fix hosszú részt 13:17 arányban), akkor a másodikat a 13., 43., vagy 73. bitnél metszheti el.

Adaptáció

A szerzők az fent ismertetett algoritmust két érdekes dologgal egészítették ki. Két olyan új operátort vezettek be, melyek a szimbolikus tanulóalgoritmusoknál gyakran

hasznosnak bizonyultak. Az első operátor (*AddAlternative*) általánosítja a szabályt úgy, hogy egy attributumnak megfelelő bitvektorban 0-ról 1-re cserél egy bitet. Az operátor hatására például az 100 bitvektorból (jelentése pl. 'hőmérséklet = alacsony') 101 bitvektor ('hőmérséklet = alacsony $\vee$ hőmérséklet = közepes') lesz. A másik operátor (*DropCondition*) egy drasztikusabb általánosítást végez, egy attribútumra vonatkozó összes bitet 1-re cseréli, vagyis a szabályból kidobja az attribútumra vonatkozó feltételt. ('hőmérséklet = alacsony $\wedge$ ég = napos' $\rightarrow$ 'hőmérséklet = alacsony') A tesztek során a két operátor 1 illetve 60 %-os valószínűséggel módosította a kromoszómákat, és az algoritmus eredményessége lényegesen megnövekedett a két új operátor hatására. Fontos megjegyezni, hogy ezek a változások természetesen a hagyományos bitváltó mutáció hatására is létrejöhetnének, de csak sokkal kisebb valószínűséggel.

A rendszer egy másik továbbfejlesztésében adaptív genetikus algoritmusokat használtak, melyhez a kromoszómát is módosították (lásd 3.6.2. rész). A kromoszóma végéhez két új bitet illesztettek, melyek azt határozták meg, hogy az AddAlternative illetve DropCondition operátorok módosíthatják-e a kromoszómát. Az operátorok csak akkor léptek működésbe, ha a bitek értéke 1 volt, ellenkező esetben nem módosították a kromoszómát. A két új bit a kromoszóma szerves része volt, a korábbi operátorok (mutáció, keresztezés) hatására értékük folyamatosan módosult az algoritmus futása során. A szerzők tesztjei alapján az adaptáció az esetek egy részében sikeresnek, egy másik részében károsnak bizonyult.

4.5. Órarendkészítés

A rész a [AA91] cikken alapszik. Egy iskola órarendjét kell elkészíteni, előre adottak a diákok diszjunkt csoportjai (osztályok), osztálytermek, tanárok, tantárgyak. A tanórák lehetséges ideje előre meghatározott, fix számú ilyen lehetséges hely (periódus) van.

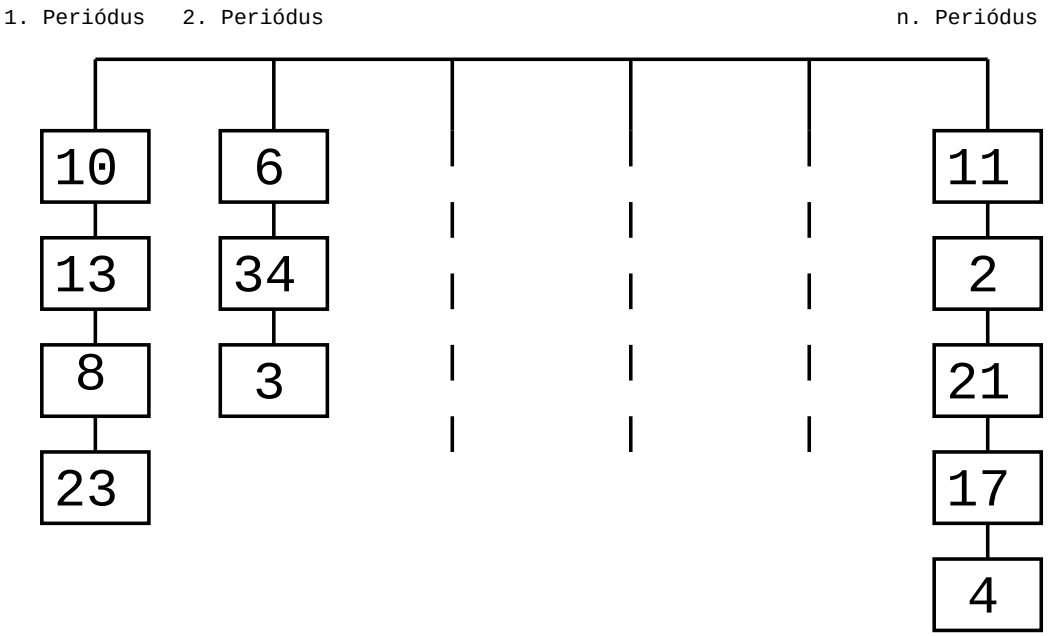
Az tanórák listájánál előre adott, hogy a tantárgyat mely osztály hallgatja, ki tanítja, és hol lesz az óra. Ezt a négyest (tantárgy, osztály, tanár, terem) a továbbiakban tanórának hívjuk. Ez a megszorítás egyszerűsíti a problémát.

Egy órarend értékeléséhez egy büntetőfüggvényt (2.9.1. rész) definiálunk, mely az elfogadható órarendeknél 0, a nem elfogadhatóaknál pozitív. A függvény három részből áll, a tanár-, osztály- és terembüntetésből.

Az osztály büntetőfüggvénye büntet minden alkalmat, amikor egy periódus során többször is szerepel a tanóra. A büntetés mértéke az előfordulások számánál eggyel kisebb. A következő képlet leírja a büntetés mértékét. $\mathcal{P}$ jelöli a periódusok halmazát, $n(c, p)$ a c osztály előfordulásainak számát p periódusban.

$$C(c) = \sum_{p \in \mathcal{P}} max(0, n(c, p) - 1) \tag{4.6}$$

A tanár- és terembüntetést hasonlóképpen definiálhatjuk.



4.17. ábra. Az órarend reprezentálása kromoszómákkal

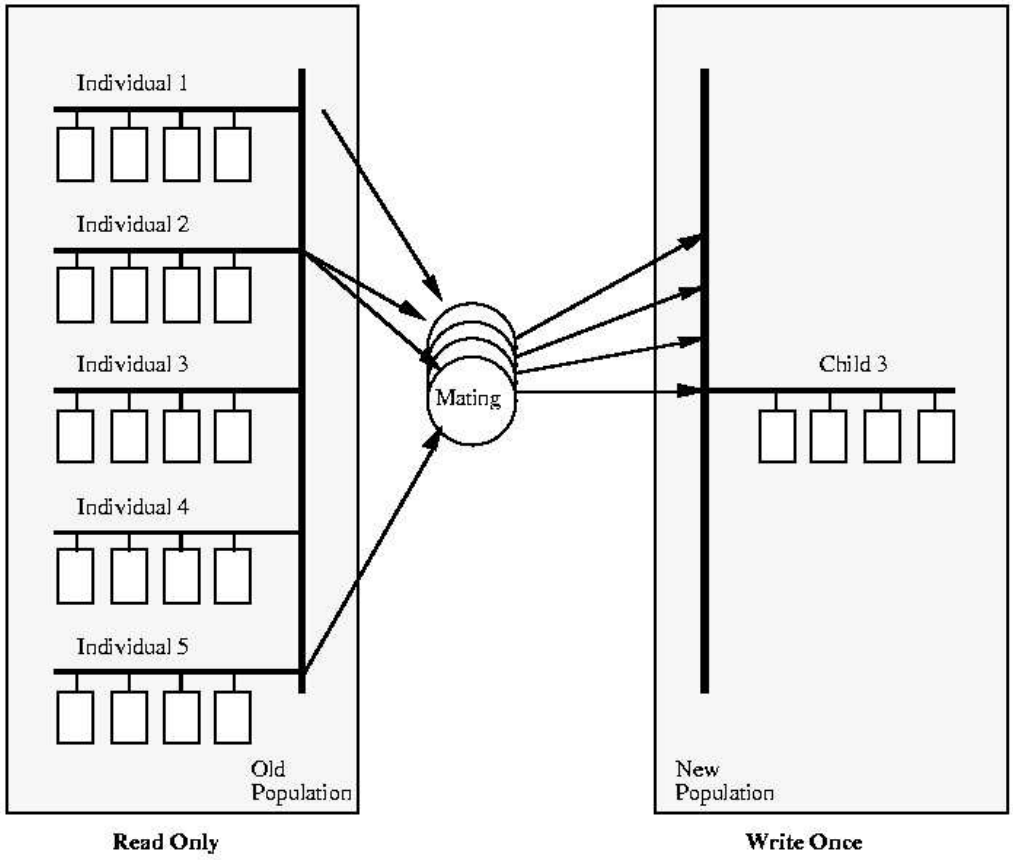
4.5.1. Reprezentáció

Az egyszerűség kedvéért a tanórákat 1-től n-ig sorszámozzuk meg. A genotípusban a tanórákat és a periódusokat kell egymáshoz rendelni. Bár ez sokféle módon megoldható lenne, a szerzők egy olyan módot választottak, mely során a genotípus nem egy kromoszómából (mint a korábbi példákban), hanem több kromoszómából áll. Minden egyes periódushoz egy kromoszóma tartozik, mely meghatározza, hogy az adott periódusban mely tanórák vannak megtartva. A kromoszómán belüli sorrend nem befolyásolja a fenotípust. A kromoszómák nem bitvektorok, a tanórákat jelölő számokat kódolatlanul tartalmazzák. A több kromoszóma haszna, hogy az egyes periódusokat reprezentáló kromoszómák egymástól függetlenebbül fejlődhetnek az algoritmus során. A 4.17. ábrán látható egy lehetséges genotípus. Az első kromoszóma négy tanórát, a második három tanórát tartalmaz.

4.5.2. Operátorok

Az algoritmus mutációs operátora véletlenszerűen választ egy órát valamely kromoszómából, és egy másik kromoszóma végéhez kapcsolja. A mutáció tehát több kromoszómát is érint.

A keresztezés során a kromoszómák egymástól függetlenül esnek át a keresztezésen. A kromoszómák azonos keresztezési módszert, az egypontos-keresztezést (2.8.1. rész) használják. Mivel a kromoszómák hossza nem feltétlenül azonos, az egypontos

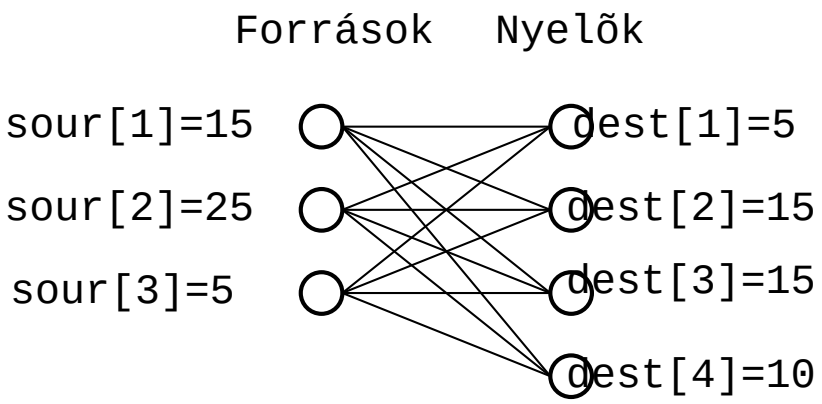


4.18. ábra. A párhuzamos működés alapelve

keresztezést kismértékben módosítani kell.

4.5.3. Párhuzamosság

Az algoritmus lassúsága miatt a szerzők párhuzamosságot használtak. Osztott memóriás többprocesszoros számítógépen globális populációt használó párhuzamos genetikus algoritmust (3.3.1. rész) használtak, az új populáció létrehozását párhuzamosították, kihasználva, hogy a keresztezések és mutációk egymástól függetlenül is elvégezhetőek. Bár a keresztezés önmaga is párhuzamosítható lenne, hiszen az egyed kromoszómáit párhuzamosan is keresztezni lehetne, a szerzők úgy vélték, a populáció mérete amúgy is meghaladja a processzorok számát, így a további párhuzamosítás nem járna jelentős sebességnövekedéssel. A párhuzamos működés során a szülőpopulációt csak olvasták a párhuzamos szálak, az új populációban pedig csak egy szál írt egy gyereket, így a szálak egymással párhuzamosan tudtak dolgozni. A párhuzamos működés alapelvét mutatja a 4.18. ábra.



4.19. ábra. Egy szállítási feladat

A tesztek során 1, 2, 5, 10 és 15 processzort használtak, és megvizsgálták, hogy a processzorok számának növekedésével miként csökken a végrehajtáshoz szükséges idő. Ideális esetben a gyorsulás aránya megegyezne a processzorok számával, ez a gyakorlatban nem érhető el. A két teszt során a maximális gyorsulás 9.2 és 9.3 volt, ezt 10 illetve 15 processzorral érték el.

Az algoritmust a tesztek során az összes esetben olyan órarendet készített, mely megfelelt az összes megszorításnak, vagyis a büntetőfüggvény értéke 0 volt. Az órarend elkészítéséhez szükséges generációszám 7 és 42973 között változott.

4.6. Szállítási feladat

A szállítási probléma egyike a legismertebb kombinatorikai problémáknak. Több változata ismert, itt a következőt vizsgáljuk: adott egy teljes páros gráf, a csúcsok két halmazát *forrás*oknak (n darab) és *nyelők*nek (k darab) nevezzük. A források előre meghatározott kínálattal ($sour(i)$), a nyelők előre meghatározott kereslettel ($dest(j)$) rendelkeznek. A 4.19. ábrán látható egy egyszerű szállítási feladat.

A forrásokat a nyelőkkel csövek kötik össze, melyeken anyagokat szállíthatunk. A szállítást költség terheli. Ha a költség egyenesen arányos a szállított anyag mennyiségével akkor *lineáris*, ha nem akkor *nemlineáris* problémáról beszélünk.

Lineáris esetben egységnyi anyag i. forrásból j. nyelőbe történő szállításának költségét $cost(i, j)$ jelöli. A költségfüggvény ebben az esetben $f_{ij}(x_{ij}) = cost_{ij} \cdot x_{ij}$.

A feladat meghatározni, hogy az egyes éleken mennyi anyag folyjon át, betarva a 4.7. – 4.9. megszorításokat. Az i. forrásból j. nyelőbe szállított anyag mennyiségét x_{ij} jelöli.

$$\sum_{j=1}^{k} x_{ij} \leq sour(i) \quad \forall \quad i = 1, 2, \ldots, n \tag{4.7}$$

$$\sum_{i=1}^{n} x_{ij} \geq dest(j) \quad \forall \quad j = 1, 2, \ldots, k \tag{4.8}$$

$$x_{ij} \geq 0 \quad \forall \quad i = 1, 2, \ldots, n \; s \; j = 1, 2, \ldots, k \tag{4.9}$$

Az első megszorítás azt fejezi ki, hogy a források legfeljebb a kínálat erejéig tudnak anyagot szolgáltatni, a második, hogy a nyelőknek legalább a keresletben meghatározott mennyiségű anyagot kell fogadniuk.

A megszorítások betartásán túl a cél a szállítás költségének minimalizálása:

$$min \sum_{i=1}^{n} \sum_{j=1}^{k} f_{ij}(x_{ij}) \tag{4.10}$$

A megszorításokból következik, hogy $\sum_{i=1}^{n} sour(i) \geq \sum_{j=1}^{k} dest(i)$ (vagyis az összkínálat nagyobb vagy egyenlő, mint az összkereslet), ellenkező esetben a feladat nem lenne megoldható. A szállítási feladat ***kiegyensúlyozott***, ha $\sum_{i=1}^{n} sour(i) = \sum_{j=1}^{k} dest(i)$. Ebben az esetben az első két megszorítás egyenlőség:

$$\sum_{j=1}^{k} x_{ij} \quad = \quad sour(i) \forall i = 1, 2, \ldots, n \tag{4.11}$$

$$\sum_{i=1}^{n} x_{ij} \quad = \quad dest(j) \forall j = 1, 2, \ldots, k \tag{4.12}$$

Ismert az a tény, hogy ha $sour$ és $dest$ tömb összes eleme egész, akkor a megoldás ($\forall i = 1, 2, \ldots, n$, $\forall j = 1, 2, \ldots, k \; : x_{ij}$) is csak egész értékeket tartalmaz, továbbá x_{ij} értékei közül legfeljebb $k + n - 1$ pozitív, a többi érték 0.

A következő két megoldást a [Mic92] könyv ismerteti.

4.6.1. Megoldás inicializáló függvénnyel

A megoldás során egyrészt figyelnünk kell arra, hogy a lehetséges megoldások megfeleljenek a megszorításoknak, másrészt a költséget is minimalizálni kell. Szerencsére könnyen találhatunk olyan inicializáló függvényt, mely olyan x_{ij} értékeket generál, melyek megfelelnek a megszorításoknak (4.7. és 4.8. egyenlőtlenségek). Amennyiben az operátorok (keresztezés, mutáció) megőrzik a megszorításokat, akkor biztosak lehetünk benne, hogy az algoritmus végén a megoldásaink is megfelelnek a megszorításoknak. Az algoritmus vázlatát mutatja a 4.1. algoritmus.

Az algoritmus az eredményt a v tömbbe írja. Az algoritmus futását mutatja be a következő példa. Legyen $sour$ és $dest$ tömb tartalma a következő:

sour[1]=15, sour[2]=25, sour[3]=5
dest[1]=5, dest[2]=15, dest[3]=15, dest[4]=10

4.1. algoritmus Inicializálás

Állítsuk a számokat 1 és $k \cdot n$ között meg nem vizsgáltra.
repeat
Válasszunk egy meg nem vizsgált számot véletlenül (q), és állítsuk megvizsgáltra.
(sor) $i \leftarrow \lfloor (q-1)/k + 1 \rfloor$
(oszlop) $j \leftarrow (q-1)\ mod\ k + 1$
val $\leftarrow min(sour[i], dest[j])$
$v_{ij} \leftarrow$ val
$sour[i] \leftarrow sour[i] - val$
$dest[j] \leftarrow dest[j] - val$
until minden szám megvizsgált

Legyen az első véletlenszám 10, ekkor i értéke 3, j értéke 2. Ekkor val értéke $min(sour[3], dest[2]) = 5$, $sour[3]$ és $dest[2]$ új értéke 0 illetve 10. Ha a következő szám 8 (i=2, j=4), akkor $val = min(sour[2], dest[4]) = 10$, $sour[2]$ és $dest[4]$ új értékei 15 és 0.

Feltételezve, hogy a véletlen számok további sorrendje 5, 3, 1, 11, 4, 12, 7, 6, 9, 2, az algoritmus végül a következő mátrixot készíti el:

	5	15	15	10
15	0	0	15	0
25	5	10	0	10
5	0	5	0	0

Az inicializáló függvényben a véletlenszerűen választott számok sorrendje határozza meg, hogy a megszorításnak megfelelő tömbök közül melyiket generálja a függvény.

Bár eredetileg csak azért kerestünk inicializáló függvényt, hogy a kezdeti populációt feltöltsük, és utána az így keletkezett kromoszómákat módosító operátorokat használjunk, másként is felhasználhatjuk az inicializáló függvényt.

Érdekes lehetőség, hogy a kromoszómában a véletlenszerűen választott mezők sorrendjét tároljuk el. A kromoszóma (egy számvektor) így indirekt módon határozza meg a mátrixot, mely segítségével már kiszámítható a költség.

Operátorok

Mivel a kromoszóma számsorozatot tartalmaz, az operátorok ezt a számsorozatot módosítják, nem a mátrixot. A kromoszóma módosításával természetesen indirekt módon módosul a mátrix is.

- **inverzió**

A vektor elemeinek sorrendjét fordítja meg $(x_1, x_2, \ldots, x_q) \rightarrow (x_q, x_{q-1}, \ldots, x_1)$.

- **mutáció**

  A vektor két elemét cseréli meg.

- **keresztezés**

  A keresztezés a 3.1.2. részben leírt PMX keresztezésre hasonlít. A keresztezés három lépésből áll:

  – Másolat készítünk a második szülőről.
  – Véletlenszerűen kiválasztunk egy részt az első szülő kromoszómájából.
  – Átmásoljuk a kiválasztott kromoszómarészt a gyerekbe úgy, hogy csak minimális mértékben módosítsuk a kromoszómát, ugyanakkor megfeleljen a megszorításoknak.

  Ha a szülők kromoszómái a

  (1, 2, 3 | 4, 5, 6, 7 | 8, 9,10,11,12)

  (7, 3, 1 | 11, 4,12, 5 | 2,10, 9, 6, 8)

  vektorokat tartalmazzák, akkor a leszármazott vektora a következő:

  (3, 1,11 | 4, 5, 6, 7 | 12, 2,10, 9, 8)

A módszeren alapszik a [Mic92] könyben ismertetett GENETIC-1 rendszer.

4.6.2. Mátrixos reprezentáció

Talán a leglogikusabb, ha a kromoszóma egy kétdimenziós mátrix, ahol az i. sor j. oszlopa x_{ij} értékét tartalmazza. Mivel a kromoszóma direkt módon tartalmazza a mátrixot, a költségfüggvény kiszámítása lényegesen egyszerűbb és gyorsabb mint az előző esetben.

Operátorok

A legegyszerűbb mutáció az lenne, ha a mátrix egy elemét változtatnánk meg. Sajnos ez elrontja a megszorításokat, további elemek változtatására lenne szükség, hogy a mátrix megfeleljen a megszorításoknak. Legalább 3 másik elem értékét kell módosítani, de gyakran ez sem elég, ezért ezt a mutációt a szerzők nem használták a szerzők.

Mutáció. A megoldás során egy másik mutációs operátort használtak. Véletlenszerűen választottak ki sorokat, oszlopokat, mindkettőből legalább 2-t. A sorokból és oszlopokból alkotott kisebb méretű mátrix egy új szállítási feladatot határoz meg. A kis mátrix értékeit töröljük, és az előző részben ismertetett inicializáló függvényt használjuk egy új kis mátrix létrehozásához. Az új értékeket az eredeti mátrixba visszaírva kapjuk meg a mutáción átesett kromoszómát.

	5	15	15	10
15	0	0	15	0
25	**5**	**10**	0	10
5	**0**	**5**	0	0

Ha például a már korábban is használt mátrixban az első két oszlopot és az utolsó két sort választjuk ki, akkor egy 2×2-es szállítási feladatot kapunk:

	5	15
15	5	10
5	0	5

Az eredeti értékeket kitörölve, az inicializáló algoritmus segítségével egy másik megoldást is kaphatunk a 2×2-es feladatra:

	5	15
15	0	15
5	5	0

Az értékeket visszamásolva, megkapjuk a mutáción átesett mátriot:

	5	15	15	10
15	0	0	15	0
25	**0**	**15**	0	10
5	**5**	**0**	0	0

Keresztezés. A keresztezés során először a két szülő mátrixából (V_1, V_2) két új mátrixot készítünk. DIV mátrix a két szülőmátrix értékeinek kerekített átlagát tartalmazza, REM mátrix elemeinek értéke attól függően 1 vagy 0, hogy az adott elemnél szükség volt-e kerekítésre. A kér mátrix értékeit a következő módon tudjuk kiszámítani:

$$div_{ij} = \lfloor (v_{ij}^1 + v_{ij}^2)/2 \rfloor \tag{4.13}$$

$$rem_{ij} = (v_{ij}^1 + v_{ij}^2) \; mod \; 2 \tag{4.14}$$

$$\tag{4.15}$$

Megvizsgálva REM mátrixot észrevehető, hogy REM soraiban és oszlopaiban az elemek összege mindig páros. Ez lehetőséget nyújt arra, hogy REM mátrixot két újabb mátrixra (REM_1, REM_2) bontsuk, a következő megszorításokat betartva:

$$REM = REM_1 + REM_2 \tag{4.16}$$
$$sour_{REM_1}[i] = sour_{REM_2}[i] = sour_{REM}[i]/2, \forall i = 1, \ldots, k \tag{4.17}$$
$$dest_{REM_1}[i] = dest_{REM_2}[i] = dest_{REM}[i]/2, \forall j = 1, \ldots, n \tag{4.18}$$

A két új mátrixot felhasználva már definiálható a két leszármazott:

$$V_3 = DIV + REM_1 \tag{4.19}$$
$$V_4 = DIV + REM_2 \tag{4.20}$$

A következő mátrixok egy keresztezés lépéseit mutatják be:

V_1	3	5	10	7	5
8	1	0	0	7	0
4	0	4	0	0	0
12	2	1	4	0	5
6	0	0	6	0	0

V_2	3	5	10	7	5
8	0	0	5	0	3
4	0	4	0	0	0
12	0	0	5	7	0
6	3	1	0	0	2

DIV	2	4	9	6	4
6	0	0	2	3	1
4	0	4	0	0	0
10	1	0	4	3	2
5	1	0	3	0	1

REM	2	2	2	2	2
4	1	0	1	1	1
0	0	0	0	0	0
4	0	1	1	1	1
2	1	1	0	0	0

REM_1	1	1	1	1	1
2	0	0	1	0	1
0	0	0	0	0	0
2	0	1	0	1	0
1	1	0	0	0	0

REM_2	1	1	1	1	1
2	1	0	0	1	0
0	0	0	0	0	0
2	0	0	1	0	1
1	0	1	0	0	0

V_3	3	5	10	7	5
8	0	0	3	3	2
4	0	4	0	0	0
12	1	1	4	4	2
6	2	0	3	0	1

V_4	3	5	10	7	5
8	1	0	2	4	1
4	0	4	0	0	0
12	1	0	5	3	3
6	1	1	3	0	1

A módszeren alapszik a [Mic92] könyvben ismertetett GENETIC-2 rendszer.

4.6.3. Eredmények

Lineáris esetben a genetikus algoritmust használó algoritmusok eredménye meg sem közelíti a hagyományos módszerek eredményeit. A genetikus algoritmus előnye akkor jelentkezik, amikor nemlineáris problémára alkalmazzuk. Míg a hagyományos módszerek több helyen kihasználják a linearitást, ezért nem alkalmazhatóak nemlineáris problémákra, a GA-s megoldások sokkal általánosabbak, és minimális módosítással (A szerzők a GENETIC-2 algoritmust módosították.) képesek a nemlineáris problémák kezelésére.

A tesztek alapján a két GA-ra épülő algoritmus közül a GENETIC-2 bizonyult jobbnak. Az algoritmusokat összehasonlították egy széles körben elterjedt GAMS rendszerrel, melynél jobbnak bizonyultak. Az összehasonlításnál figyelembe kell vennie azonban, hogy a GENETIC-2 rendszert kifejezetten szállítási problémák megoldására készítették, míg a GAMS rendszert egy ennél általánosabb problémakör megoldására.

5. fejezet

GALOPPS

Ha genetikus algoritmusokat szeretnénk használni, akkor két választási lehetőségünk van. Egyrészt saját magunk megírhatjuk a szükséges algoritmust, másrészt más által elkészített GA csomagot is használhatunk. Ebben a fejezetben az egyik elterjedt GA csomagot, a GALOPPS („Genetic ALgorithm Optimized for Portability and Parallelism” System = Hordozhatóságra és Párhuzamosságra optimalizált Genetikus Algoritmus) csomagot ismerhetjük be.

A megfelelő GA csomag kiválasztásakor több szempontot is figyelembe kell vennünk. A következő lista a szempontok egy részét tartalmazza:

- Általános célú

  A GA csomagok egy része csak a feladatok egy speciális részhalmazával tud foglalkozni, például csak permutációs problémákkal, vagy csak egy hangyaboly szimulációjával. Ha a megoldandó problémánk nem része a speciális részhalmaznak, vagy több feladatot is szeretnénk megoldani, érdemes általános célú GA-t használnunk.

- Ingyenes

  Mivel a GA csomagok nagy részét oktatási célokkal készítik, a csomagok túlnyomó többsége ingyenes.

- Forráskód elérhető

  Egyrészt lehetőséget nyújt a forráskód módosítására (például kisebb bugok javítása, apróbb továbbfejlesztések), másrészt ha a dokumentáció alapján bizonytalanok vagyunk a csomag egy részének működésében, ellenőrizhető miként működnek a csomag egyes részei. Egy jól megírt csomag forráskódjának tanulmányozásából amúgy is sokat megtanulhatunk.

- Hordozható

Minél több operációs rendszer alatt működik a csomag, annál tágabb körben használhatjuk a csomagot. A GALOPPS csomag DOS, Windows, UNIX (pl. Linux) és Macintosh rendszereken működik. (Személyes tapasztalat: Linux (Mandrake 7.0) és Windows NT alatt működött, Windows 98 alatt nem)

- Bővíthető

  Ha a GA csomag által kínált lehetőséget szűknek érezzük, akkor lehetőségünk van arra, hogy például új operátorokkal bővítsük a GALOPPS csomagot.

- Párhuzamos GA-t támogatja

  Minimális időráfordítással elérhetjük, hogy szekvenciális és párhuzamos GA segítségével is megoldjuk ugyanazt a problémát.

5.1. A kromoszóma felépítése

Egy probléma megoldásánál az első fontos döntés annak az eldöntése, miként fogjuk reprezentálni a lehetséges megoldásokat. A GALOPPS csomag egyik legfőbb előnye, hogy sokféle módon meghatározhatjuk a kromoszóma felépítését. A kromoszómareprezentációról szóló 2.6. részben ismertetett reprezentációk mindegyike használható, a kromoszóma lehet bit- és sztringvektor is. Az ismertetett alkalmazások során gyakran használt mátrix kromoszómát nem támogatja közvetlenül a rendszer, de egy egydimenziós tömbben könnyen tárolhatjuk a mátrix elemeit.

A kromoszóma mezőkből áll, ezek számát a *numfields* paraméter adja meg. Két további paraméter határozza meg a kromoszóma felépítését: *alpha_size* és *permproblem*.

- *permproblem=y*

  A *permproblem* paraméter határozza meg, hogy a megoldandó problémánk permutációs probléma-e (3.1.3. rész). Ha értéke igaz, akkor minden mező egy egész számot reprezentál 0 és *numfields*-1 között, és minden szám pontosan egyszer szerepel a kromoszómában.

- *permproblem=n, alpha_size=2*

  Az *alpha_size* paraméter a Σ ABC (ld. 2.6.) méretét határozza meg. Ha értéke 2, akkor az ABC-nek két lehetséges értéke van: 0 és 1. Vagyis ebben az esetben a kromoszóma egy *numfields* bitből álló bitvektor. Ezt a bitvektort tetszőlegesen értelmezhetjük, mi határozzuk meg, hogy melyik bit mit jelent, az egyes gének hány bitből állnak. A bitvektor egy részét számmá konvertálhatjuk az `ithruj2int` függvény segítségével.

- *permproblem*=n, *alpha_size* > 2

Ebben az esetben a kromoszóma sztringvektor, minden mező *alpha_size* különböző értéket vehet fel (0,1,... *alpha_size*-1).

- *permproblem*=n, *alpha_size* < 2

  Minden mező egy-egy egész számot reprezentál, mind az előző esetben, azonban az ábrázolható maximális szám nem feltétlenül egyezik meg. Az i. mezővel ábrázolható maximális érték *field_sizes[i] -1*. Az 5.8. részben láthatunk egy példát ilyen esetre. A kromoszómát `int` tömbbé alakíthatjuk a `chromtointarray` függvény segítségével.

5.2. A paraméterek meghatározása

A reprezentációt befolyásoló paraméterek mellett, még nagyon sok más paraméter értékét is meg kell határoznunk. Az egyes paramétereket többféle módon is meghatározhatjuk a csomagban. A módszerek többsége csak a paraméterek egy-egy csoportjánál használható.

- Makefile segítségével

  Bizonyos alapvető operátorokból (pl. keresztezés, mutáció) egy probléma megoldása során csak egyfajtát használhatunk a GALOPPS csomagban. Minden egyes lehetséges operátor implementálását azonos interfésszel rendelkező külön-külön forrásfile-ok tartalmazzák. A makefile-ban környezeti változóknak kell értékül adni, hogy mely forrásfile-okat használja a rendszer.

  - SELECT

    Lehetséges szelekciós módszerek (5 darab), például a rulettkerék módszer (2.5. rész)

  - CROSSOVER

    Lehetséges keresztezési módszerek, például egypontos (2.8.1. rész), kétpontos (2.8.2. rész), uniform (2.8.3. rész), OX (3.1.2. rész), CX (3.1.2. rész), PMX (3.1.2. rész).

  - MUTATION

    Lehetséges mutációs módszerek, például a bitváltó mutáció (2.7. rész).

  - INVERSION

    Lehetséges inverziós (3.2.) módszerek.

- Forráskódban

  A paraméterek egy részét inicializáló függvényekben is be tudjuk beállítani. Forráskódban rugalmasabban tudunk paramétereket állítani, ugyanakkor minden

paramétermódosításhoz a kód újrafordítása szükséges, hasonlóan a makefile-os módszerhez.

Többnyire csak a paraméterek egy szűk csoportját állítjuk be így, például a már említett *field_sizes* tömböt. Erre is példa az 5.8. rész.

- Parancssorral

  A paramétereknek egy kis részét parancssori paraméterként is megadhatjuk. Mivel a paraméterek nagy részét amúgy sem tudjuk így meghatározni, érdemesebb inkább input file-t használni.

- Billentyűzetről beolvasva

  Ha nem adunk meg input file-t, akkor a program kérdéseket tesz fel, és interaktívan megadhatjuk a paramétereket. Mivel a feltett kérdések száma elég nagy, e módszer helyett is érdemesebb input file-t használni.

- Input file-lal

  Az előző két módszer helyett érdemesebb input file-t használni. Az input file nevét a `-i` opcióval adhatjuk meg.

5.3. Callback függvények

Egy GA csomagnak tartalmaznia kell többek között alapvető genetikai operátorok implementációit, és a genetikus algoritmus alapalgoritmusát (2.1. rész), ahonnan a genetikai operátorokat meghívjuk. Ha több információt szeretnénk az algoritmus futásáról, például minden esetben amikor az eddigi legjobb egyednél jobbat találunk ki szeretnénk írni valamit, akkor módosítanunk kellene az alapalgoritmust.

Ez nyilván nem elfogadható megoldás, hiszen nem módosíthatjuk egy általános célú csomag függvényeit saját megoldásunk kedvéért.

Callback függvények használatával megoldhatjuk az előző problémát. Az alapalgoritmus a GALOPPS csomagban úgy van kiegészítve, hogy több helyen úgynevezett callback függvényeket hív meg. Például a csomag minden esetben ha az eddigi legjobb egyednél jobbat talál, meghívja az `app_new_global_best_report()` függvényt. A megoldásunkhoz minden callback függvényt implementálnunk kell, de az esetek többségében a callback függvények nagy részét üresen hagyhatjuk.

5.4. A work alkönyvtár

A sok callback függvény hátránya, hogy még egy nagyon egyszerű feladatot megoldó program forráskódja sem túl rövid. Mivel a kód nagy része (pl. üres callback függvé-

nyek, Makefile nagy része) nem függ a feladattól, a GALOPPS csomag tartalmaz egy alkönyvtárat (work), melyet template-ként felhasználhatunk.

Az alkönyvtár a következő file-okat tartalmazza:

- `makefile`

  A makefile. Segítségével a szekvenciális és a párhuzamos változata is lefordítható a programnak.

- `appxxxxx.c`

  Az alkalmazásunk forrását tartalmazó file. Üres callback-függvényeket, és nagyon egyszerűen implementált függvényeket tartalmaz. Például a fitnesz függvény implementációja, minden esetben 10-et ad vissza fitneszértéknek.

- `appxxxxx.in`

  Input file szekvenciális (Onepop) futtatáshoz.

- `appxxxx8.in`

  Input file párhuzamos (Manypops) futtatáshoz.

- `8pop2nbr.mst`

  Példa master file (5.6.1. rész), 8 alpopulációval, alpopulációnként két-két szomszéddal.

- `continu1.in`

  Megszakított futtatás folytatásának paramétereit leíró file, szekvenciális esetben.

- `continu8.in`

  Megszakított futtatás folytatásának paramétereit leíró file, párhuzamos esetben.

5.5. Egy egyszerű példa

Példaként oldjuk meg a 2.4. részben is leírt feladatot, vagyis maximalizáljuk az $x \cdot (sin(x) + 1)$ függvényt, $x \in [0, 20 \cdot \pi]$ esetén.

- Másolás

  Másoljuk le a work alkönyvtár tartalmát saját alkönyvtárunkba.

- Átnevezés

  Nevezzük át az `appxxxxx.c`, `appxxxx8.in`, `appxxxxx.in` file-okat rendre `app_t1.c`, `app_t1_8.in`, `app_t1.in` névre.

- Makefile módosítása

  A makefile-ban A 'H' és 'S' változók tartalmazzák a header file-ok (alapértelmezés: `../include`), és a galopps forrásfile-ok (alapértelmezés: `../src`) elérési útvonalát. Tartalmukat írjuk át úgy, hogy a megfelelő helyre mutassanak.

  Az 'APP' változó tartalmazza az alkalmazás fő file-jának nevét, értékét írjuk át `appxxxxx.c` -ről `app_t1.c` -re.

  Az egyszerűség kedvéért a makefile által alapértelmezésként javasolt keresztezési és mutációs módszert használjuk.

- Fitnesz függvény implementálása

  Bár `app_t1.c` file nagyon sok függvényt tartalmaz, melyeket mind módosíthatnánk elég ha csak a fitnesz függvény implementálását módosítjuk. A *void objfunc(critter)* függvényt kell módosítanunk (ez az első függvény a file-ban). A függvény az eredeti változatban néhány inicializáló sor (pl. a függvény számolja, hány alkalommal lett a fitnesz kiszámítva) után egyetlen egy sort tartalmaz:

  ```
  critter->init_fitness       = 10.;
  ```

  Az eredeti változatban minden egyed fitnesze 10, ezt módosítanunk kell:

  ```
  ival  = ithruj2int(1,10,      critter->chrom);
  dval  = 1.0  * ival  / pow(2,10)    * 20  * 3.1415;
  critter->init_fitness       = dval  * (sin(dval)+1);
  ```

  Az első sor segítségével a kromoszóma első és tizedik bitje közötti részt egész számmá konvertáljuk. A második sor skálázza át ezt a 0 és 1023 közötti számot 0 és $20 \cdot \pi$ közé. Az utolsó sor végzi el a valódi fitneszszámítást.

- Input file módosítása

  A paramétereket tartalmazó file-ok közül az egyszerűség kedvéért csak a szekvenciális futásért felelős `app_t1.in` file-t módosítjuk. A legtöbb paraméter értékét nem módosítjuk, a maximális generációszámot (*maxgen*) 20-ra, a populáció méretét (*popsize*) 20-ra, a mutációs rátát (*pmutation*) 0.01-re növeljük.

  Az a file tartalmazza a kromoszóma hosszát is, melynek értéke alapértelmezés szerint 10.

  ```
  maxgen   = 20
  popsize   = 20
  pmutation    = .01
  ```

- Fordítás

  A ’make Onepop’ parancs hatására lefordulnak a szükséges file-ok, és elészül a futtatható file, melynek neve *Onepop*. (A párhuzamos futtatáshoz szükséges file a ’make Manypops’ parancs hatására készül el.)

- Futtatás

  Az elkészült programot a ’Onepop -i app_t1.in -o app_t1.out’ paranccsal indíthatjuk el. A program `app_t1.in` file-ból veszi az input paramétereket, és az eredményt az `app_t1.out` file-ba írja. Az így keletkezett file nagyon sok információt tartalmaz, esetünkben a hossza kb. 70K.

5.6. Párhuzamosság

A GALOPPS csomag a durva szemcsés párhuzamosságot (3.3.2. rész) támogatja. A párhuzamosság három alapvető módon valósulhat meg:

- Szimulált párhuzamosság - 1 process

  Az alkalmazás egyetlen szálat futtat, és saját maga ütemezi az alpopulációkat, melyek szinkron módon futnak, vagyis az alpopulációk sorban egymás után kapnak időszeletet. Mivel egy futattható file-unk van, a fordítás során meghatározott paraméterek (pl. keresztezés) megegyeznek az egyes alpopulációkban, a többi paraméterben azonban lehet eltérés. (pl. keresztezési ráta).

- Szimulált párhuzamosság - több process

  Ebben az esetben is egyetlen gépen futnak a programok, így a párhuzamosság csak szimulált, azonban az előző esettel szemben itt minden alpopuláció külön szálként fut, az ütemezést az operációs rendszer végzi. Ez lehetőséget ad arra, hogy a különböző alpopulációk a fordítás során meghatározott paraméterekben is eltérjenek.

  A processek file-okon keresztül kommunikálnak, egyszerűen lockolják a file-t, melyet írni szeretnének.

- Valódi párhuzamosság

  Az egyes alpopulációkat kezelő programok (maximum 99) külön gépen futnak. Mivel ebben az esetben is file-ok segítségével kommunikálnak, kell lennie egy alkönyvtárnak, melyet minden process elérhet.

  Mivel a GALOPPS csomag hordozható, lehetőség van arra, hogy a programok különböző architectúrájú gépeken fussanak. Két megszorítás van csupán, a szóhossznak és byte-sorrendnek (little-endian, big-endian) meg kell egyeznie.

5.6.1. Master File (.mst) formátuma

Az egyes alpopulációk közti kapcsolatot az *mst* kiterjesztésű master file segítségével írhatjuk le. Leírhatjuk a migráció három alapvető jellemzőjét (3.3.2. rész), a topológiát és a migrációs rátát.

A file-nak két lehetséges formátuma van, egy hosszabb általános, és egy rövidebb speciális. Először mindkettőre látunk egy-egy példát (a GALOPPS dokumentációjából), majd a formátum leírása következik.

Rövidített formátum:

```
PÉLDA MASTER FILE:          | MAGYARÁZAT
subcnt  = -4                | 4 alpopuláció van, a "-" azt jelenti,
                              hogy mindegyik azonos mintát követ
subpop  = 0 2               | A 0. alpopulációnak 2 szomszédja van
  neighbor  = 1  2  3  4    | Az 1. alpopulációból 2 véletlenül
                              választott egyedet kap.
  neighbor  = 3  2  3  3    | A 3. alpopulációból 2 véletlenül
                              választott egyedet kap.
```

Általános formátum:

```
PÉLDA MASTER FILE:          | MAGYARÁZAT
subcnt  = 4                 | 4 alpopuláció van
subpop  = 0 2               | A 0. alpopulációnak 2 szomszédja van
  neighbor  = 1  2  3  4    | Az 1. alpopulációból 2 véletlenül
                              választott egyedet kap
                              migration_interval_reduction = 3;
                              migration_crowding_amt = 4
  neighbor  = 3  2  3  4    | A 3. alpopulációból 2 véletlenül
                              választott egyedet kap
subpop  = 1 1               | Az 1. alpopulációnak 1 szomszédja van
  neighbor  = 0 -1  4  3    | A 0. alpopulációból a legjobb egyedet
                              kapja
subpop  = 2 2               | A 2. alpopulációnak 2 szomszédja van
  neighbor  = 3  2  3  4    | Az 3. alpopulációból 2 véletlenül
                              választott egyedet kap
  neighbor  = 1 -2  3  4    | Az 1. alpopulációból a legjobb, és
                              egy véletlenül választott egyedet kap
subpop  = 3 1               | A 3. alpopulációnak 1 szomszédja van
  neighbor  = 0  2 3  3     | A 0. alpopulációból 2 véletlenül
                              választott egyedet kap
```

A file formátuma:

```
subcnt   = <alpopulációk     száma>  vagy  -<alpopulációk     száma>
subpop   = <alpopuláció     index>  <szomszédok     száma>
neighbor    = <alpopuláció     index>  <FLAG>  \
              <migration_incest_reduction>  \
              <migration_crowding_amount>
```

Ha `subcnt` értéke negatív, akkor minden alpopuláció azonos mintát követ, ezért csak egy alpopulációra kell a `subpop` részt megadni, ha pozitív akkor az alpopulációk különböznek egymástól, így minden alpopulációra külön `subpop` rész vonatkozik.

A `neighbor` rész határozza meg, hogy a szomszédos alpopulációkból milyen elv alapján kerülnek át egyedek az alpopulációba.

FLAG jelentése:

- pozitív

  Az alpopulációba FLAG darab véletlenszerűen kiválasztott egyed kerül át.

- nulla

  Nem kerül át egyed az alpopulációba.

- negatív

  A legjobb egyed, és $|FLAG|$-1 darab véletlenszerűen kiválasztott egyed kerül át az alpopulációba.

A másik két paraméter jelentése:

- migration_incest_reduction

  A paraméter a véletlenszerűen választott egyedekre vonatkozik. Meghatározza azon egyedek számát, melyeket véletlenszerűen (valójában három véletlen egyed közül a legjobbat választva) választunk a donor alpopulációból. Ezeket az egyedeket összehasonlítjuk a fogadó alpopuláció legjobb egyedével, és a Hamming távolságot figyelembe véve legmesszebb lévő egyedet migráljuk át.

- migration_crowding_amount

  Azt határozza meg, hogy a migrálandó egyed melyik egyed helyére kerüljön a fogadó alpopulációban. A paraméter által meghatározott számú egyedet választunk véletlenül a fogadó populációból, és a migrálandó egyedhez Hamming távolságot figyelembe véve legközelebb lévő egyed helyére kerül a migrálandó.

Ha valaki ennél bonyolultabb migrációs módszert szeretne használni, akkor lehetősége van implementálni, a csomag támogatja ezt a bővítést.

5.7. Utazóügynök probléma megoldása

A probléma megoldását tartalmazza a GALOPPS csomag. A szükséges file-okat az examples/btsp alkönyvtárban találjuk meg.

Az input file-okat tartalmazó appbtsp.in file-ban a következő paramétereket érdemes megvizsgálni:

```
permproblem    = y    /* permutációs    probléma   */
numfields    = 10     /* mez ők száma   */
maxgen   = 30         /* maximális    genrációszám     */
popsize   = 100       /* populáció    mérete   */
```

A GALOPPS csomag megoldása a 3.2.3. részben leírt útvonal-vektoros ábrázolást alkalmazza.

A városok adatait egy .dst kiterjesztésű file-ból olvassa be a program. Ilyen file-okat egy mellékelt program segítségével lehet készíteni, ezt a mellékelt programot nem vizsgáljuk.

A file beolvasását egy callback függvény segítségével végezhetjük el. Minden alpopuláció inicializálásakor meghívódik a `void app_init()` függvény. Mivel a filebeolvasást elég egyetlen egyszer elvégeznünk (akkor is ha több alpopulációnk van), egy statikus változó bevezetésével érhetjük el, hogy a filebeolvasás csak egyszer történjen meg:

```
void  app_init()
{
  static   int      firstcall    = 1;


  ...
  if  (firstcall)    {
    firstcall    = 0;
    ...
    /*  filebeolvasas     */
  }
  ...
}
```

A legfontosabb függvény amit meg kell írnunk a fitnesz kiszámítását tartalmazó függvény. A következő kódrészlet mutatja a függvény implementálását. A városok távolságát a `distbetween` tömb tartalmazza.

```
void
objfunc(critter     )
```

```
/* Alkalmazás függő fitnesz függvény */
     struct individual *critter;
{
    int      i, start, stop, firstcity, city, prevcity;
    double   tmp;

    neval++;
    local_cycle_neval++;

    critter->neval    = neval;
    tmp  = 0.;

    start  = 1;
    stop  = fieldlength;
    firstcity   = ithruj2int(start, stop, critter->chrom);
    prevcity   = firstcity;
    for (i = 1; i < numfields;  i++) {
        /* A kromoszóma azon része, */
        /* mely az aktuális számot tartalmazza */
        start  = (i * fieldlength)  + 1;
        stop  = ((i + 1) * fieldlength);

        /* A kromoszómában található bitvektor */
        /*  egész számmá konvertálása */
        city  = ithruj2int(start, stop, critter->chrom);
        tmp += distbetween[prevcity][city];
        prevcity   = city;
    }
    tmp += distbetween[city][firstcity];
    /* Minimalizáland ó távolság konvertálása, hogy */
    /* megfeleljen a maximalizálandó fitnesz függvénynek */
    /* 1000. elosztva a távolsággal */
    critter->init_fitness = 1000. / tmp;
}
```

5.8. Példa változó mezőméretre

A csomaghoz mellékelt `examples/demoinv` program példa a változó mezőméretre, és az inverzió használatára is. A kromoszóma 20 számot tartalmaz, a fitneszfüggvény két részből áll össze. Egyrészt azt szeretnénk elérni, hogy a kromoszóma szimmetrikus legyen, másrészt a cél az, hogy a kromoszóma első 10 száma sorban 0, 1, 2, . . ., 9 legyen.

Ennek megfelelően a büntetőfüggvényt (2.9.1. rész) használó fitnesz kiszámítás

érdemi része a következő:

```
{
...
  chromtointarray    (fields, critter->chrom)    ;

  tmp_fitness    = 50.;

  for  (i=0;i<numfield    s/2;i++) {
    tmp_fitness    -= (0.01*(abs(fiel    ds[i] -
                                  fields[numfield    s - i - 1])
                              + 5.0*(float)abs(    field s[i] - i)));
  }
  critter->init_f    itness = tmp_fitness;
}
```

Az `apdemoi4.in` input file-t használva észrevehetjük, hogy az egyes mezők mérete nem egyezik meg (hiszen `alpha_size` < 2):

```
alpha_size=1
numfields=20
```

Ha a mezők mérete (vagyis a velük ábrázolható maximális szám) nem egyezik meg, akkor a mezők méreteit nekünk kell beállítanunk a forrásfile-ban:

```
void  app_set_field_sizes()
{
  field_sizes[0]     = 10;
  ...
  field_sizes[4]     = 10;
  field_sizes[5]     = 20;
  ...
  field_sizes[14]     = 20;
  field_sizes[15]     = 10;
  ...
  field_sizes[19]     = 10;
}
```

A függvény csak akkor hívódik meg, ha `alpha_size` < 2.

5.9. Megszakított futás folytatása

Komolyabb feladat megoldásakor előfordulhat, hogy az algoritmus olyan hosszú ideig fut, melyet nem tudunk kivárni. Ebben az esetben sokat segíthet, ha meg tudjuk szakítani az algoritmus futását, és később ugyanabból az állásból folytatni tudjuk. A GALOPPS csomag támogatja a futás megszakítását, és további lehetőségeket is nyújt.

Előre megadott generációváltás után mindig lementi az algoritmus állását, így ha a program valamilyen okból le is áll, a számításnak csak kis része veszik el.

Ha az algoritmus tartalmaz olyan változókat, melyeket szintén el kell mentenünk ahhoz, hogy a futtatást folytatni tudjuk, akkor az `app_write_ckp_hdr` függvényben írhatjuk ki ezeket a paramétereket a file-ba. A paraméterek beolvasása is a mi feladatunk ebben az esetben, az `app_read_ckp_hdr` függvényt kell módosítanunk. A két függvény használatára az utazóügynök probléma megoldása során láthatunk példát.

További lehetőség, hogy ha a futtatás újraindításakor módosíthatjuk a GA paramétereit. Nemcsak a legegyszerűbb paramétereket, mint például a keresztezési ráta, hanem a populáció méretét, sőt a kromoszóma mezőinek számát is. Ez lehetőséget ad arra, hogy a GA paramétereit futás közben az algoritmus addigi eredményeit figyelembe véve módosítsuk.

A. Függelék

Köszönetnyilvánítás

Köszönet illeti az elkészítésben nyújtott segítségért Szabó Richárdot, aki hasznos megjegyzéseivel járult hozzá az írás elkészüléséhez.
Köszönet továbbá a hibajegyzékért Sass Bálintnak és Ézsiás Béla Gábornak.

Irodalomjegyzék

[AA91] D. Abramson and J. Abela. A parallel genetic algorithm for solving the school timetabling problem. Technical report, Division of Information Technology, Commonwealth Scientific and Industrial Scientific Organisation (CSIRO), Australia, April 1991.

[Asz96] Aszalós László. Az animat "születése". *Új Alaplap*, 2:28 – 30, 1996.

[Bäc91] T. Bäck. Self-adaptation in genetic algorithms. In F. J. Varela and P. Bourgine, editors, *Proceedings of the First European Conference on Artificial Life. Toward a Practice of Autonomous Systems*, pages 263–271, Paris, France, 11-13 December 1991. MIT Press, Cambridge, MA.

[BBM93a] David Beasley, David R. Bull, and Ralph R. Martin. An overview of genetic algorithms: Part 1, fundamentals. 15(2):58–69, 1993.

[BBM93b] David Beasley, David R. Bull, and Ralph R. Martin. An overview of genetic algorithms: Part 2, research topics. *University Computing*, 15(4):170–181, 1993.

[BBM93c] David Beasley, David R. Bull, and Ralph R. Martin. A sequential niche technique for multimodal function optimization. *Evolutionary Computation*, 1(2):101–125, 1993.

[Ber89] Berend Mihály. Genetika, 1989.

[CP99] Erick Cantú-Paz. A summary of research on parallel genetic algorithms. Technical Report 95007, Department of General Engineering, University of Illinois at Urbana-Champaign, Urbana, IL, 1999.

[ESKB97] A.E. Eiben, I.G. Sprinkhuizen-Kuyper, and B.A.Thijssen. Competing crossovers in an adaptive ga framework. Technical Report 97-11, 1997.

[Fea99] Futó et. al. *Mesterséges Intelligencia*. Aula Kiadó, 1999.

[FSJP97] P. Funes, E. Sklar, H. Juill'e, and J. Pollack. The internet as a virtual ecology: Coevolutionary arms races between human and artificial populations, 1997.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Publishing Company, INC., 1989.

[Gre90] John J. Grefenstette. Genesis (genetic search implementation system) 5.0, 1990.

[HB98] Jörg Heitkötter and David Beasley, editors. *Hitch Hiker's Guide to Evolutionary Computation: A List of Frequently Asked Questions (FAQ)*. 1998. USENET: comp.ai.genetic.

[Hen] Al Hensel. Life 1.06 shareware software.

[Jel96] Jelasity Márk. *The Wave Model of Genetic Algorithms*. Dept. of Applied Informatics József Attila University, Szeged, Hungary, 1996.

[LG83] Mark Wheelis Larry Gonick. *KépreGén*. 1983. Magyar fordítás: Gondolat Könyvkiadó, 1988.

[Mic92] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer-Verlag, 1992.

[Mit] Tom M. Mitchell. *Machine Learning*.

[Moh96] Mohay Jolán. *Genetikai kislexikon*. Natura, 1996.

[Sig93] Karl Sigmund. *Az élet játékai*. Oxford University Press, 1993. Magyar fordítás: Akadémiai Kiadó, Budapest, 1995.

[SV96] Volker Schnecke and Oliver Vornberger. An adaptive parallel genetic algorithm for VLSI-layout optimization. In Hans-Michael Voigt, Werner Ebeling, Ingo Rechenberg, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature – PPSN IV*, pages 859–868, Berlin, 1996. Springer.

[SZ95] Szlávi Péter és Zsakó László. *μlógia 9: Szimulációs modellek a populációbiológiában*. ELTE TTK Általános Számítástudományi Tanszék, 1995.

[Yao95] Xin Yao. A new simulated annealing algorithm. *International Journal of Computer Mathematics*, 56:161–168, 1995.