Paraméterezett modell illesztése a STAGE algoritmus segítségével

Grad László és Mészáros Róbert

programtervező matematikus szak

nappali tagozat

2001. június 8.

Témavezető: dr. habil. Lőrincz András

Tézisek:

- Bevezettük és megvalósítottuk a STAGE algoritmust
- Bemutattuk, hogy a genetikus algoritmus (GA) gyenge mind a STAGE-hez, mind a Stagenis algoritmushoz képest (numerikus kísérletek segítségével)
- Bemutattuk, hogy a Stagenis javíthat a STAGE teljesítményén (numerikus kísérletek segítségével)
- Bemutattuk, hogy a Stagenis algoritmus, amely eleve párhuzamos számítást szimulál, valós párhuzamosság esetén lényegében a párhuzamos szálak számának arányában csökkenti az optimalizálási időt (numerikus kísérletek segítségével)
- Numerikus kísérletek azt mutatják, hogy a STAGE algoritmus többszöri indítása nem párhuzamos gépen is gyorsítani fogja az optimalizációs eljárást.

Tartalom

1. Bevezetés *(Mészáros Róbert)* **1**

2. Optimalizációs algoritmusok ismertetése *(Mészáros Róbert)* **6**

2.1 Adaptív módszerek 6

2.2 Genetikus algoritmusok 8

2.3 Lokális keresések 13

2.3.1 Hegymászó algoritmusok 16

2.3.2 Irányított véletlen séták 17

2.3.3 Szimulált hőkezelés 19

2.4 A STAGE algoritmus 20

2.4.1 A Markov-lánc tulajdonság 21

2.4.2 Lokális keresések Markov-tulajdonságúvá alakítása 21

2.4.3 A STAGE algoritmus működése 22

2.4.4 A függvény-approximátor 28

2.5 A Stagenis algoritmus 32

3. A vizsgálatokhoz készített szoftver *(Grad László)* **34**

3.1 A feladat specifikációja 34

3.2 Megvalósítás 37

3.2.1 Az adatok reprezentálása 38

3.2.2 Felhasznált szoftver-eszközök 40

3.2.3 Megvalósított algoritmusok 42

4. Eredmények *(Grad László)* **45**

5. Diszkusszió *(Mészáros Róbert)* **46**

5.1 A genetikus algoritmus értékelése 46

5.2 Párhuzamosan több optimalizáció indítása 47

5.3 Az állapotjellemzők megválasztása 48

5.4 Összefoglalás 48

6. Jelölések *(Mészáros Róbert)* **49**

Irodalom **52**

(A címek után zárójelben feltüntetett név a címben foglalt rész szerzőjét, készítőjét nevezi meg.)

1. Bevezetés

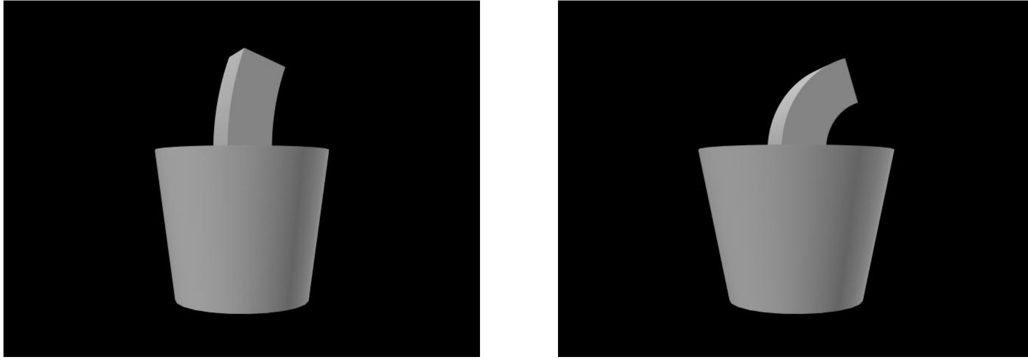
Dolgozatunk alapproblémáját az a kutatás adja, amely a Leuveni Katolikus Egyetem neurobiológus csoportjával együttműködésben zajlik egyetemünkön. E kutatás célja az agy látóidegrendszerének mélyebb megismerése, annak vizsgálata, hogy a szem által érzékelt térbeli tárgyak elhelyezkedését, a tárgyak térbeli egymáshoz való viszonyát hogyan dolgozza fel az agy a látás során.

Ennek egy kézenfekvő módszere a szembe érkező kép által kiváltott neurális jel megfelelő idegrendszeri szinten való vizsgálata. Ezt majomkísérletekkel végzik Leuvenben, az agy inferotemporális cortexének neuronsejtjeit vizsgálják. Ahhoz, hogy jól fel tudják térképezni ezt a területet, megfelelően megválasztott bemenő képekből (képcsoportokból) kell kiindulniuk. Ezért számítógéppel készítik azokat a képeket, amelyeket a majomnak mutatnak, és minden megmutatott képnél megmérik egy, a látóidegrendszerben levő neuron aktivitását. Abból, hogy a neuron milyen képekre reagál és milyenekre nem, következtetéseket lehet levonni. Ezáltal próbálják megismerni, hogy az az agyterület, ami az illető neuront tartalmazza, pontosan milyen funkciót lát el a látás mechanizmusában.

A képeken egyszerű térbeli testek (pl. henger, téglatest) láthatók különféle helyzetekben (1.1. ábra). A kutatók olyan térbeli elrendezéseket próbálnak találni e testek számára, amelyek látványa megdolgoztatja a megfigyelt neuronokat.

Ez a kép-keresés optimalizációs feladattá alakul, ha a képeken szereplő testek térbeli adatait numerikus paraméterekkel írjuk le, és a vizsgált neuron aktivitási szintjét optimalizálandó jelnek tekintjük. Fontos lenne, hogy ezt az optimalizációt számítógép végezze intelligens módon, mert így sokkal gyorsabban, kevesebb próbálkozással, az állatot kevésbé terhelve találhatnának releváns képeket. Dolgozatunkban ennek az optimalizációnak az automatizálási lehetőségeit vizsgáljuk.

Megjegyezzük, hogy ez az automatizáció számos más számítógépes alkalmazásban



1.1. ábra: *A látóidegrendszeri agysejtek stimulációjára használt képek*

is jól használható lenne. Például az arckifejezés-felismerés feladata is megközelíthető oly módon, hogy az emberi arcot felvevő kamera-képet egy háromdimenziós modellből generált számítógépi grafikával megközelítőleg rekonstruáljuk, és amikor a képek egymásra illesztése kellően sikeres, a térbeli modellről leolvassuk a paraméterek értékét. Ilyen módon ugyanis sokkal lényegibb információt kapunk az arckifejezésről, mint amit a képpontok színinformációi eredeti formájukban nyújtanának.

A következőkben a további tárgyalás érdekében mindenekelőtt definiáljuk az optimalizációs problémák alapfogalmait. A fogalmak pontos matematikai jelöléseit összefoglalóan a dolgozat végén, a 6. fejezetben közöljük.

Globális optimalizáció

Az optimalizálási problémákban a cél valamilyen változóhalmaz vagy paraméterhalmaz értékeinek egy olyan konfigurációját (beállítását) megtalálni, amely az összes lehetséges konfiguráció közül a lehető legjobb valamilyen *jósági függvény* (vagy *célfüggvény*) szerint. A lehetséges konfigurációkat *állapotoknak* nevezzük, az összes lehetséges konfiguráció halmazát pedig *keresési térnek.* A jósági függvény értékének kiszámítását valamely konfigurációra az állapot *kiértékelésének* nevezzük. Ennek eredménye egy valós szám: az állapot *jósága.* Ez gyakran mérésből származik, vagy más nemdeterminisztikus módszerrel számítódik, és így bizonytalanságot tartalmazhat: ugyanazon állapot ismételt kiértékelései eltérő jóságokat eredményezhetnek. Emiatt a jósági függvényt az egész keresési téren értelmezett olyan függvénynek tekintjük, amelynek értékei valós értékű valószínűségi változók.

A jósági függvénynek bizonyos esetekben a minimumát (értékét és helyét), más ese-

tekben pedig a maximumát keressük. Így tehát megkülönböztetünk minimalizálási és maximalizálási problémákat. A különbség csak látszólagos, hiszen ha az eredeti jósági függvény helyett az ellentettjét tekintjük, akkor a minimalizálási problémák maximalizálásba mennek át, valamint ugyanez fordítva is fennáll. Ezért bármely módszer, amely maximalizálási problémákat tud megoldani, alkalmazható minimalizálási problémákra is és viszont. A továbbiakban e problémákat egyenértékűeknek tekintjük, és ahol csak lehet, optimalizálási problémaként említjük őket.

Amikor a keresési tér kicsi, a feladat megoldása triviális: mindegyik állapotot kiértékeljük és kiválasztjuk a legjobbat. Ennek a módszernek a neve *teljes bejárás* (az angol irodalomban *exhaustive search*). Általában a keresési tér olyan nagy, hogy ez kivitelezhetetlen. Amikor a probléma rendelkezik valami speciális szerkezettel, amit ki lehet használni, például lineáris program (azaz a jósági függvény értéke az optimalizálandó változók értékeinek valamilyen lineáris kombinációja), akkor hatékony optimumkereső algoritmusok készíthetők, amelyek a keresési tér nagy volta ellenére egzakt megoldást tudnak adni. Vannak azonban olyan feladatok — a kombinatorikus optimalizálás területén számos ilyen problémával találkozhatunk — amikor ez nem lehetséges, például mert a probléma NP-teljes, és valószínűleg egyáltalán nem létezik polinomiális idejű algoritmus a globális optimum megkeresésére. Ilyenkor a célunk csak az lehet, hogy a rendelkezésre álló idő és számítási kapacitás mellett a lehető legjobb megoldást állítsuk elő. A talált megoldásról néha még azt sem lehet eldönteni, hogy milyen messze esik az optimumtól, amikor annak még a jóság-értéke sem ismert.

A globális optimalizálási problémáknak van két fontos jellemzője. Az egyik az, hogy az állapotok jóságát elvileg tetszőleges sorrendben vizsgálhatjuk meg. Ez első ránézésre egészen természetesnek tűnik, mégis fontos tisztázni, mivel az optimalizálási problémák megoldására gyakran olyan módszereket használnak, amelyek korlátozzák saját maguk számára az állapotok kiértékelésének sorrendjét (lásd pl. a szomszédsági struktúra szerepét a lokális keresésekben, 2.3.). Ez a korlátozás azonban az eredeti problémának sosem része. Az a keresési feladat, amelyben az állapotokat *eredendően* nem lehet tetszőleges sorrenben kiértékelni, korlátozhatja a globális jelleget. A másik fontos dolog pedig az, hogy az állapotok jóságát önmagukban is mindig meg lehet vizsgálni. Ez is egy lényeges megszorítás, mert számos olyan probléma van, amelynél ez nem áll fenn, vagy csak nagyon nehezen valósítható meg.

Jelen dolgozatban olyan globális optimalizálási feladattal van dolgunk, melynél az állapotok valós számokból álló paramétervektorok: a képen szereplő testek térbeli elhelyezkedését és kinézetét — szakszóval mondva: a *színteret* — leíró numerikus adatok összessége. A jósági függvény értéke a paramétervektorból számítógéppel generált képre adott válasz, amely származhat mérésből (pl. a majom agyában lévő neuronsejt aktivitásának mértéke a kép láttán), vagy bármi más módon (pl. egy célnak kitűzött képtől való képpontonkénti távolság). Ezeknek konkrét megvalósulásairól a 3. fejezetben fogunk részletesen szólni.

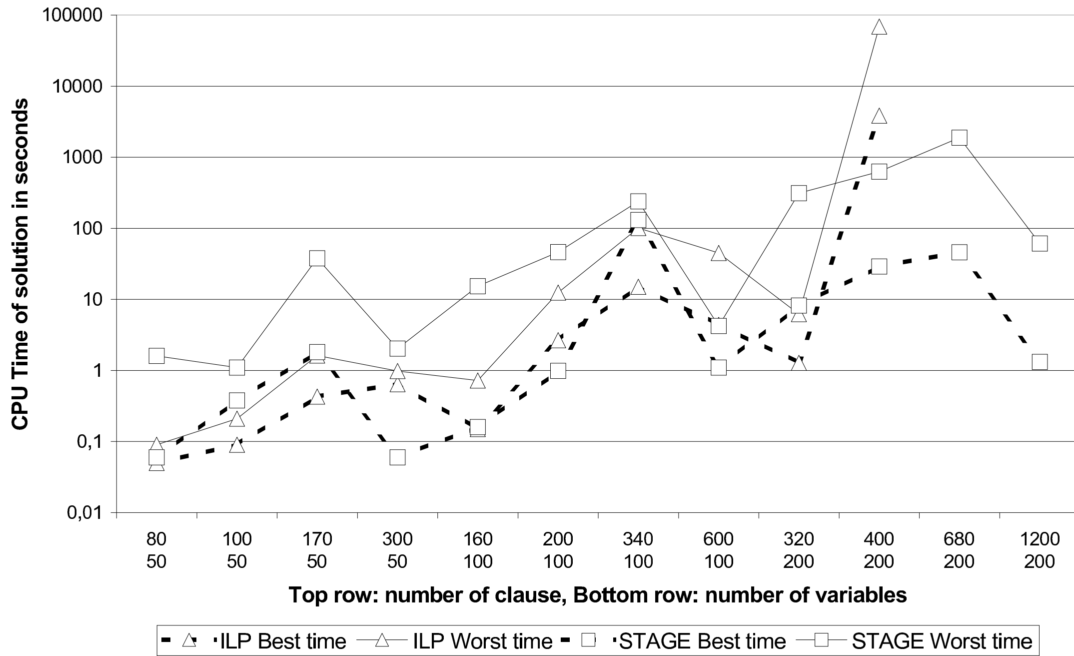
Miért éppen a STAGE algoritmus?

Ahhoz, hogy az optimalizáció számítógéppel való megoldása valódi segítséget nyújtson a majomkísérletekben, annyira gyorsan kell működnie, hogy valós időben (*on-line*) alkalmazhassák. Mivel a majom fárad, valamint agysejtjeinek működése is módosul, ahogy a képek ismerőssé válnak a számára, a számítógépnek maximum 30 perce van arra, hogy legfeljebb egy-két ezer kép kipróbálásával megtalálja a kívánt hatást eredményező színteret. Mivel a keresési tér nagy (6-10 dimenziós), ezt azt jelenti, hogy hatékony optimalizációs módszert kell kidolgoznunk.

A J. A. Boyan által létrehozott STAGE algoritmussal igen figyelemreméltó eredményeket értek el a közelmúltban többek között logikai formulák kielégíthetőségének optimalizációjában. Az 1.2. ábrán a STAGE teljesítménye a lineáris programok megoldására specializált CPLEX kereskedelmi szoftver teljesítményével összehasonlítva látható (forrás: [17]). A CLPEX egészértékű programok (ILP) megoldására specializált algoritmusának futásidejével szemben a STAGE futásidejét (a legjobb és a legrosszabb eseteket) a kielégítendő formula klózai illetve változói számának függvényében ábrázoltuk. A STAGE időigényét a négyzetek, a CPLEX időigényét a háromszögek jelölik. A STAGE algoritmusban függvény-approximátorként kvadratikus regressziót alkalmaztak.

Látható, hogy amint a változók száma megugrik, a STAGE futásideje még nem növekszik a kombinatorikus robbanásnak megfelelően, amikor a CPLEX futásideje már igen (a függőleges skála logaritmikus). Mivel ez a probléma NP-teljes, mindebből arra következtettünk, hogy az előttünk álló sokdimenziós globális optimalizálási feladatban is érdemes a STAGE algoritmusra építkezni.

A dolgozat következő fejezeteiben először röviden áttekintjük a globális optimalizáció



1.2. ábra: *A STAGE algoritmus időigénye a CPLEX kereskedelmi szoftver egészértékű programokat (ILP) megoldó algoritmusával összehasonlítva, logikai formulák kielégíthetőségéről szóló feladatokban.*

problémakörével kapcsolatos irodalmat, majd ismertetjük az általunk felhasznált algoritmusokat és bemutatjuk a vizsgálatainkhoz készített programot. A 4. fejezetben közreadjuk kísérleteink eredményeit, az 5. fejezetben pedig összefoglaljuk tapasztalatainkat. A kutatások közben tovább folytatódnak, mert noha az algoritmus elnyei egyérteműen megmutatkoztak, mégis a jelenlegi tanulóképesség tovább javítandó.

2. Optimalizációs algoritmusok ismertetése

2.1 Adaptív módszerek

A mesterséges intelligencia hagyományos módszereivel általában úgy kezdődik egy probléma megoldása, hogy egy modellel le kell írnunk a valóságnak a megoldás szempontjából lényeges elemeit. Ezt a modellt azután be kell táplálnunk egy számítógépes programba (ez sokféle formában történhet), és a program *ebben a modellben* keresi meg a megoldást (általában valamilyen heurisztikát is használ a keresés gyorsításához). A talált megoldás így a modell fogalmaival lesz leírva, tehát még vissza kell ültetnünk a valóságba.

A gyakorlatban ez az út sokszor nem járható, leginkább azért, mert a modellek a rendelkezésre álló emberi energia, szaktudás és/vagy az idő szűkös volta miatt rendszerint gyengék, és nem elég jól írják le a valóságot. Emiatt a valóságnak a modellbe való leképezése (a modellezés) és talált megoldás visszaültetése (az implementálás) egyaránt jelentős hibák forrásává válhat. Ez gyakran olyan méreteket ölt, hogy a gyakorlatban nem használhatók a modellben egyébként jónak talált megoldások.

Igazából maga az a feladat, hogy jó modellt alkossunk a valóságról, nagyon nehéz: olyan modellt kellene alkotni, ami nem is kezelhetetlenül nagy és bonyolult, ugyanakkor „valósághűen" megőriz minden fontos tulajdonságot a való világból. Sőt, a modellnek még a kereső algoritmusban szereplő heurisztika képességeihez is igazodnia kellene. Valójában ez a munka rendkívül sok tapasztalatot és speciális szaktudást igényel. Emiatt ez a tudás csak keveseknek van birtokában, ettől azonban a hagyományos módszerek alkalmazása nagyon nehézkessé vált és így a fejlődésük is lelassult. Manapság az esetek többségében egyre ritkábbak és nehézkesebbek.

Inkább egy olyan módszerre volna szükség, amelynek nem szükséges modellt megadni az elinduláshoz, vagy legalábbis egy gyenge modellel is megelégszik kezdetben, és máris tud eredményeket produkálni, még ha nem is a legjobbakat. Később, illetve közben, saját tapasztalatai alapján is javulna, fejlődne, átvéve ezzel az embertől a sok tapasztalat megszerzésének hosszadalmas és legtöbbször kevéssé kreatív munkáját.

Az úgynevezett adaptív (alkalmazkodó) megközelítés pontosan ezt valósítja meg. A modellt a heurisztikus programon kívülre helyezi, illetve elhagyja: a program potenciális megoldásokat generál folyamatosan, amelyeket rögtön a valóságban próbál ki, és megméri sikerességüket. A heurisztika feladata ilyenkor az, hogy a próbák mért sikerességeiből kinyerjen valamiféle jól használható tapasztalatot, ami alapján a továbbiakban majd egyre jobb és jobb lehetséges megoldásokat tud javasolni, másképp mondva: egyre jobb döntéseket tud hozni. Manapság a mesterséges intelligenciában ezt tanulásnak nevezzük. Tulajdonképpen arról van szó, hogy az algoritmus információkat gyűjt a rajta kívül lévő világról, ami az ő számára egy „fekete doboz", megpróbálja kiismerni és alkalmazkodni hozzá. Nincs beleépítve semmiféle fix modell a világról, ellentétben a hagyományos módszerekkel, amelyek kimondottan a tudásmérnökök által beléjük táplált modellre építik tudásukat. Egy adaptív algoritmus ha használ is modellt, azt saját maga alakította ki és formálja mindig tovább a megfigyelései alapján. Ezek az algoritmusok tehát a következő ciklust végzik:

(a) a lehetséges megoldások közül véletlenszerűen választunk egyet

(b) átültetjük a valóságba: implementáljuk, kipróbáljuk

(c) mérjük a sikerességét (hogy mennyire jó megoldás ez)

(d) ha még nem elég jó, akkor valamilyen heurisztika szerint választunk egy másik — lehetőleg jobb — lehetséges megoldást, és megismételjük az eljárást a (b) lépéstől

Ez a tanulási folyamat természetesen sok próbálkozást igényel, és különösen az elején még nagyon sok rossz megoldás-javaslatot tesz. Ezért rendszerint nem dobják rögtön a mély vízbe az ilyen programokat, hanem előbb egy szimulált valóságban, egy modellben gyakorlatoztatják őket. A valóság modellezésére tehát változatlanul szükség van, azonban csak addig, amíg az alkalmazott heurisztikát nagyjából behangoljuk és betanítjuk a

megoldandó problémára úgy, hogy azután a valóságban is hatékony legyen majd. Amikor már kellő mennyiségű „tapasztalatot” gyűjtött a modellben gyakorlatozva, akkor elvehetjük fölüle a modellt és a helyére engedhetjük a valóságot. A tanulást nem kell abbahagynia, sőt, a folyamatos tanulás teszi lehetővé azt, hogy a program áthidalja a modell és a valóság közötti — nemritkán jelentős — különbséget, és alkalmazkodjon a számára „új valósághoz”.

Az adaptív módszerek nagy előnye, hogy meglehetősen érzéketlenek az implementálásból és a sikeresség méréséből fakadó hibákra (akár szisztematikus, akár zajszerű hibákról van szó), mivel a döntéseik alapjául szolgáló „tapasztalatokat” sok mérés eredményéből alakítják ki, így a hibák kiátlagolódhatnak. Mindezek miatt az adaptív megközelítés széles körben elterjedt, és olyan heurisztikák születtek általa, mint a genetikus algoritmus (GA), a szimulált hőkezelés (SA) (lásd [1], [2]), a megerősítéses tanulás (RL, lásd [11]), vagy a STAGE algoritmus ([12]).

A fejezet hátralévő részében ezek közül azokat mutatjuk be, amelyeket felhasználtunk a kitűzött feladat vizsgálata során.

2.2 Genetikus algoritmusok

A genetikus algoritmus ötlete a biológiából ered. Az a lényege, hogy folyamatosan „fejben tart” sok lehetséges megoldást, és ezeket egyrészt véletlenszerűen módosítgatja, másrészt a jobbakat kombinálja egymással ahhoz hasonló módon, ahogyan a természet az élővilág egyedeit keresztezi egymással. A gyengébbeket elfelejti, mindig csak a legjobbakat tartja meg az emlékezetében. A múlt tapasztalatainak legjavára építve javasol tehát újabb megoldásokat, amelyek lassan egyre sikeresebbek lesznek, míg csak egy elég jó megoldásra nem talál.

A lehetséges megoldásokat *egyedeknek* nevezzük és véges hosszúságú szimbólumsorozatokkal reprezentáljuk, magukat a szimbólumokat *géneknek* mondjuk. Az egyszerre fejben tartott lehetséges megoldások (egyedek) halmaza a *populáció*, melynek mérete (elemszáma) előre rögzített, nem változtatjuk. Mint minden adaptív algoritmus, a genetikus algoritmus (GA) is iteratív, az előző 2.1. pontban vázolt ciklust végzi:

(a) kezdetben feltölti a populációt véletlenszerűen választott egyedekkel

(b) ezután *dekódolja* az aktuális populáció minden tagját a génekkel való leírásból, egyenként *kiértékeli* őket a megoldandó feladat szempontjából és

(c) megméri sikerességüket, hogy mennyire jó megoldások a problémára, azaz menynyire „életképes egyedek”. A kapott számértéket *fitnessznek* nevezzük, magát a kiértékelő függvényt pedig *fitnessz-függvénynek*.

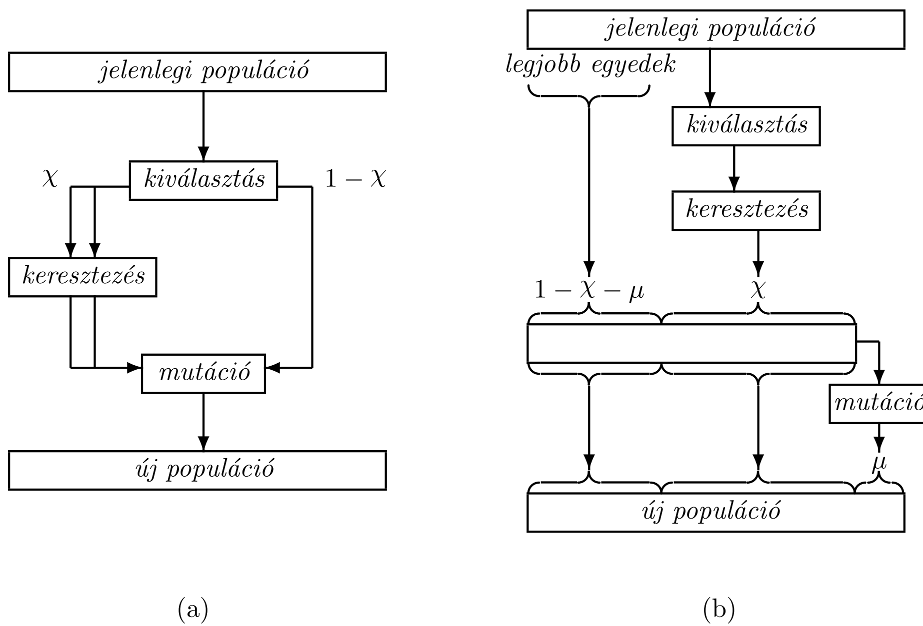
(d) végül egy speciális heurisztika szerint új populációt hoz létre a jelenlegi helyére, és megismétli az eljárást a (b) lépéstől.

A fitnessz-függvény a *jósági függvény* megtestesítője: a genetikus algoritmus ennek a függvénynek keresi a globális optimumát, tehát egy olyan egyedet, amelynek a fitnessze maximális[1]. Az iteráció akkor áll le, amikor a populáció tartalmaz legalább egy „elég jó” egyedet. Az „elég jó” megfogalmazása rendszerint egy előre rögzített küszöbérték bevonásával történik: ennek meghaladása esetén állítjuk le a keresést. Azokban az esetekben, amikor annyira ismeretlen a fitnessz-függvény, hogy még azt sem tudjuk, mekkora az elérhető maximális fitnesszérték, általában időkorlátot szabunk az algoritmus működésére.

Az új populáció kialakítását vezérlő heurisztika úgynevezett *genetikus operátorokat* alkalmazva a meglévő populációból származtatja az újat, amit ezért új *generációnak* is nevezünk. A genetikus operátorok némileg az élővilág működését imitálják, ennek köszönhetik nevüket. Általában a következő három operátort szokták használni: a kiválasztást, a keresztezést és a mutációt.

A *kiválasztás* a populációból kiválaszt egy egyedet véletlenszerűen oly módon, hogy a nagyobb fitnessz-értékű egyedek közül nagyobb valószínűséggel választ. Ennek gondolata abból a biológiai megfigyelésből ered, hogy a populációk életképesebb egyedei nagyobb eséllyel vesznek részt az új generáció nemzésében, mint gyengébb társaik. A *keresztezés* operátor kombinál egymással két kiválasztott egyedet (a génekkel való leírásukat) valamilyen módon, így két új egyedet produkál. Ez az operátor felelős a keresés előrehaladásáért: a múltból fennmaradt jó megoldások tulajdonságait kombinálva vélhetően még jobb megoldásokhoz jutunk. Végül a *mutáció* operátor valamilyen (kicsi)

[1] A genetikus algoritmusoknál általában maximalizálási szemszögből szoktuk nézni a feladatokat. Minimalizálási feladatnál természetesen a legkisebb fitnesszértékű egyedet kell keresni.



2.1. ábra: *A genetikus operátorok alkalmazásának két lehetséges sémája. Az (a) változatnál nagyon ritkán megeshet, hogy a populációban lévő legjobb egyedet elveszítjük. A (b) változat mindig megőrzi a legjobb egyedeket.*

μ valószínűséggel véletlenszerűen módosítja az új egyedekben a géneket. Ennek a degeneráció megelőzésében van szerepe: folyamatosan biztosítja a teljes keresési térből való mintavételezést, ezáltal az algoritmus nem rekedhet meg véglegesen egyik lokális optimumban sem.[2]

A genetikus operátorokat sokféleképpen használhatjuk új populáció származtatására. A 2.1. ábra ennek két lehetséges módját mutatja. Az (a) változatban az új generáció mindegyik egyedét egyetlen véletlenre épülő műveletsorral származtatjuk: választunk egy vagy két egyedet a populációból a kiválasztás operátorral, ezekre vagy alkalmazzuk a keresztezés operátort, vagy nem ($1-\chi$ valószínűséggel nem, ilyenkor az egyed változatlan marad); végül mielőtt az új populációba betennénk, minden egyedre alkalmazzuk a mutáció operátort, hogy génenként μ valószínűséggel véletlenszerűen átírja őket. Ez az igen egyszerű műveletsor a genetikus algoritmus heurisztikájának a mintapéldája.

Ezt a műveletsort azonban másképpen is meg lehet valósítani. A (b) változatban

[2]Lokális optimumok alatt most azokat a populációkat értjük, amelyeknek tagjait bárhogyan keresztezzük, nem kapunk jobb egyedet a benne lévőknél. Az ilyen degenerált populációk mutáció nélkül képtelenek lennének javulni.

a populáció legjobb egyedeit feltétel nélkül mindig átörökítjük az új generációba (egyszerű másolással), és az új generációnak csak a fennmaradó részét származtathatjuk keresztezéssel és mutációval. Ez azért lehet fontos, mert az (a) változat bizonyos eséllyel megengedi a legjobb fitnesszű egyed elveszítését is. Ez nem szerencsés, mert sérülhet miatta az a monotonitás, hogy az új generáció legjobb egyede mindig legalább olyan jó, mint a szülő generáció legjobb egyede volt. Márpedig *ez a monotonitás az alapja annak, hogy a genetikus algoritmus hosszú távon célba talál.*

Ha a legjobb egyedet elveszítjük, akkor a fitnesszértékben visszaesés történik. Ha azonban kiszámoljuk, hogy az (a) változatnál mekkora eséllyel következhet ez be, akkor azt látjuk, hogy csak kis elemszámú populációk esetén okoz problémát. Jelöljük egy pillanatra q_i-vel annak a valószínűségét, hogy a populáció egyik i egyede kiválasztódik és változtatás nélkül át is kerül az új generációba. Ehhez ugyebár az kell, hogy a keresztezés önmagával keresztezze őt, vagy ne legyen keresztezés egyáltalán, valamint egyik génjén se történjen mutáció: $q_i = p_i(\mathcal{X} p_i + (1 - \mathcal{X}))(1 - \mu)^{|i|}$. Ezt felhasználva annak az esélye, hogy ez egyszer sem történik meg egy $|\mathcal{P}|$ elemű populációban, vagyis elveszítjük az i egyedet, $(1 - q_i)^{|\mathcal{P}|}$. Ez a valószínűség a legnagyobb q_i érték (a legjobb egyed) esetén a legkisebb, és $|\mathcal{P}| \to \infty$ esetén 0-hoz közelít. Nagy populációkban tehát a legjobb egyed elveszítésének az esélye nagyon kicsi, nem okoz problémát, sőt, néha nem is árt, például amikor a fitnesszértékek mérése tudnivalóan nagyon zajos.

Kis elemszámú populáció esetén a (b) változat mintájára érdemes megvalósítani a genetikus algoritmus heurisztikáját. A keresztezések száma itt nem függ a véletlentől: az új generációnak mindig pontosan $\mathcal{X}$ hányada keletkezik keresztezéssel. Természetesen a kiválasztás operátor adja a szülő egyedeket a keresztezésekhez. A mutáció operátor mindezek eredményéből válogat teljesen véletlenszerűen (nem kiválasztással) néhány egyedet, és egy-egy gént véletlen értékre átír bennük. Nem rontja el a kiválasztott egyedeket, hanem újakat produkál ezzel a módszerrel: az új generáció μ hányadát állítja elő.

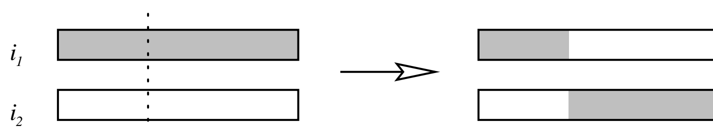
A genetikus algoritmusok sokféleségét tovább szaporítják az eddig nem említett részletek megvalósítási lehetőségei. Jelen dolgozatban az ún. kanonikus genetikus algoritmusból (CGA) indultunk ki, amelynek operátorai egész pontosan az alábbi szabályok szerint működnek:

– a kiválasztás operátor az egyedek (nemnegatív) fitnesszének relatív arányában

osztja le a kiválasztási valószínűségeket. Az i egyed kiválasztásának esélye így

$$p_i = \frac{f(i)}{\sum_{j \in \mathcal{P}} f(j)}$$

– a keresztezés operátor úgynevezett *egypontos keresztezést* valósít meg: a szülő egyedeket leíró génsorozatok egy bizonyos szakaszát cseréli fel egymással (lásd 2.2. ábra). A cserélendő szakasz kezdőpontját (az ábrán szaggatott vonal jelöli) véletlenszerűen sorsolja ki.



2.2. ábra: *Az egypontos keresztezés működése*

A genetikus algoritmusok általános célú globális optimalizációs módszerek, bármilyen globális optimalizálási feladatra alkalmazhatók. Bizonyított, hogy a kanonikus genetikus algoritmus bármely problémának 1 valószínűséggel megtalálja az optimális megoldását ([3]). Mivel azonban sztochasztikus folyamatról van szó, amely nem mindig konvergens, az említett tétel azt is jelenti, hogy végtelenül kis valószínűséggel ugyan, de előfordulhat olyan eset, amikor az algoritmus egyáltalán nem konvergál az optimum felé. Emiatt a gyakorlatban mindig olyan terminálási feltételt kell használni, amely valamely maximális lépésszám után mindenképpen leállítja a ciklust, még akkor is, ha a populáció még nem érte el a kívánt fitnessz-szintet.

Az irodalom szerint a genetikus algoritmusok kiemelkedően jó hiba- és zajtűrő képességgel rendelkeznek. Ezt az teszi lehetővé, hogy nem bízzák rá magukat egyetlen jónak talált megoldásra — szemben például a lokális keresésekkel —, hanem számos lehetséges megoldásra emlékeznek, és folyamatosan kombinálják egymással őket. Mindenféle zaj és hiba természetesen jobban kiátlagolódik a sok mérés következtében, és ezért kevésbé zavarja össze az algoritmust.

Ennek a sok számolásnak természetesen ára van, mégpedig az idő: a genetikus algoritmus nem tartozik a leggyorsabbak közé. A lokális keresések általában lényegesen kevesebbet számolnak, és gyakran gyorsabban célba is juttatnak. Ezért kísérleteinkben egy olyan genetikus algoritmussal dolgoztunk, amelyben a fitnessz-függvény értékének

kiszámítását lokális kereséssel végeztük. A lokális keresés a jósági függvényt optimalizálja, abból az állapotból kiindulva, amelyet a kiértékelendő egyed jelent (lásd a 6. fejezetben). Lokális keresésként a mohó hegymászó algoritmust használtuk. Az így kapott algoritmusnak a *GA with RR* nevet adtuk.

2.3 Lokális keresések

A globális optimalizálási feladatok megoldásakor a cél az, hogy egy olyan állapotot találjunk, amelyen a jósági függvény maximalizálási problémánál nagyobb vagy egyenlő, minimalizálási problémánál kisebb vagy egyenlő értéket vesz fel, mint az összes többi állapoton. Ezen alaptulajdonsága alapján azonban gyakorlatilag lehetetlen megtalálni az optimális állapotot, mert annak ellenőrzéséhez, hogy egy állapot rendelkezik-e ezzel a tulajdonsággal vagy sem, össze kellene hasonlítani az ő jóságát a keresési tér ($\mathcal{X}$) összes többi állapotának jóságával, és ez, amint a bevezetőben már mondottuk, általában kivitelezhetetlen. Jó volna tehát az optimumnak további tulajdonságait is megfogalmazni, olyanokat, amelyek alapján könnyebben megtalálható, mert könnyebben ellenőrizhetőek, mint a fenti.

Az alaptulajdonság egyszerű gyengítéséből adódik az, hogy az optimális állapot olyan, amelyen a jósági függvény jobb értéket vesz fel, mint a környező szomszédos állapotokon. Az ennek eleget tevő állapotokat *lokálisan optimális állapotoknak* nevezzük. Sajnos ezek közül általában csak kevés optimális globálisan is, különösen olyankor, amikor a jóság mérése zajos. Mégis a lokális optimalitásnak ez a szükséges, de korántsem elégséges tulajdonsága az alapja a mostanában leggyakrabban alkalmazott technikának, a globális optimalizálási feladatok *lokális kereséssel* való megoldásának.

A lokális keresések olyan optimalizációs módszerek, amelyek egy adott kezdőállapotból elindulva lokálisan optimális állapotot keresnek, mégpedig lehetőleg minél jobbat. Vagyis olyan állapotot, amelyen a jósági függvény jobb értéket vesz fel, mint a környező *szomszédos állapotokon.* De melyik állapotok szomszédosak egymással?

Ennek megválaszolására a lokális keresések mindig odaképzelnek egy *szomszédsági struktúrát* a keresési tér állapotai közé. Ez egy $N : \mathcal{X} \to 2^{\mathcal{X}}$ függvény alakjában formalizálható, amely mindegyik állapotra megadja a vele szomszédos állapotok halmazát. Ez tulajdonképpen egy irányítatlan gráfot határoz meg az állapotok fölött: azon állapo-

tok vannak összekötve éllel, amelyek szomszédosak egymással. Ez a struktúra a lokális keresések egyik bemenete, amit nekünk kell megalkotnunk, mielőtt a lokális keresést alkalmazni kezdenénk.

Gyakran a szomszédos állapotokat perturbációval (a változók értékeinek kis léptékű módosításával) állítják elő, különösen folytonos keresési tér esetén, amikor az állapotok leírása valós értékű változókból tevődik össze. Ilyenkor ugyanis ez a legtermészetesebben adódó szomszédsági struktúra, és még a jósági függvény értékének a szomszédos állapotokban való közelítését is segíti, ha az aktuális állapotbeli függvény- és gradiens-értéket már ismerjük.

Amikor a szomszédos állapotok bizonyos előre rögzített irányokban való elmozdulásokból adódnak, a szomszédsági struktúrát szokás *mozgási operátorokkal* megadni. Ezek $\mathcal{X} \to \mathcal{X}$ típusú függvények, amelyek mindegyike egy-egy elmozdulási iránynak felel meg: azt az állapotot adja meg, amibe akkor kerülünk, hogyha az aktuális állapotból ebbe az irányba mozdulunk el. Az elmozdulás helyett szokás *lépést* is mondani. Mozgási operátorok használata esetén a szomszédos állapotok halmaza az aktuális állapotban rendelkezésre álló mozgási irányokba való lépések eredményeinek összessége (a pontos képlet a 6. fejezetben található). Például ha a kétdimenziós sík pontjaiból áll a keresési tér, akkor *fel, le, jobbra* és *balra* mozgási operátorok lehetnek. Ha valamely $x \in \mathcal{X}$ állapotban csak fel és jobbra lehet menni, balra és le nem, akkor ott $N(x) = \{fel(x),\ jobbra(x)\}$. A továbbiakban feltesszük, hogy a szomszédsági struktúra mindig mozgási operátorokkal van megadva.

A szomszédsági struktúra és a jósági függvény együttesen kialakítják az úgynevezett *költségfelszínt* a keresési tér felett. A lokális keresések rendszerint e felszínen mozognak (a mozgási operátorok által) és próbálnak egy minél jobb lokális optimumba eljutni. Valójában nem kellene, hogy ragaszkodjanak a mozgási operátorokhoz, hiszen a feladat megengedi, hogy bármikor tetszőleges állapotokat vizsgáljanak meg. Mégis, a legtöbb lokális keresési módszer csak a mozgási operátorok használatával kér be új, eddig még nem vizsgált állapotot.

Az ilyen lokális keresések állapotról állapotra lépdelnek, mondhatni sétákat járnak be a költségfelszínen, a keresési térben. Séta alatt egy olyan állapotsorozatot értünk, amelynek minden újabb tagja valamely megelőzőnek a szomszédja. Nem feltétlenül a legutóbbinak, hanem valamely megelőzőnek, mert visszaugrás is lehetséges, amennyiben

a keresés emlékszik bizonyos régebben érintett állapotokra is. A séta során megkülönböztetjük azt, amikor csak kiértékelünk egy szomszédos állapotot, attól, amikor át is lépünk oda. Akkor mondjuk, hogy *átlépünk*, amikor arra az állapotra is alkalmazzuk valamelyik mozgási operátort, és bekérjük, kiértékeljük az ő egyik szomszédját is.

Ha a sétából mint állapotsorozatból elhagyjuk azokat az állapotokat, amelyeket csak kiértékeltünk, de nem léptünk át oda, akkor egy rövidebb állapotsorozatot kapunk, ezt nevezzük a keresés *pályagörbéjének*. A pályagörbe tehát a keresés során érintett állapotokat az odalépés sorrendjében sorolja fel (amelyik állapotba nem léptünk át, azt nem tartalmazza), a továbbiakban ez lesz számunkra a keresés *kimenetele*.

Természetesen mindig van egy kiinduló állapot, ahonnan az egész keresés elindult. Már mondtuk, hogy ezt a *kezdőállapotot* a lokális keresések bemenő paraméterként kapják, vagyis nekünk kell meghatároznunk még a keresés megkezdése előtt. Megválasztása rendszerint nagy mértékben befolyásolja a keresés hatásfokát. Véges ideig való keresgélés után a keresés nyilván megállapodik valahol (csak ilyen módszerekkel foglalkozunk), ezt a helyet nevezzük *végállapotnak*, a jósági függvény ottani értékét pedig a lokális keresés *eredményének*. A továbbiakban feltételezzük, hogy a végállapot mindig a keresés során érintett állapotok legjobbika.

Mindezek alapján formálisan is definiáljuk a lokális keresés fogalmát.[3] A lokális keresési módszereket $\mathcal{V} \times \mathcal{X} \to \tilde{\mathcal{X}}^*$ típusú függvényeknek tekintjük és általában π-vel jelöljük. $\forall o \in \mathcal{V}$ jósági függvény és $\forall x \in \mathcal{X}$ kezdőállapot esetén $\pi(o, x)$ egy olyan valószínűségi változó, ami $\mathcal{X}$-beli állapotok sorozatai — a lehetséges pályagörbék — közül vesz fel értéket. Azonban $\pi(o, x)$-et tekinthetjük $\tilde{\mathcal{X}}$-beli valószínűségi változók sorozatának is: $\xi_i^{\pi(o,x)}$-vel jelöljük a $\pi(o, x)$ állapotsorozat i-edik tagjának megfelelő valószínűségi változót ($i = 1, 2, \ldots$). $\xi_i^{\pi(o,x)}$ tehát azt adja meg, hogy az o-t optimalizáló, x-ből indított π lokális keresési módszer az i-edik lépés megtételekor hol tart, melyik állapotból lép tovább. Természetesen $\xi_1^{\pi(o,x)}$ értéke mindig x, és állapodjunk meg abban, hogy a keresés leállását e ξ-változók szempontjából úgy értelmezzük, hogy minden további „lépés" előtt a keresés változatlanul a végállapotnál tart.

$\pi(o, x)$ értéke minden megfigyeléskor egy konkrét pályagörbe lesz, amelyet τ-val jelölünk és így írjuk: $\tau = \pi(o, x)$. Nyilván $\tau = x_1 x_2 \ldots x_{|\tau|} \in \mathcal{X}^*$. Ezek a pályagörbék mindig olyan állapotsorozatok, amelyekre teljesülnek az alábbiak:

[3] Lásd még a jelöléseket a 6. fejezetben.

- $x_1 = x$
- ha a keresés i-edik lépésében a π módszer az $m_i \in \mathcal{M}$ mozgási operátor alkalmazásával kapott állapotba lépett át, akkor $x_i \in D_{m_i}$ és $x_{i+1} = m_i(x_i)$
  $(i = 1, 2, \ldots, |\tau| - 1)$

A továbbiakban áttekintjük a leggyakrabban használt lokális kereső módszereket.

2.3.1 Hegymászó algoritmusok

Ez talán a legegyszerűbb lokális keresési módszer. Legfontosabb tulajdonsága az, hogy mindig csak szigorúan jobb állapotba hajlandó átlépni, a jósági függvény értékének romlását nem engedi meg.

Számos variációja van ennek az algoritmusnak. A *mohó* lokális keresés például kiértékeli az összes szomszédos állapotot, és mindig a legeslegjobb állapotba lép tovább. (Amikor ez nem volna egyértelmű, akkor véletlenszerűen választ a legjobbak közül.) Ez a módszer az útjába eső első lokálisan optimális állapotban megreked. Szigorú monotonitása miatt garantáltan nem kerül végtelen ciklusba; véges keresési terekben ez a leállását is garantálja.

Egy másik változata a *sztochasztikus hegymászó* algoritmus (vagy „legelső javító lépés módszere"), amely véletlen sorrendben értékeli ki a szomszédos állapotokat, és az első olyanba átlép, amelyik javít valamit a jósági függvény értékén. Ez a keresés akkor áll le, ha bizonyos számú kiértékelés után még mindig nem talált a jelenleginél jobb szomszédos állapotot, amibe átléphetne. Ez a kiértékelés-szám, a *tűrés*, egy konstans bemenő paraméter. Ennél az algoritmusnál nyilván sosem lehetünk biztosak abban, hogy lokális optimumban állt meg, csak valószínűsíthetjük, hogy igen. Ennek ellenére gyakran alkalmazzák, mivel más kereső módszerekhez képest kevés állapot-kiértékelést végez, ugyanakkor jó megoldásokat produkál. Ez is szigorúan monoton módszer, következésképp véges terekben garantáltan terminál.

A sztochasztikus hegymászó algoritmusoknak van egy olyan változata is, amely megengedi az átlépést azokba a szomszédokba is, amelyek bár nem rontanak, de nem is javítanak, vagyis nem változtatnak a jósági függvény értékén. Bizonyos feladatoknál ezek a „nem változtató lépést is elfogadó" sztochasztikus hegymászó algoritmusok lényegesen jobban teljesítenek, mint a fenti, „nem változtató lépést visszautasító" társaik.

Mivel azonban ezek már nem szigorúan monoton keresések, ügyelni kell arra is, hogy a költségfelszín valamely lapos részletén ki ne alakuljon végtelen ide-oda ingázás, vagy körbe-körbe járás. Ennek orvoslására következő pontban (2.3.2) ismertetünk néhány lehetőséget.

2.3.2 Irányított véletlen séták

Valójában a hegymászó algoritmusok is ebbe a kategóriába tartoznak. Irányított véletlen séták mindazok a módszerek, amelyeknél az állapotátmenetek feledékenyek, nincs emlékezőképességük: nem számlálnak semmi olyat, ami ne nullázódna állapotváltáskor, és nem jegyeznek meg korábban érintett állapotokat sem. A 2.3.1. pontban tárgyalt hegymászó algoritmusokon kívül ide tartoznak még a „legjobb szomszéd"-módszerek, a GSAT és a WALKSAT is.

A „legjobb szomszéd"-módszerek (az angol irodalomban *force-best-move*) a mohó kereséshez hasonlítanak, azzal a különbséggel, hogy akkor is továbblépnek a legjobb szomszédos állapotba, amikor az valójában rosszabb, mint a jelenlegi. Ennek egyik sikeres alkalmazása a GSAT algoritmus ([8]).

A WALKSAT algoritmus nagy logikai formulák kielégítésével (angolul: *SATisfiability*) foglalkozó optimalizálási problémákra kifejlesztett kereső módszer. Minden lépése azzal kezdődik, hogy a lehetséges elmozdulási irányokból véletlenszerűen kiválaszt egy kisebb csoportot (konkrétan: a megoldandó CNF-ből kiválaszt egy még kielégítetlen klózt), és megvizsgálja, hogy ezek közül melyik irány mennyit változtatna a célfüggvényen, ha arra lépne tovább. Ha vannak olyan irányok, amelyek javítanak a célfüggvény értékén, akkor a legtöbbet javító irányt kiválasztja (az iránynak megfelelő logikai változó értékét átbillenti). Ha egyik irányba sem javul a célfüggvény, akkor $1 - zaj$ valószínűséggel kiválasztja a legkevesebbet rontó irányt, zaj valószínűséggel pedig véletlenszerűen akármelyik irányt a csoportból. A zaj paraméter optimális beállítása problémafüggő ([10]).

Az irányított véletlen séták eredeti formájukban általában nem végesek, nem állnak le. Ahhoz, hogy terminálásukat biztosítsuk, valamilyen leállási feltételt kell bevezetni. Az alábbiakban erre adunk néhány javaslatot, egyúttal megemlítve azt is, hogy a STAGE algoritmus szempontjából fontos Markov-tulajdonsággal (lásd a 21. oldalon) mennyire

egyeztethetők össze.

i. *Rögzített számú lépés után leállítjuk az algoritmust*

Ez a megoldás nem egyeztethető össze a Markov-tulajdonsággal. Ugyanis az előttünk álló pályagörbe-hossz a megtett lépések számlálása következtében már nemcsak az aktuális állapottól fog függeni, hanem a globális lépésszám számlálótól is, tehát „a múlttól". Az így kapott keresés Markovivá tételére még a 2.4.2. pontban ismertetett átalakítás sem jelent valódi megoldást, mivel az ilyen keresés semmilyen körülmények között sem nullázza a globális lépésszám számlálót.

ii. *A terminálás véletlenszerű, minden lépésben $\epsilon > 0$ eséllyel*

Ennek a megoldásnak bizonyos feladatokban nagy hátránya lehet, hogy olykor félbeszakít ígéretesen induló keresési folyamatokat is, és így értékes információkat veszíthetünk el. Viszont megőrzi az irányított véletlen séták eredendő Markov-tulajdonságát.

iii. *Leállás akkor, ha egymást követő „tűrés" számú lépésben nincs javító állapot*

Bevezetünk egy számlálót a jósági függvény javítására tett próbálkozások számolására, és akkor állítjuk le a keresést, amikor egymás utáni *tűrés* próbálkozás eredménytelen. Hogy mit tekintünk egy próbálkozásnak, az igazából algoritmustól függ, legtöbbször egy újabb szomszédos állapot kiértékelését szokták próbálkozásnak venni. Amikor a próbálkozás sikeres, vagyis olyan állapotátmenetet hajt végre a keresés, amely a jósági függvény értékét javítja, akkor ezt a számlálót mindig lenullázzuk.

Az így kapott algoritmus természetesen csak akkor marad Markov-tulajdonságú, hogyha a számláló nullázása minden állapotátmenetnél megtörténik, különben ugyanis a keresés megelőző szakaszából származó vezérlési információvá válna. Ezzel a megoldással tehát csak azok a módszerek maradnak Markov-tulajdonságúak, amelyek egyébként szigorúan monotonak, vagyis csak javító lépéseket engednek meg.

Ha olyan módszerre alkalmazzuk ezt a megoldást, ami nem szigorúan monoton (például a sztochasztikus hegymászónak a nem változtató lépést is elfogadó változata), akkor a kapott algoritmust a 2.4.2. pontban ismertetett átalakítással tehetjük is-

mét Markov-tulajdonságúvá. Minden alkalommal ugyanis, amikor sikerül javító lépést tennie az algoritmusnak és ezért lenullázza a *tűrés*-számlálót, tulajdonképpen alaphelyzetbe kerül: ebből a jobb állapotból „tiszta lappal indul" tovább, elölről kezdi a keresést. A csak ilyen állapotokból álló pályagörbékre tehát fennáll a Markov-lánc tulajdonság.

2.3.3 Szimulált hőkezelés

A szimulált hőkezelés (*simulated annealing*, SA) fizikai indíttatású lokális kereső algoritmus. Tudjuk, hogy ha egy fémolvadékot lassan hűtenek le, akkor az kristályos szerkezetet vesz fel. A kialakult szerkezet a rendkívül nagy számú lehetséges elrendeződés közül a lehető legalacsonyabb energiájú lesz. Ha azonban gyorsan hűtjük az olvadékot, akkor amorf szerkezet alakul ki, ami egy lokális energiaminimumnak felel meg. Ez a megfigyelés az alapja a szimulált hőkezelés módszerének.

Az eljárás működése a sztochasztikus hegymászó algoritmushoz hasonlít, azzal a különbséggel, hogy a lokálisan optimális állapotokba való „befagyást" elkerülendő, megengedi a jósági függvény értékét rontó elmozdulásokat is, azonban időben egyre csökkenő valószínűséggel. Tehát a sztochasztikus hegymászóhoz hasonlóan a szomszédos állapotokat véletlen sorrendben értékeli ki, és az első javító (pontosabban nem rontó) állapotot elfogadja. A rontó állapotokkal azonban másként cselekszik. A Boltzmann-statisztika szerint annak valószínűsége, hogy egy rendszer T hőmérsékleten E energiájú állapotban legyen, $\mathrm{P}(E) \approx e^{\frac{E}{k \cdot T}}$. Ennek mintájára az algoritmus valamely $x \in \mathcal{X}$ állapotból a szomszédos, x-nél rosszabb $x' \in N(x)$ állapotba $e^{-\frac{|o(x')-o(x)|}{t_i}}$ valószínűséggel enged meg átlépést, ahol $t_i > 0$ a *hőmérséklet* paraméter értéke a keresés i-edik lépésében. Rontó állapotba tehát annál kisebb valószínűséggel lép át, mennél kisebb a hőmérséklet és mennél nagyobb rontást jelentene az átlépés. A *hőmérséklet* értékét úgy kell meghatározni, hogy szép lassú „hűtést" szimuláljon a keresés során. Például a

$$t_i := \begin{cases} 2.0 - i/1000 & \text{ha } i < 1000, \\ 0.99^{(i-1000)} & \text{ha } i \geq 1000 \end{cases}$$

beállítás kezdetben lineáris, majd exponenciális gyorsasággal csökkenti a hőmérsékletet. Az ideális hűtési stratégia feladatfüggő, de bizonyított, hogy kellően lassú lehűtés esetén ez a módszer 1 valószínűséggel megtalálja a globális optimumot.

A szimulált hőkezelés határértékben a nem változtató lépést is elfogadó sztochasztikus hegymászóval válik egyenértékűvé. Leállási feltételként az állapot-kiértékelések számát szokták figyelni: amikor ez elér egy előre beállított maximumot, akkor a keresést leállítják.

2.4 A STAGE algoritmus

Amikor a lokális keresés olyan lokálisan optimális állapotban áll meg, amely nem optimális globálisan is, annak nem örülünk. Az előzőekben tárgyalt módszerek különféle ötletes módokon próbálják áthidalni ezt a problémát. Ezek az ötletek egyes feladatok megoldásában rendszerint látványos sikereket érnek el, más problématerületekre alkalmazva azonban gyakran gyengén teljesítenek. Ilyenkor a lokális keresés alkalmazói általában azzal kezdenek próbálkozni, hogy a jósági függvény értéke mellé az állapotok különféle számszerűen megfogalmazható tulajdonságait társítják, és ezekkel „büntetik" illetve „jutalmazzák" az egyes állapotok jóságát, így próbálják a keresés lépéseit az optimum feltételezett irányába igazítani.

Milyen tulajdonságokról van itt szó? Olyan, a keresési tér egészén értelmezett számértékű állapotjellemzőkről, amelyeknek az értéke minden állapotban azt próbálja jellemezni, jelezni, hogy érdemes-e oda átlépnie a keresésnek, onnan folytatva megtalálja-e majd a globális optimumot vagy sem. A gyakorlat azt mutatja, hogy ez az ügyeskedés sokszor megéri és jelentősen feljavítja a kereső módszer hatékonyságát. Azonban fáradságos azt meghatározni, hogy milyen állapotjellemzőkkel próbálkozzunk és azokat hogyan, milyen kombinációban, milyen súlyokkal építsük össze az eredeti jósági függvénnyel ahhoz, hogy a kapott összetett jósági függvény eredményesen segítse a keresést.

Justin Andrew Boyan, a STAGE algoritmus készítője azt a kérdést tette fel, hogy lehetne-e automatizálni ezt a folyamatot, vagyis olyan algoritmust készíteni, amely a rendelkezésére bocsátott számértékű állapotjellemzők közül automatikusan használatba veszi azokat, amelyek tényleg segíthetik a lokális keresést, és kialakítja az összetett jósági függvényt is? A szóban forgó algoritmussal ezt sikerült megvalósítania.

A STAGE algoritmus tehát egy lokális keresési módszerre épül rá, annak hatékonyságát javítja fel. Nem közömbös azonban, hogy milyen ez a lokális keresés, amelyre a STAGE algoritmust ráépítjük. Bizonyos lokális keresési módszerekkel jobban tud együtt-

működni, mint másokkal. Pontosabban: csak a Markov-tulajdonságú lokális keresések esetén igazolt a STAGE algoritmus működése matematikailag is — a tapasztalatok azonban azt mutatják, hogy nem csak ilyen keresésekkel érdemes alkalmazni.

2.4.1 A Markov-lánc tulajdonság

Formálisan egy π lokális keresési módszer akkor Markov-tulajdonságú (Markov-láncként működő), ha $\forall o \in \mathcal{V}$, $\forall x \in \mathcal{X}$ mellett a keresés bármely lehetséges $\tau = \pi(o, x)$ kimenetelére fennáll a

$$\mathrm{P}(\xi_i^{\pi(o,x)} = x_i \mid \xi_1^{\pi(o,x)} = x_1, \xi_2^{\pi(o,x)} = x_2, \ldots, \xi_{i-1}^{\pi(o,x)} = x_{i-1}) = \mathrm{P}(\xi_i^{\pi(o,x)} = x_i \mid \xi_{i-1}^{\pi(o,x)} = x_{i-1})$$

$$(i = 2, 3, \ldots |\tau|, \quad x_1 := x, \quad x_i \in \mathcal{X})$$

Markov-lánc tulajdonság.

A lényeg az, hogy az x-ből induló $\tau = x_1 x_2 \ldots x_i x_{i+1} \ldots x_{|\tau|}$ pályagörbe bármelyik x_i-t követő befejező szakasza pontosan ugyanolyan valószínűséggel álljon elő, mintha x_i-ből indult volna el a keresés. Vagyis a Markov-tulajdonságú módszereknél gyakorlatilag a pályagörbék közbülső pontjai is kezdőpontok. A fenti Markov-lánc tulajdonság azt mondja ki, hogy ez akkor teljesülhet, ha a séta során annak az állapotnak a megválasztása, amelyikbe átléptünk, sosem függ a korábban megvizsgált állapotoktól (még azok számától sem), hanem csak a pillanatnyi helyzettől.

2.4.2 Lokális keresések Markov-tulajdonságúvá alakítása

Az alábbiakban ismertetünk egy olyan átalakítást, amellyel elvileg bármilyen lokális keresési módszer Markov-tulajdonságúvá tehető. A gyakorlatban sajnos nem minden esetben jelent valódi segítséget.

Az ötlet az, hogy a keresési módszer kimenetére egy szűrőt helyezünk, amely a keresés által generált pályagörbék állapotait megválogatva, az eredeti pályagörbe helyett annak csak egy olyan részsorozatát engedi át, amelyre teljesül a Markov-lánc tulajdonság.

Formailag egy új, π_1 lokális keresési módszert hozunk létre, amely magában foglalja az eredeti π módszert. A π_1 működése az, hogy pontosan akkor lép (mindig abba az állapotba, amibe π), amikor

a) π olyan állapotátmenetet hajt végre, aminek során alaphelyzetbe kerül, mintha most kezdené a keresést (pl. nullázza minden számlálóját és kiüríti minden memóriáját)

b) π végállapotba lép.

π_1 tehát követi π-t: a fenti esetekben ugyanabba az állapotba lép át, mint amaz. Könnyű látni, hogy π_1 kimenetele mindig π kimenetelének egy részsorozata lesz, továbbá e részsorozatra π_1 működésének definíciója miatt mindig teljesül a Markov-lánc tulajdonság.

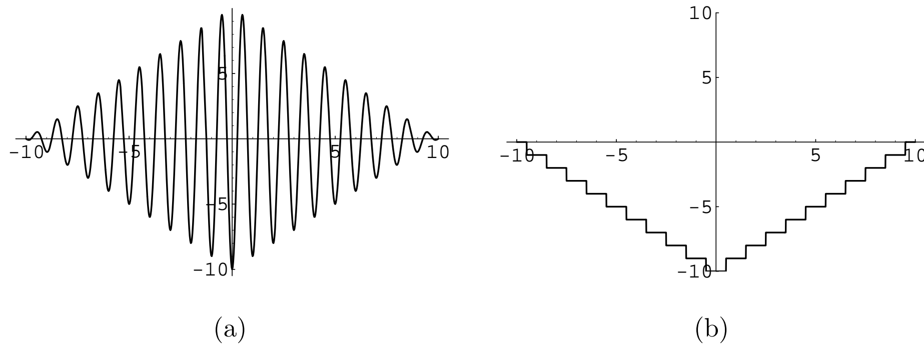
Ez a megoldás elvileg bármilyen π lokális keresési módszer Markov-tulajdonságúvá tételére alkalmas. Természetesen amikor π-nek egyáltalán nincs olyan állapotátmenete, aminek során alaphelyzetbe hozza magát (például az SA-módszerek esetén), akkor π_1 kimenetele mindig egy csupán 2 (vagy 1) hosszúságú pályagörbe lesz, ami a kezdőállapotból és a végállapotból áll (ha különbözők). Az ilyen módszerek számára tehát nem jelent érdemi segítséget ez az átalakítás.

Fontos, hogy π_1 általában nem monoton, amennyiben π nem volt az. Azonban monotonná alakítható egy teljesen hasonló megoldással, mint a fenti, tehát egy π_2 lokális keresésbe foglalva, amely π_1-et követi, de csak akkor (pontosan akkor) hajt végre állapotátmenetet, amikor π_1 olyan állapotba lép, amely az összes eddiginél jobb. (Az angol irodalom ezt *best-so-far*, azaz „eddigi legjobb" átalakításnak nevezi.) Természetesen ez nem rontja el π_1 Markov-tulajdonságát.

2.4.3 A STAGE algoritmus működése

A módszer alapját az a megfigyelés képezi, hogy a lokális keresések hatékonysága nagyban függ attól, hogy mennyire szerencsés kezdőállapotból indították el őket. Az algoritmus több lokális keresést végez és ezek kimenetelei alapján egy tapasztalati állapotértékelő függvényt alakít ki, amit arra használ, hogy egyre ígéretesebb állapotokból indítsa újra az alája rendelt lokális keresést.

Az algoritmus bemutatásához tekintsük a 2.3. (a) ábrán látható $[-10, 10] \ni x \mapsto o(x) := (|x| - 10)\cos(2\pi x)$ függvényt. Ennek globális minimumát általában egyik lokális keresési módszer sem találná meg, kivéve, ha nagyon szerencsés módon éppen a $(-\frac{1}{2} \ldots \frac{1}{2})$ intervallumból indítják el. A (b) ábrán egy ebből képzett másik függvényt ábrázoltunk:



2.3. ábra: *(a): jósági függvény egy egydimenziós minimalizálási feladahoz.*
(b): optimális állapotértékelő függvény, mely a keresési tér pontjaiban megjósolja az onnan indított lokális keresés végeredményét

ez minden ponthoz az onnan indított lokális keresés által talált legjobb értéket rendeli. Ez a függvény sokkal egyszerűbb struktúrájú és láthatóan sokkal informatívabb a globális optimum helyét illetően, mint az eredeti. A STAGE algoritmus ezt az utóbbi, *optimális állapotértékelő függvénynek* nevezett f függvényt[4] igyekszik megtanulni *előrejelezni.* Más szóval azt tanulja előrejelezni, hogy egy x állapothoz milyen eredményt rendelne a π lokális keresés, ha onnan indítanánk el. Ezt gépi tanulás bevonásával teszi, amelyet esetünkben $(x \mapsto o(x_{|\tau|}))$ tanító adatokkal tanít (ahol $\tau = x_1 x_2 \ldots x_{|\tau|} = \pi(o, x)$). A gépi tanulásnak sok példára van szüksége, de hogy ne kelljen emiatt rengeteg lokális keresést csinálni, a STAGE algoritmus a keresések pályagörbéinek analizálásából is nyer adatokat. Ha az alkalmazott lokális keresési módszer *Markov-tulajdonságú* (lásd a 2.4.1. pontban), akkor a pályagörbék mindegyik pontja egy-egy tanító adatként szolgálhat, hiszen ezek a közbülső pontok is tekinthetők keresési kezdőpontoknak. Így egyetlen lokális keresés keresések tucatjait, vagy akár százait is „szimulálhatja". Ilyenkor a pályagörbe mindegyik pontjához a keresés végeredményét rendelve kapjuk a tanító adatokat: $(x_1 \mapsto o(x_{|\tau|}))$, $(x_2 \mapsto o(x_{|\tau|}))$, ... stb.

A keresési tér általában olyan nagy, a rendelkezésünkre álló idő pedig olyan rövid, hogy általában nem számíthatunk arra, hogy akár csak egyetlen fontos részterületet is fel fogunk tudni deríteni. Ezért az optimális állapotértékelő függvényről gyűjtött tapasztalatokat egy nagyon jó előrejelző képességgel rendelkező közelítéssel kell leírnunk, és ennek előrejelzésére kell rábíznunk magunkat. Erre egy természetesen adódó megol-

[4] Formális definíciója a 6. fejezetben található

dás az, hogy függvény-approximátort alkalmazunk a szóban forgó f függvény értékének közelítésére. Az ilyet hívják értékfüggvény-közelítésnek (*value function approximation*, VFA).

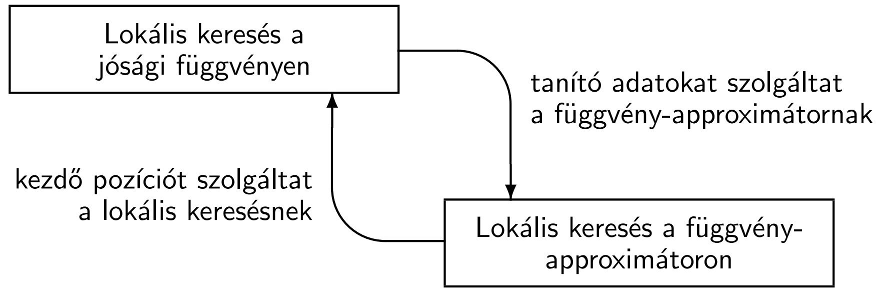
A függvény-approximátor alkalmazhatóságához rendszerint szükség van arra, hogy a függvény-approximátor bemenetei (esetünkben az állapotok) valós számokból álló vektorok legyenek. Tehát az állapotokat valós számokból álló vektorokká kell kódolnunk valahogyan. (Ennek módja nem mindig kézenfekvő, például amikor különféle gráfok az állapotok.) Ezeket a valós elemű vektorokat a továbbiakban *tulajdonságvektoroknak* nevezzük. Formálisan a tulajdonságvektorok $\Re^d \supseteq \mathcal{Y}$-beli vektorok, ahol d valamilyen pozitív egész szám, és az $F : \mathcal{X} \to \mathcal{Y}$ tulajdonság-függvényt használjuk előállításukra. A függvény-approximátort (Φ) tehát nem közvetlenül az $\mathcal{X}$-beli állapotokra, hanem F kimenetére alkalmazzuk: $\Phi(F(x)) \approx f(x)$. (Így tehát a tanító adatok is $(F(x_i) \mapsto o(x_{|\tau|}))$ alakú példák az általános esetben, ahol $i = 1 \ldots |\tau|$.) A $\Phi(F(x))$ közelítést a továbbiakban $\hat{f}(x)$-el jelöljük, és ezt az $\hat{f}$-ot értékbecslő függvénynek mondjuk.[5]

Az így kapott közelítést a következőképpen használjuk: abból az állapotból kiindulva, amelyben a jósági függvényen végzett lokális keresés megállt, egy újabb lokális keresést indítunk, ezúttal az $\hat{f}$ optimumának megkeresésére. Ezáltal ugyanis egy olyan állapotba jutunk, amelynek becsült értéke — vagyis az onnan induló lokális keresés becsült eredménye — jobb, mint ott, ahol az előző lokális keresés megállt. Ebből az új állapotból tehát érdemes újrakezdeni a jósági függvény optimalizációját. Mindez persze csak akkor igaz, amikor az $\hat{f}$ közelítő függvény már jól becsüli a lokális keresés eredményét. A becslés folyamatos javulása érdekében az $\hat{f}$ optimalizációjának megkezdése előtt mindig előbb analizáljuk a jósági függvényen történt lokális keresés pályagörbéjét, tanító példákat gyártunk belőle és tanítjuk a függvény-approximátort.

A STAGE algoritmus tehát „kétütemű”: egyikszer a jósági függvényt optimalizálja lokális kereséssel, másikszor az $\hat{f}$ értékbecslő függvényt optimalizálja egy másik lokális kereséssel. (A két lokális keresési módszer lehet ugyanaz, de ez nem szükségszerű.) Ezt a működési ciklust szemléteti a 2.4. ábra. A két ütem egymást felváltva működik, innen ered az algoritmus neve (*stage* = szakasz, lépcsőfok).

Azért használunk lokális keresést az $\hat{f}$ optimalizálására is, mert az F tulajdonságfüggvényről nem tettük fel, hogy invertálható. Ha létezne is valamely — a lokális keresésnél

[5] $\hat{f}$ pontos definícióját lásd a 6. fejezetben



2.4. ábra: *A STAGE algoritmus kétlépcsős működési ciklusa*

hatékonyabb — módszer a Φ függvény-approximátor optimumának megtalálására, a kapott $y \in \mathcal{Y}$ tulajdonságvektor $\mathcal{X}$-beli ősét nem tudnánk megmondani. Ezért tehát $\mathcal{X}$-beli mozgásokat végezve kell eljutnunk az $\hat{f}$ optimumát jelentő $\mathcal{X}$-beli újraindító állapothoz. Azokban az esetekben, amikor ez az állapot nem különbözik a jósági függvényen végzett lokális keresés végállapotától — mert ez az állapot lokális optimuma mind a jósági függvénynek, mind $\hat{f}$-nak — egy véletlenszerűen választott állapotot jelölünk ki újraindító állapotnak.

A STAGE algoritmus tehát felfogható egyfajta intelligens újraindító rendszerként is az alkalmazott lokális keresési módszer fölé. Alkalmazásához a következő bemenő információkra van szükség:

- maga a globális optimalizálási probléma: egy keresési tér (tetszőleges nem üres $\mathcal{X}$ halmaz) a rajta értelmezett o jósági függvénnyel

- mozgási műveletek a keresési térben, vagy ezzel ekvivalens módon egy N szomszédsági struktúra. (Bizonyos esetekben annak is lehet értelme, hogy eltérő szomszédsági struktúrákat használjunk a jósági függvény és az értékbecslő optimalizálásakor: ilyenkor tehát két szomszédsági struktúra kell.)

- egy Markov-tulajdonságú lokális keresési módszer (π)
  (Ha az értékbecslő függvényre másmilyen keresési módszert alkalmazunk, annak nem kell Markov-tulajdonságúnak lennie.)

- tulajdonság-függvény ($F : \mathcal{X} \to \mathcal{Y}$)

- függvény-approximátor ($\Phi : \mathcal{Y} \to \Re$)

- véletlen állapotot előállító módszer ($r \in \tilde{\mathcal{X}}$)

- $\omega \in \Re$ felső korlát a jósági függvény értékére (minimalizálási problémánál alsó korlát). Ha a korlát nem ismert, $\omega = \infty$ is lehet. Használatát illetően lásd a következő példát.

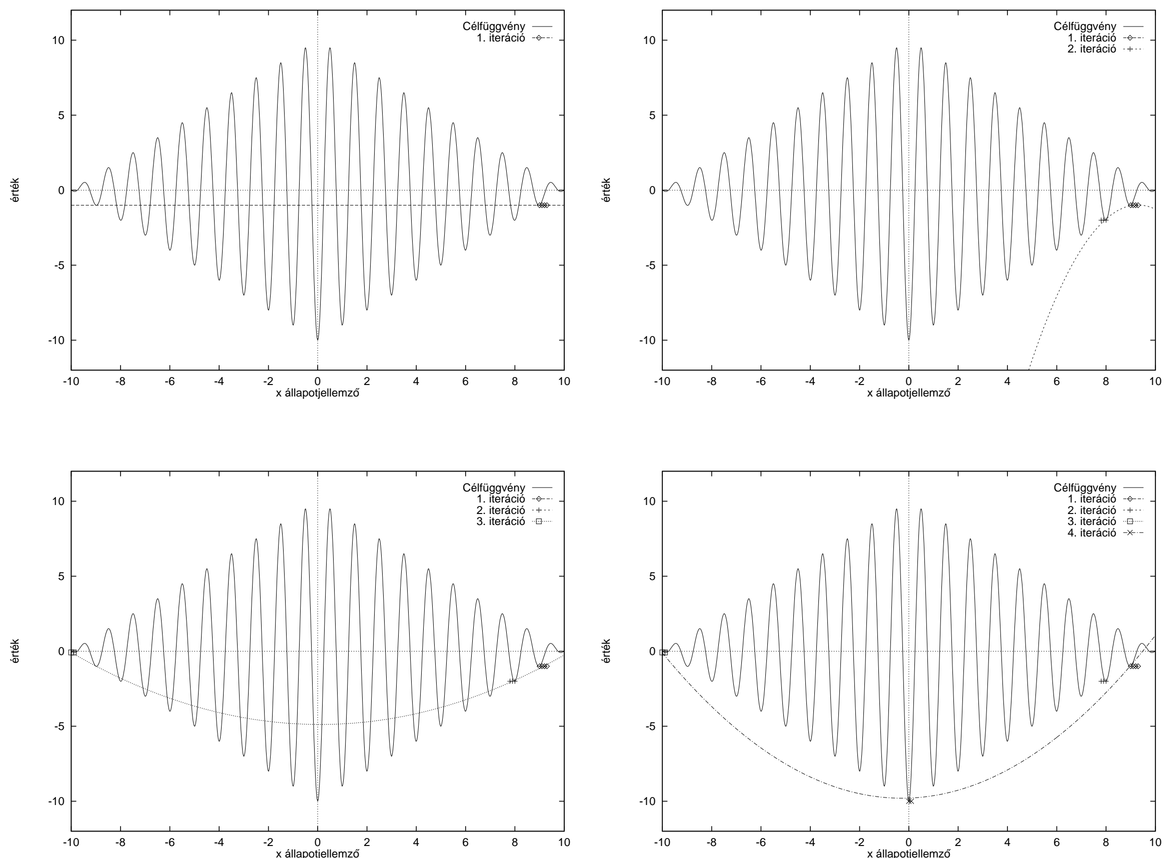
Példa

Tekintsük ismét a 2.3. (a) ábrán szereplő egydimenziós hullámfüggvényt.

Ennél a keresési tér az $\mathcal{X} = [-10, 10]$ intervallum, a jósági függvény pedig az $\mathcal{X} \ni x \mapsto o(x) := (|x| - 10)\cos(2\pi x)$ függvény, amit minimalizálni akarunk. Az egyszerűség kedvéért legyen most $\omega = -\infty$. Lokális keresésként alkalmazzunk egyszerű hegymászó algoritmust, a szomszédsági struktúra pedig legyen $N(x) = \{x - 0.1, x + 0.1\} \cap \mathcal{X} \quad (\forall x \in \mathcal{X})$. Az F tulajdonságfüggvény lehet egyszerűen az identitás ($\mathcal{Y} := \mathcal{X}$), függvény-approximátorként pedig használjunk kvadratikus regressziót (ami $d = 1$ miatt most másodfokú polinommá egyszerűsödik).

Az algoritmus először egy véletlen állapotból indítja el a hegymászó algoritmust. Tegyük fel, hogy ez a véletlen állapot az $x = 9.3$. A lokális keresés három lépésben lejut az o függvény grafikonjának $(9, -1)$ pontjába, ami egy lokális minimum (lásd a 2.5. ábrán). Ez a pályagörbe a következő tanító adatokat szolgáltatja: $(9.3 \mapsto -1)$, $(9.2 \mapsto -1)$, $(9.1 \mapsto -1)$, $(9.0 \mapsto -1)$. (A bal felső grafikonon kis rombuszok szemléltetik őket.) Ezen adatok hatására a függvény-approximátor természetesen konstans függvénnyé válik: $\hat{f} \equiv -1$, amint az ábrán is látható. Így az $\hat{f}$-on végzett optimumkeresés nem mozdul el semerre, hiszen egyik irányba sem javítana az elmozdulás, és az a helyzet alakul ki, hogy az értékbecslő lokális optimuma megegyezik a legutóbbi lokális keresés eredményével. A mondottak értelmében ilyenkor véletlen újraindító állapotot kell választanunk.

Tegyük fel, hogy ez a véletlen kiinduló állapot a második iteráció elején $x = 7.8$. Innen a hegymászó algoritmus a grafikon $(8, -2)$ pontjába jut le, ezáltal a következő tanító adatokat produkálja: $(7.8 \mapsto -2)$, $(7.9 \mapsto -2)$, $(8.0 \mapsto -2)$. Az eddigiek alapján a függvény-approximátornak azt kell hinnie, hogy az elérhető minimum meredeken csökken az x állapotjellemző értékének csökkenésével együtt, és ennek megfelelően egy „meredeken” optimista értékbecslő függvényt hoz létre, amint az a jobb felső grafikonon látható. Az ezen való lokális keresés elvisz bennünket egészen a keresési tér túlsó széléig, az $x = -10$ állapotig, ahova az értékbecslő $\hat{f}(x) = -212$ értéket jósol! Az ilyen



2.5. ábra: *A STAGE algoritmus működése az egydimenziós hullámfüggvény-példán*

esetek korlátozására szolgálna az ω korlát: amikor a becslés már ennek az értékét is meghaladja, vagyis bizonyosan túlságosan optimista, akkor az $\hat{f}$-on való lokális keresést leállítjuk, hogy ne vigyen ki bennünket a keresési térnek egészen a széléig. Jelen esetben ez nem okoz problémát, ezért nem használjuk ezt a korlátozást. A harmadik iteráció így az $x = -10$ állapotból indul, és hamar megbünteti az $\hat{f}$ túlságos optimizmusát. A lokális keresés csupán egyetlen lépést tesz: $(-10, 0)$-ból $(-9.9, -0.08)$-ba. Két tanító adat keletkezik: $(-10 \mapsto -0.08)$, $(-9.9 \mapsto -0.08)$. (Kis négyzetkék mutatják őket az ábrán.) Miután betanítjuk ezeket, a kialakuló közelítő függvény már helyesen jelzi, hogy a globális minimum a keresési tér közepe táján várható (bal alsó grafikon). Az értékbecslőn való lokális keresés az $x = 0.1$ új indító állapotba juttat el bennünket.

A negyedik lépés innen már könnyen eltalálja a globális minimumot, a $(0, -10)$-et. Ennek az egylépéses pályagörbének az adatait betanítva olyan $\hat{f}$ alakul ki, amely már

mindig ide, a globális optimumba juttatja vissza a lokális keresés kezdőállapotát.

Ezt a problémát tehát sikerült megoldani. Jól látható, hogy a STAGE nem egyszerűen kisímítja a jósági függvény rücskeit, hanem közben a lokális keresésre vonatkozó előrejelző tudást is beépít a símító függvénybe.

2.4.4 A függvény-approximátor

Az előző példában kvadratikus regressziót használtunk, de valójában nagyon sokféle függvény-approximációs technika létezik még ezen kívül: vannak magasabb fokú polinomiális regressziók is, azután ott vannak a legközelebbi szomszéd módszerek, a különféle neurális hálók, a többdimenziós spline-ok, a döntési fák stb., hogy csak a legjelentősebbeket említsük. Melyik ezek közül az, amelyik legjobban megfelel a STAGE algoritmusban való alkalmazásra? Egyáltalán milyen technikák jöhetnek szóba erre a célra? A legfontosabb elvárásaink a következők:

Inkrementalitás A STAGE algoritmus nagyon sok, nemritkán milliós nagyságrendű példát generálhat a függvény-approximátor (Φ) számára az optimalizáció futása során, ezért a függvény-approximátornak képesnek kell lennie ilyen mennyiségű példa feldolgozására anélkül, hogy túl sok memóriát vagy számítási kapacitást vonna el hozzá. A tanítás továbbá minden STAGE-ciklusban ismétlődik, ezért gyorsnak kell lennie. Φ kiértékelésének pedig különösen is gyorsnak kell lennie, mivel $\hat{f}$ optimalizálásának minden lépésében szükség van rá. Az ezeknek eleget tevő függvény-approximátorokat *szigorúan inkrementálisnak* nevezzük.

Zajtűrés Az optimális állapotértékelő függvényről gyűjtött tanító adatok hosszú sztochasztikus keresések pályagörbéinek eredményeiből származnak, ennélfogva eredendően zajosak. A függvény-approximátornak tehát képesnek kell lennie a tanító halmazban jelenlévő lényegi zaj elviselésére.

Előrejelző képesség Az $\hat{f}$ optimalizálása során sokszor olyan állapotokon is megkérdezzük a függvény-approximátor véleményét, amelyeket előzőleg egyetlen egyszer sem értékeltünk ki. Emiatt a STAGE nagy hasznát veszi, ha a függvény-approximátor előre tudja vetíteni a példahalmazban rejlő tendenciákat. A **??**. ábrán a kvadratikus regressziót és a legközelebbi szomszéd módszert hasonlítottuk össze

egy kis egydimenziós példán. Bár a kvadratikus approximáció csak jóval nagyobb közelítési hibával tud illeszkedni az adatokra, mégis sokkal hasznosabb a hegymászó algoritmus számára, mint a másik módszer. Emlékezzünk még a STAGE ω korlátjára is, melynek segítségével a függvény-approximátor túlzó becsléseit kompenzálhatjuk.

E követelmények alapján az adódik, hogy a STAGE számára ideális függvény-approximátorok a *lineáris architektúrák* osztályába tartoznak, melyeknek általános alakja

$$\Phi(y) = \sum_{j=1}^{K} c_j \phi_j(y) \qquad (y \in \mathcal{Y})$$

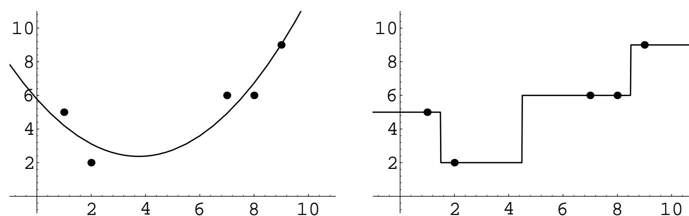
ahol a ϕ_j-k előre rögzített, könnyen számolható, valós értékű *bázisfüggvények*; a c_j-k pedig alkalmas konstansok: ezeknek értékét tanulja meg optimálisan beállítani a függvény-approximátor. A különböző polinomiális regressziók is ebbe az osztályba tartoznak, így például a kvadratikus regresszió az alábbi bázisfüggvényekkel adódik:

$$\begin{array}{rcllllll} (\phi_1(y), \ldots, \phi_K(y)) & = & (1, & y_1, & y_2, & y_3, & \ldots, & y_d, \\ & & & y_1^2, & y_1 y_2, & y_1 y_3, & \ldots, & y_1 y_d, \\ & & & & y_2^2, & y_2 y_3, & \ldots, & y_2 y_d, \\ & & & & & & \ddots & \\ & & & & & & & y_d^2) \end{array}$$

ahol $y = (y_1, \ldots, y_d) \in \mathcal{Y}$ és $K = \frac{(d+1)(d+2)}{2}$.

A kvadratikus regresszió azért jó, mert képes felismerni és előrevetíteni mind az első-, mind a másodfokú tendenciákat, amennyiben léteznek a tulajdonságvektorok terében. Továbbá jól tűri a zajt is, és a szigorú inkrementalitás követelménye is teljesül rá. Ez utóbbi bemutatásához jelöljük a tanító példákat az alábbi módon:

$$\{(x_1 \mapsto z_1), (x_2 \mapsto z_2), \ldots, (x_m \mapsto z_m)\} \qquad (x_1, \ldots, x_m \in \mathcal{X},\ z_1, \ldots, z_m \in \Re)$$



2.6. ábra: *A kvadratikus regresszió és a legközelebbi szomszéd módszer összehasonlítása.*

A tanulás célja az, hogy megtaláljuk azt a $\underline{c}^* = (c_1, \ldots, c_K) \in \Re^K$ vektort, amellyel a

$$z_k = \eta_k + \Phi(F(x_k)) = \eta_k + c_1\phi_1(F(x_k)) + \ldots + c_K\phi_K(F(x_k)) \qquad (k = 1 \ldots m)$$

egyenletrendszerben az $\underline{\eta} = (\eta_1, \ldots, \eta_m)$ közelítési hibavektor euklideszi normája minimális lesz. Ez nem más, mint a legkisebb négyzetes közelítés problémája, amelynek megoldására hatékony algebrai módszerek léteznek (lásd pl. [7]). Jelöljük $\underline{\phi}(y)$-nal a $(\phi_1(y), \ldots, \phi_K(y))$ vektort ($\forall y \in \mathcal{Y}$). Ekkor $\underline{c}^*$ előállításához elegendő a következő két statisztikát folyamatosan vezetni:

$$\mathbf{A} = \sum_{k=1}^{m} \underline{\phi}(F(x_k))\underline{\phi}(F(x_k))^T \qquad\qquad \mathbf{b} = \sum_{k=1}^{m} \underline{\phi}(F(x_k))z_k$$

ahol $\mathbf{A}$ egy $K \times K$-as valós elemű mátrix, $\mathbf{b}$ pedig egy $\Re^K$-beli vektor. Ezekből $\underline{c}^*$ így számítható:

$$\underline{c}^* = \mathbf{A}^{-1}\mathbf{b}$$

Itt az inverzszámítás elvégzésére szinguláris érték felbontást javaslunk, mivel ez a olyankor is robusztus, amikor $\mathbf{A}$ szinguláris. Ez egyébként olyankor következik be, amikor a közelítendő adatok nagyon ellentmondásosak. Kvadratikus regressziónál például akkor, amikor nagyon kevésen múlik az, hogy pozitív vagy negatív előjelű parabola illeszkedik-e jobban az adatokra. Tipikusan olyankor történik ez, amikor az állapotjellemzők nem igazán jók, és ezért a tulajdonságvektorok terében nincs sem elsőfokú, sem másodfokú tendencia.

A STAGE algoritmus minden ciklusában tanító példák sokasága gyűlik össze, ezeket egyszerre, egy lépésben adjuk át a függvény-approximátornak tanulásra. Lineáris architektúrák esetén ez a lépés egyszerű, mert csak a fenti statiszikákat (amelyek voltaképpen számlálók) kell aktualizálni, esetünkben $\mathbf{A}$ és $\mathbf{b}$ elemeit növelni az új példákból számított részletösszeggel. Ezután a tanító példákat elfelejthetjük. $\underline{c}^*$ frissítésének számításigénye $O(K^3)$, tehát nem függ az eddigi tanító példák számától. (Fontos azonban megjegyezni, hogy az állapotjellemzők számának növelése viszont lényegesen megnöveli a kvadratikus regresszió használatának költségeit. Mivel K d-nek a négyzetével arányos, azért az állapotjellemzők számának minden megduplázódása $\underline{c}^*$ számításigényét 64-szeresére növeli!)

A lineáris architektúrák alkalmazásának másik előnye, hogy nemcsak a számításigény, hanem a memóriaigény sem növekszik az összegyűjtött példák gyarapodásával együtt.

Nem kell tárolni a teljes tanítóhalmazt, hanem csak az **A** és **b** számlálókat. Ennélfogva a szűk keresztmetszet az aktuális pályagörbe adatainak tárolására tevődik át, mert ezeket viszont tárolni kell ahhoz, hogy a végén tanító példákat tudjunk generálni belőlük. Ez azonban általában nem szokott igazán komoly gondot jelenteni.

J. A. Boyan a dolgozatában ([12]) számos nagyszabású problémán (ládapakolás, csatorna-irányítás VLSI tervezésben, Bayes-hálók struktúrájának kialakítása, radioterápiás kezelések tervezése, térképtervezés, logikai formulák kielégíthetősége) próbálta ki a STAGE algoritmust. A tapasztalatok mind azt mutatják, hogy egyszerű függvény-approximátort, lineáris vagy kvadratikus regressziót érdemes használni.

2.5 A Stagenis algoritmus

A Stagenis algoritmus a STAGE és a genetikus algoritmus ötvözete. A STAGE heurisztikáját a GA heurisztikájával egészíti ki és viszont. Technikailag a Stagenis algoritmus egy módosított genetikus algoritmus, amelynek populációjában az egyedek a jósági függvényen való lokális keresés számára jelentenek kezdőállapotokat. Az egyedek fitnesszértékét a jósági függvényen való lokális keresés végeredménye szolgáltatja. A legjobbnak értékelt egyedeket mindig átmásoljuk az új generációba, ez adja az új generáció 30%-át. További 60%-ot egypontos keresztezéssel származtatunk (mint a 12. oldalon). A fennmaradó 10%-ot pedig egy új genetikus operátorral, az úgynevezett *stagenis operátorral* állítjuk elő.

A stagenis operátor alkalmazásának van egy előfeltétele: az addig végzett fitnesszérték-számítások pályagörbéinek adataiból tanító példákat kell készíteni, és azokat be kell tanítani a STAGE függvény-approximátorának. Ha ez megtörtént, akkor a kiválasztás operátor alkalmazásával bekérünk egy egyedet, és ennek végállapotából (a fitnesszértékszámítás lokális keresésének végállapotából) kiindulva optimalizáljuk az $\hat{f}$ értékbecslő függvényt, ugyanúgy, mint a STAGE algoritmus második ütemében. Az így kiválasztott újraindító állapot lesz a stagenis operátor kimenete, ez az egyed kerül be a következő generációba új egyedként.

Ezen a ponton meg kell említeni egy technikai részletet. Annak érdekében, hogy az operátor ismételt alkalmazásai egymástól eltérő egyedeket eredményezzenek, a stagenis operátort az új generáció egyedeinek kiértékelései között egyenletesen elszórva érdemes alkalmazni. Így az egymást követő stagenis operátor alkalmazások között fitnessz-számítások is történnek, ami azért fontos, mert módosítják az értékbecslőt a stagenis operátor következő alkalmazására. Annak ugyanis nem túl sok értelme volna, hogy változatlan értékbecslővel alkalmazzuk ismételten a stagenis operátort.

A stagenis operátor az újraindító állapot kiválasztásakor szükség esetén véletlen állapot választást is végrehajt (lásd a 25. oldalon). Ezáltal szükségtelenné válik a mutáció alkalmazása a Stagenis algoritmusban. A véletlen állapot választás biztosítja a teljes térből való véletlen mintavételezést, így a populáció degenerálódása hosszú távon nem következhet be.

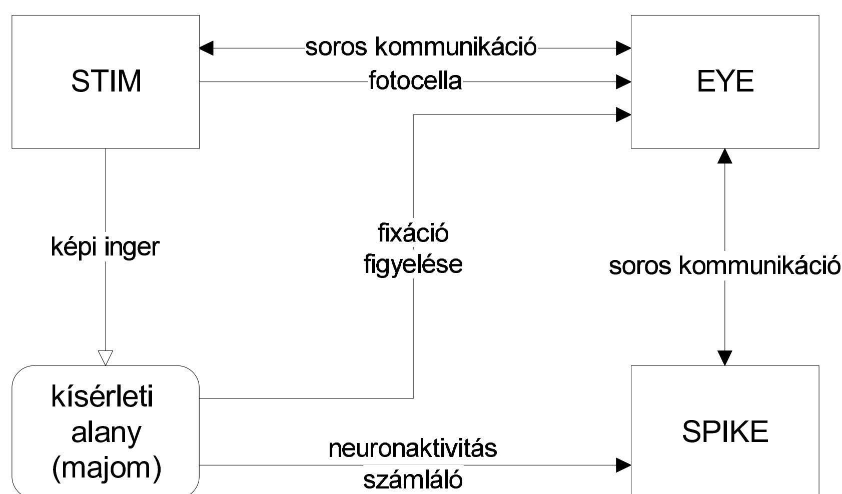
Mint mondottuk, a populációban szereplő egyedek *kezdőpontjai* a jósági függvényen

való lokális kereséseknek. Azért döntöttünk a kezdőpontok mellett, mert ezek kevésbé specifikusak az éppen megoldás alatt álló feladatra nézve, mint a végállapotok volnának. A nem túlságosan specializálódott állapotokból álló populáció sikeres egyedeit ugyanis várhatóan eredményesen lehet használni majd más, hasonló feladatok megoldásakor is. Tehát ha egy problémacsaláddal állunk szemben, amely számos hasonló feladatból áll, akkor az újabb és újabb feladatok megoldását gyorsabbá teheti az, ha a populáció kezdeti feltöltésekor korábbi feladatokban már sikeresnek bizonyult állapotokból indulunk ki.

3. A vizsgálatokhoz készített szoftver

3.1 A feladat specifikációja

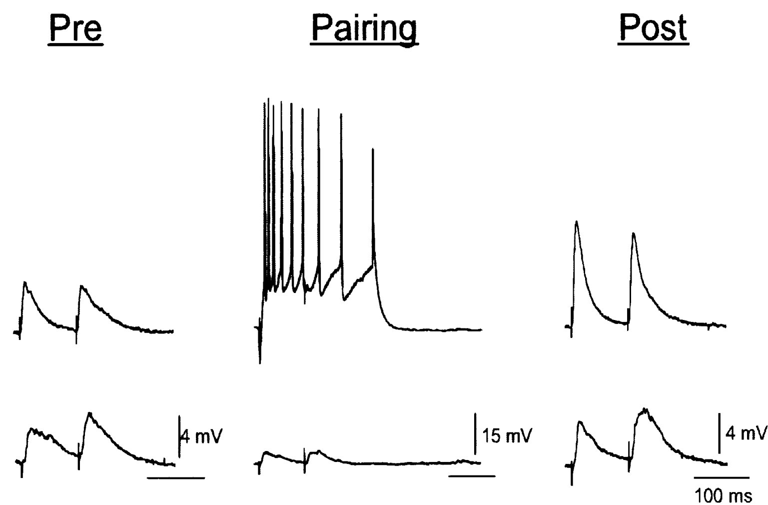
A bevezetőben leírt majomkísérletekhez három számítógépre és két szobára van szükség. Az egyik szobában a kutató tartózkodik, a másikban a kísérleti majom. Erre azért van szükség, hogy az állatot semmi ne zavarja a koncentrálásban. A 3.1. ábrán látható a hardver eszközök kapcsolati rajza. A számítógépeket a kísérletben játszott szerepük alapján rövidített névvel jelöltük.



3.1. ábra: *A majomkísérletekben használt komponensek*

A három számítógép általában hagyományos soros porton (RS 232 szabvány) keresztül kommunkál egymással. A STIM és az EYE számítógépek között a pontosabb szinkronizálás érdekében egy fotocellás kapcsolat is fel van építve.

A STIM név az angol *stimuli* szó rövidítése, ennek a számítógépnek a feladata a kért input kép megjelenítése a majom számára. Itt nagyon fontos a jó szinkronizálás, mivel egy tizedmásodpernyi eltérés már nagy mérési hibát eredményezhet. A méréshez



3.2. ábra: *Egy neuron aktív állapotban való működése. Középen láthatjuk az inger hatására kiváltott csúcsokat („tüzelés"), baloldalt az inger előtti, jobboldalt pedig az inger utáni viselkedést.*

pontosan tudnunk kell, hogy mikor kapta meg a kísérleti állat a kívánt ingert, és hogy az inger mikor szűnt meg.

A SPIKE gép interfészként működik. A kísérleti állat látóidegrendszerében levő neuron által leadott jelet konvertálja át feldolgozható formába. A 3.2. ábrán látható egy neuron aktív állapotban való működése. A vízszintes tengelyen az eltelt idő, a függőlegesen pedig a neuron kimeneti nyúlványán mérhető elektromos potenciál látható (forrás: [13]). Megfigyelhetjük, hogy hirtelen ugrások, úgynevezett hegycsúcsok, idegen szóval *spike*-ok láthatók az aktivitásfüggvényen. Innen ered a SPIKE elnevezés, ugyanis egy bemeneti kép által kiváltott inger mértéke a hegycsúcsok számával mérhető.

Az EYE gép feladata egyrészt a fókuszálás figyelése. Ez nagyon fontos, mivel csak akkor tekinthető egy próba (*trial*) értékelhetőnek, ha közben az állat folyamatosan a neki mutatott ábrát nézte, és nem kalandozott el a tekintete más irányba. A méréseket éppen ezért egy viszonylag hosszú betanítási szakasszal kell előkészíteni, hogy a majom a kellő pillantban a neki mutatott képet nézze. Másrészt ez a számítógép vezérli a kísérletet. Elvégzi a szükséges szinkronizálást, archiválja a mért adatokat, és ha a mérésbe hiba csúszott (ez általában fókuszálási hiba), akkor megismétli azt.

Az a szoftver, amellyel jelenleg a kísérleteket folytatják, Leuvenben készült. Ellátja a három számítógép szinkronizálását, és képes előre elkészített képek megjelenítésére a

STIM gépen. Itt kapcsolódik be a mi munkánk. Egy olyan keresőprogramot készítünk, ami előre elkészített képek helyett a kellő időpillanatban generálja le a színteret az addigi mérések eredményei alapján. A szoftver a STIM gépen fog futni. Ez a módszer lényegesen felgyorsítja a keresést, mivel nem kell megvárni egy kísérlet végét ahhoz, hogy megtervezzük a következő képcsoportot, ezenkívül a kereső algoritmus az éppen aktuális neuron által leadott jelre támaszkodik, amit off line módon generált képeknél lehetetlen lenne kivitelezni.

A neuron válaszának modellezése számítógépen

Mivel a kísérletek Leuvenben folynak, a szoftver fejlesztése pedig Budapesten, a fejlesztés idejére szükség van a neurális válasz számítógépes modellezésére. Konkrétan egy olyan függvény megadására törekedtünk, ami véleményünk szerint jól modellezi a neurális választ, és számítógéppel lehetőség szerint gyorsan számítható.

A képek reprezentálása számítógépen általában három $m \times n$-es mátrixban történik, ahol m a kép függőleges, n a vízszintes mérete pontokban mérve. A három színkomponens (vörös, zöld, kék) tárolása miatt van szükség három mátrixra. Mivel a látóidegrendszernek az a része, amellyel foglalkozunk, invariáns a színekre, elég a kép fényintenzitás-térképét figyelembe vennünk, azaz a kép szürkeárnyalatokra konvertált változatát használnunk. Erre egyfajta konvertálási lehetőség a kép RGB színskálából HSL színskálába való transzformációja után az L, azaz a fényerő komponens megtartása. Nem helyes módszer a színenkénti fényintenzitások átlagolása. Jó lineáris közelítő módszer viszont az $RGB \to L$ transzformációra az alábbi

$$L = \frac{1}{256}(77R + 150G + 29B)$$

képlet.

Tegyük fel, hogy egy neuron egy bizonyos képre érzékeny. Nevezzük ezt a bizonyos képet célképnek, és jelöljük M-mel. Ekkor M egy $m \times n$-es valós mátrix, a célkép fényintenzitás-térképét tartalmazza.

Jelöljük M^k-val az M mátrix (kép) k-adik elmosottját. Elmosott mátrixot többféleképpen hozhatunk létre:

1. Az egyik módszer a wavelet-analízis. Legyen T az M mátrix egy wavelet felbontásának fája, T^k pedig a fa k-adik szintjének 0. eleme. Ez felel meg a wavelet

felbontás alacsony frekvenciás komponeneseinek. Ekkor $M^k := T^k$ az M mátrix k-adik elomosottja.

2. Másik módszer a Gauss-függvénnyel való diszkrét konvolúció. Legyen $g(x) := exp(...)$... Ekkor $M^k := ...$ az M mátrix k-adik elomosottja.

Elmosásra példát a **??**. ábrán láthatunk.

<!!! ábra (kép és különböző szintű elmosottjai, különböző módszerekkel) !!!>

Legyen N egy másik $m \times n$-es mátrix, azonos méretű M-mel. Ekkor definiálhatjuk M és N távolságát. Legyen $\delta(M, N) := \sqrt{\sum_{i=1...m} \sum_{j=1...n} (M_{i,j} - N_{i,j})^2}$. Legyen $\delta^k(M, N) := \delta(M^k, N^k)$ a k-adik szintű elmosottak távolsága. Legyen $\Delta^k(M, N) := \sum_{n=0...k} \delta^n(M, N)$ az M és N k-adik szintű távolsága.

Az elgondolás alapja egy sejtés, miszerint a látóidegrendszer inferotemporalis cortexet megelőző területei wavelet analízishez hasonlóan bontják fel a képet alcsony- és magasfrekvenciás komponensekre. A wavelet analízis segítségével történő elmosással kapott távolságfüggvény erre az elgondolásra alapozna. A konvolúciós eljárás előnye a gyorsabb számíthatóság.

A neuron működésének szimulálására végül a 0. szintű távolságot választottuk, azaz a $\delta(M, N)$ függvényt. Ennek előnye a gyors számíthatóság. Egy wavelet analízisen alapuló, 4. szintű távolság kiszámítása körülbelül egy percet vesz igénybe egy mai számítógépen, ami nem elégséges az optimalizációs szoftver teszteléséhez. Természetesen a szoftver készítése során ügyeltünk arra, hogy ha a számítógépek teljesítménye elégséges lesz egy magasabb szintű távolság gyors számításához, akkor a távolságfüggvény könnyen kicserélhető legyen egy magasabb szintű távolságot kiszámító függvényre.

<!!! ábra (szint-idő, szint-érték) !!!>

3.2 Megvalósítás

Mivel a Leuvenben használandó szoftver nem alkalmas különböző keresési algoritmusok összehasonlító analizálására, ezért készült egy másik program is, amelyből hiányzik a soros vonali kommunikáció, a képi megjelenítés, viszont jobban paraméterezhető, és tartalmazza a fenti módon szimulált neuronaktivitás függvényt. Ez a program kötegelt és hosszú idejű futtatásokra lett tervezve, futása közben statisztikát készít.

3.2.1 Az adatok reprezentálása

A jelenleg folyó kísérletekben a stimuli képeket a Kinetix cég által készített 3D Studio MAX szoftverrel tervezik. A mi programunk is ezt a szoftvert használja a megjelenítendő képek generálásához. A 3D Studio MAX háromdimenziós szinterek modellezésére képes, azokról tetszőleges szögből, nagyításban és fényviszonyok közötti valószerű képek előállítására használható. A szinterek úgynevezett geonokból állnak. A 3D Studio MAX program geonnak nevezi a szinteret felépítő egyszerűbb térgeometriai formákat. Leuvenben főleg gömböt, téglatestet és hengert használnak a szinterek megtervezésekor.

A 3D Studio MAX színtérkezelése engedélyezi a geonok többszintű módosítását. Ez azt jelenti, hogy az alap geonra módosítókat építhetünk rá. Három gyakori módosító a csavarás (*twist*), a hegyezés (*taper*) és a hajlítás (*bend*). Ezek általában valamilyen látványos geometriai transzformációt jelentenek. A módosítókra példa látható a **??**. ábrán. A képcsoportok megtervezésekor gyakori ezeknek a módosítóknak a különböző mértékű alkalmazása.

<!!! ábra (A módosítók hatása a geonokra) !!!>

A színterek reprezentálása rendkívül összetett feladat. Egy színtérleírásnak tartalmaznia kell a színteret alkotó geonokat, esetleg más térgeometriai formákat, a geonok színét, anyagát, fényvisszaverési és fényelnyelési tulajdonságait, a színtéren található fényforrásokat, az ambiens fényt, a kamera helyzetét és állását. Látható, hogy a lista már így is eléggé hosszú, és valószínű, hogy ennél hosszabb lesz a felhasználás során. Ezért úgy döntöttünk, hogy nem készítünk egy új eszközt a színterek reprezentációjához, esetleg megtervezéséhez, hanem inkább felhaszálunk egy létező és a gyakorlatban már kipróbált eszközt. A 3D Studio MAX grafikus interfésze jól megtervezett, három dimenziós modellezésre tökéletesen alkalmas. A színterek leírására a 3D Studio MAX program `.max` kiterjesztésű fájljait választottuk.

A színterek reprezentálásán kívül szükség van még a keresési tér specifikációjára és reprezentációjára. A keresési tér esetünkben azon input képek reprezentációiból áll, amelyeket a kereső algoritmus érinthet működése során.

A geonok tulajdonságai, például egy henger alkotójának átmérője, valós számokkal leírhatók. Leírhatók ugyanígy a módosítók is, konkrétan hogy milyen mértékben alkalmaztunk egy adott módosítót. Ez jelenthet például szöget fokban, de jelenthet bármilyen

önkényesen megválasztott mértékegységben mért mértéket is.

Ha ismerjük a teljes színteret, akkor egy paraméter jelentheti akár egy téglatest magasságát, de jelentheti egy henger behajlítottságát is, a hajlítás módosító használata esetén. Így ha n darab paramétert jelölünk ki keresendőnek, akkor egy n dimenziós keresési problémát kell megoldanunk. A keresési tér legegyszerűbben egy n dimenziós intervallummal és annak diszkretizációjával adható meg. így a keresési tér megadható egy listával, amely lista elemei tartalmazzák a keresendő paraméter egyedi nevét, a paraméter típusát, a paraméterhez tartozó intervallum alsó és felső határát, annak diszkretizációját, valamint az aktuális állapotát. A keresési tér megadására szolgáló formátum szöveges fájlformátum, kiterjesztése `.sea`, ami az angol *search* szóbol ered. A keresési tér leírására példa az alábbi fájl: `test.sea search objects[3][3][2].value.x_rotation float -45 45 5 20.0 search objects[3][3][2].value.y_rotation float -45 45 5 -20.0 search objects[3][3][2].value.z_rotation float -45 45 5 20.0 search objects[3][4][1][3].value float -1 1 0.05 -0.5 search objects[3][4][2][3].value float -50 50 5 35.0 search objects[3][4][3][3].value float -50 50 5 35.0` Minden sornak, ami keresendő paramétert ír le, a search szóval kell kezdődnie. A második oszlopban van a keresendő paraméter 3D Studio MAX által használt belső, egyedi neve. A harmadik oszlop taralmazza a paraméter típusát. Jelenleg csak float típussal dolgozunk. Ez megfelel a C++ double típusának, azaz egy lebegőpontos számábrázolású valós számnak. A következő oszlopok megadják az adott paraméter által a keresés során felvehető minimumot, maximumot, a tartomány diszkretizációját, és a paraméter aktuális értékét.

Az aktuális állapot leírására nem lenne szükség, hiszen azt a színtér tartalmazza, de bizonyos segédprogramok használják ezt az adatot, így a segédprogramokat egyszerűbbé lehetett tervezni.

Egy keresési probléma specifikációja tehát egy 3D Studio MAX `.max` fájl, egy ahhoz tartozó, általunk tervezett `.sea` fájl, és egy keresendő `.bmp` célkép megadásával lehetséges. Itt a célképpel adjuk meg a „neuront”, a `.max` és `.bmp` fájlokkal pedig a „mutatott” képeket. Ennek a megoldásnak az előnye az, hogy egy `.max` típusu fájlhoz több `.sea` fájl is megadható.

3.2.2 Felhasznált szoftver-eszközök

Mivel a Kinetix cég a 3D Studio MAX szoftveréhez a Microsoft Windows NT operációs rendszert ajánlja, mi is ezt választottuk a programjaink elkészítéséhez, teszteléséhez és használatához. A képek generálásához 3D Studio MAX-ot használunk. Az optimalizációs módszereket összerhasonlító program egy Microsoft Visual C-ben készült konzol módú C++ program.

Mivel ezt a két különböző programot használjuk az optimalizációhoz, ezért szükség van a programok közti kommunkációra. A Windows NT operációs renszer többféle processzközti kommunikációs lehetőséget támogat (*interprocess communication*). Például a socket alapú kommunikációt, az osztott memóriás kommunikációt, az üzenetküldést, a távoli metódushívást és az OLE-t. A képek előállításához a két program közti kommunikációt kellett megoldanunk, tehát nincs szükség hálózati kommunikációra. Mivel — jelenlegi tudásunk szerint — a 3D Studio MAX az elkészített képet csak a merevelemzen tudja elhelyezni, ezért nincs szükség arra sem, hogy nagyobb mennyiségű adatot mozgassunk a programjaink között a kommunikációs csatornán. Tehát egy olyan kommunikációs módszerre van szükség, amellyel parancsot adhatunk ki a 3D Studio MAX-nak színterek betöltésére, módosítására és a képek előállítására.

Erre a feladatra mi az OLE (*Object Linking and Embedding*) szabványt választottuk. Az OLE lehetővé teszi egy program számára, hogy függvényeit egy szabványos programozói felületen keresztül meghívjuk egy másik programból.

Az OLE az objektum orientált szabvány. Az OLE objektumok definíciója tartalmazza az interfész definícióját is. Az OLE úgy lett megtervezve, hogy ahhoz, hogy egy OLE objektumot létre tudjunk hozni, nem kell különleges fájlformátumú programkódot létrehoznunk. Egy futtatható programot kell elkészítenünk, amelyben vagy statikusan (fordítási időben), vagy dinamikusan (futási időben) regisztráljuk a megfelelő függvényeket az OLE alrendszerbe. A OLE interfészban levő függvényeket az egyedi nevükkel azonosítja a rendszer. Így az interfész és a mögötte levő függvények definiálhatók.

Definiálandó még az OLE objektum egyedi neve, amelyen az OLE objektumra hivatkozni lehet. Ehhez a Windows rendszerleíró adatbázisában (registry) a HKEY_CLASSES_ROOT kulcs alá kell adatokat beírni. A 3D Studio MAX esetén az alábbi bejegyzéseknek kell bekerülniük a rendszerleíró adatbázisba, ha az OLE interfészt használni

akarjuk:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; registration info MAX 2.0
; (Application Object)
HKEY_CLASSES_ROOT\MAX.Application.2 = OLE Automation MAX 2.0 Application
HKEY_CLASSES_ROOT\MAX.Application.2\Clsid = {7FA22CB1-D26F-11d0-B260-00A0240CEEA3}
HKEY_CLASSES_ROOT\CLSID\{7FA22CB1-D26F-11d0-B260-00A0240CEEA3} =
    OLE Automation MAX 2.0 Application
HKEY_CLASSES_ROOT\CLSID\{7FA22CB1-D26F-11d0-B260-00A0240CEEA3}\ProgID =
    MAX.Application.2
HKEY_CLASSES_ROOT\CLSID\{7FA22CB1-D26F-11d0-B260-00A0240CEEA3}\
    VersionIndependentProgID = MAX.Application
HKEY_CLASSES_ROOT\CLSID\{7FA22CB1-D26F-11d0-B260-00A0240CEEA3}\
    LocalServer32 = e:\3dsmax2.5\3dsmax.exe -U MAXScript
```

A 3D Studio MAX OLE objektum egyedi neve `Max.Application.2`. Minden OLE objektumhoz tartozik egy egyedi azonosító szám, a `CLSID`. Az OLE rendszer ezen a számon hivatkozik az OLE objektumra, és a rendszerleíró adatbázisban a lényegi információt ennek az azonosítonak az ismeretével találhatjuk meg. Programjainkhoz generálható ilyen szám egy, a Microsoft által kiadott segédprogrammal, amely a rendszeridőből, és bizonyos hardverazonosítókból generál egyedi azonosítókódot. `HKEY_CLASSES_ROOT\MAX.Application.2\CLSID\` kulcs alatt található meg a 3D Studio MAX egyedi azonosító száma. Ennek ismeretében a `HKEY_CLASSES_ROOT\CLSID\azonosító szám\` kulcs alatt található meg a további információ az OLE objektumról. Legfontosabb a `LocalServer32` bejegyzés, ami megadja az OLE objektumhoz tartozó futtatható programot, annak elérési útját és paraméterezését.

Az OLE szabvány rendkívül kényelmes, mert mivel az OLE objektumok regisztrációjakor meg kell adni az objektum mögötti programot, így a rendszer egy adott OLE objektumra történő hivatkozás esetén el tudja indítani a megfelelő programot.

A 3D Studio MAX program tartalmaz egy script nyelvet, amit MAXScriptnek neveznek. Ebben a nyelvben minden fontosabb grafikus kezelői felületen kiadható parancsnak megvan a megfelelője. Ezen kívül definiálhatók benne változók, függvények, és megvannak az általában egy script nyelvtől elvárható funkciói. A 3D Studio MAX engedélyezi MAXScript függvények OLE interfészbe való regisztrálását, azaz bármely

MAXScriptben írt függvény meghívható egy másik programból OLE metódushívással. Így létrehozható egy interfész a színterek betöltésére, módosítására és a stimuli képek előállítására.

A 3D Studio MAX OLE interfészének megtervezésekor arra törekedtünk, hogy az interfész minél egyszerűbb és rugalmasabb legyen. Így jutottunk el az alábbi OLE interfészig:

mse_oleinit()

Egyszer meghívandó, az OLE felületen való kapcsolódás után. Inicializálja a globális változókat, beállítja a gyorsító funkciókat.

mse_loadscene <*filename*>

Betölt egy 3D Studio színteret a megadott fájlnév alapján.

mse_setparam <*name*> <*propname*> <*value_string*>

Megváltoztatja a name névvel megadott objektum propname tulajdonságát a value_string ertékre.

mse_renderto bmp_filename <*width*> <*height*>

Létrehozza a színtér width szélességű height magasságú két dimenziós leképezését, és a kapott képet kiírja a bmp_filename nevű fájlba.

execute

Bármilyen egyébb maxscript parancs végrehajtása. Például a `quitmax #noPrompt` parancs kilép a 3D Studio Max programból.

3.2.3 Megvalósított algoritmusok

Mivel egy globális optimalizációs problémához nincs egyértelműen legjobb megoldó módszer, ezért több algoritmust is kipróbáltunk, hogy megtaláljuk a problémához legjobban illeszkedő keresési stratégiát. Ezek a módszerek a STAGE algoritmus, a genetikus algoritmus lokális kereséssel (*GA with RR*), a genetikus algoritmus lokális kereséssel és stagenis operátorral, és a sorsolt pontokból indított lokális keresések.

A programok írásakor felhasználtuk a Neurális Információ Feldolgozási Csoport által készített Stage és Stagenis C++ template osztálykönyvtárat. A Stagenis könyvtár

a Stage könyvtár átírásával keletkezett, tehát a Stage könyvtár ismertetése elegendő a könyvtár alapelveinek megértéséhez. Jelen szakdolgozatnak nem célja a Stage könyvtár ismertetése, viszont célja útmutatót adni a használatához. A Stage könyvtár dokumentációja megtalálható a ?? internetoldalon.

Mivel a Stage könyvtár template könyvtár, ezért ahhoz, hogy használni tudjuk, a keresési problémától függő osztályokat implementálnunk kell.

A legalapvetőbb osztály az állapot (`state`) leírására szolgáló osztály. Erre az osztályra nincs megkötés, bármilyen másik osztálytól származtatható. A keresési tér reprezentálására egy n elemű vektort választottunk, ahol n a keresési tér dimenziószáma. Egy ilyen osztály a `Geonsearch_Console` projekt `stageplugin` alkönyvtárában levő `GeonState.h` és `GeonState.cpp` fájlokban található. Ezentúl minden osztályt ebből a projektből fogunk bemutatni.

A tárgyfüggvény kiszámításához is külön osztályt kell implementálni. Ezt a Stage könyvtárban definiált `AttributeServer` osztályból kell származtatni. Ennek a megoldásnak az előnye, hogy így az osztály tárolni tudja a tárgyfüggvény kiszámításához szükséges additív információkat is. Az `AttributeServer` osztályban egy `getObjective` függvény van definiálva, amit a konkrét feladathoz való illesztés során implementálni kell.

Ennek az osztálynak a feladata a szimulált neuronaktivitás függvény értékének meghatározása. Mivel a optimalizáció kezdetén a 3D Studio MAX-hoz való OLE interfészen keresztüli kapcsolódás után az opimalizáló program betölti a megadott színteret, a szimulált neuronaktivitás függvény értékének meghatározásához arra van szükség, hogy a program

– a keresendő paraméterek értékeit beállítsa az aktuális színtéren,
– meghívja az OLE interfész `mse_renderTo` parancsát, hogy a stimuláló kép elkészüljön,
– a stimuláló képet beolvassa a merevlemezről,
– kiszámítsa a célkép és az aktuális (most beolvasott) kép távolságát.

Az osztály megtalálható a `stageplugin` alkönyvtár `GeonAttrServer.h` fájljában.

Az állapotjellemzők kiszámításához egy a `StageAttributeServer` osztályból származtatott osztályt kell implentálni. A `StageAttributeServer` osztályban egy getFeatu-

re függvény van definiálva, amit a konkrét feladathoz való illesztés során implementálni kell.

Mivel ez az optimalizálási feladat rendkívül összetett, ezért a kutatás kezdeti szakaszában nem foglalkoztunk állapotjellemzők keresésével. A dolgozat írása közben már megtörténtek az első próbálkozások állapotjellemzők keresésére, de ezek az eredmények még nem kerülnek be a dolgozatba, egy későbbi írás tárgyát fogják képezni. Állapotjellemzőként magát az állapotot választottuk, tehát a `getFeature` függvény visszatérési értéke maga a paraméterül kapott állapot. Az osztály megtalálható a stageplugin alkönyvtár `GeonStageAttrServer.h` fájljában.

Ahhoz, hogy a keresési teret be tudjuk járni, szükség van az állapotok egymáshoz való viszonyának megadására. Ehhez implementálni kell a Stage könyvtárban levő `NavigationServer` egy leszármazottját. A `NavigationServer` nem direkt módon definiál szomszédsági struktúrát, hanem az állapotokhoz akciókat rendel, és megad egy metódust, amely egy adott állapotban végrehajt egy adott akciót. Egy ilyen akció lehet például a „lépj jobbra" utasítás reprezentációja, de lehet valami sokkal komplexebb navigációs utasítás reprezentációja is. Ez a módszer magasabb szintű absztrakciós lehetőségeket engedélyez, ezenkívül az akciók végrehajtása tartalmazhat mögöttes hatásokat végrehajtó programkódot. Ugyanúgy érvényes itt is, ami az `AttributeServer` osztály esetén, miszerint így az osztály tárolni tudja a tárgyfüggvény kiszámításához szükséges additív információkat is.

A mi esetünkben a keresési tér egy n dimenziós diszkretizált intervallum. Ennek bejárásához úgy választottuk meg a szomszédsági struktúrát, hogy azon pontokat tekintünk egymás mellettinek, amelyek valamelyik dimenzió irányában szomszédos rácspontokon vannak. A navigációs szerver osztály akcióit egy egész számmal reprezentáljuk. A szám előjele adja meg a lépés irányát, abszolút értéke pedig azt, hogy a lépés melyik dimenzióban történjen. Az osztály megtalálható a `stageplugin` alkönyvtár `GeonNaviServer.h` fájljában.

A 3D Studio kapcsolati modell

Szoftvertervezési szempontból érdemes még bemutatni a `GeonSearch_Console` program és a 3D Studio MAX kommunikációs kacsolati modelljét.

4. Eredmények

5. Diszkusszió

5.1 A genetikus algoritmus értékelése

A numerikus kísérletek eredményei azt tanúsítják, hogy ezen a problémán a genetikus algoritmus nem működik hatékonyan. A ??. oldalon szereplő ??. grafikonon figyelhetjük meg ezt: még a lokális keresésekkel megerősített genetikus algoritmus (*GA with RR*) is gyengén teljesít a többi módszerhez képest.

A GA-technikát a Stagenis algoritmusban is alkalmaztuk. Ezzel kapcsolatban vegyes képet mutatnak a tapasztalatok: bizonyos értelemben jobb a Stagenis a STAGE-nél, más tekintetben nem.

A Stagenis algoritmus a GA-felépítésből adódóan jól párhuzamosítható, szemben a STAGE algoritmussal, amely alapvetően szekvenciális, és nehéz lenne párhuzamosítani. A Stagenis algoritmusban a populáció egyedeinek kiértékelése több egymástól független lokális keresést jelent a jósági függvényen, így ez egy olyan részfeladat, ami minden további nélkül párhuzamosítható. Ezáltal a Stagenis algoritmus szinte annyiszorosára gyorsul, mint a populáció mérete (azért csak „szinte”, mert nem kell mindig az összes egyedet kiértékelni a populációban, hiszen 30%-ukat az előző generációból vesszük át, így őket már kiértékeltük korábban. Továbbá a STAGE függvény-approximátorának tanítása és az értékbecslő optimalizálása nem osztható szét a párhuzamos szálakra, ennek egy központi gépen kell történnie.)

Ha ezt a párhuzamosíthatóságot figyelembe vesszük, akkor a Stagenis algoritmus — bár nem végez kevesebb lépést — időben gyorsabb a STAGE algoritmusnál. Kísérleteinkben 20 elemű populációkat használtunk: ha párhuzamosan valósítanánk meg a Stagenis lokális kereséseit, akkor a Stagenis kb. tízszeresére gyorsulna. Ez azt jelenti, hogy a valódi párhuzamosság megvalósítása révén a grafikonokon a Stagenisnek megfelelő vonal

egy nagyságrenddel balra tudna eltolódni (lásd ??. és ??. grafikonok). Láthatóan ekkor bizonyos esetekben a Stagenis gyorsabbá válik a STAGE algoritmusnál.

Amikor azonban nem áll rendelkezésünkre a párhuzamosítás megvalósításához szükséges hardver vagy szoftver, és ezért nem tudjuk figyelembe venni mindezt, akkor általában a STAGE algoritmus működik hatékonyabban (lásd ??. ábra).

5.2 Párhuzamosan több optimalizáció indítása

Tapasztalataink azt is mutatják, hogy ezen a problémán még a függvény-approximátoros fejlett módszerek is (mint a STAGE, Stagenis) olyan viselkedést produkálnak, ami miatt érdemes egy régi módszerhez folyamodni: többször, egymástól függetlenül párhuzamosan elindítani ugyanazt a keresést.

A STAGE esetében ezt a ??. oldalon látható futásidő-eloszlás grafikonok igazolják. Az ábra alapján konkrét számokkal példával elmagyarázod, hogy miért éri meg többször elindítani egy futtatás helyett.

Másik alátámasztás: a korrelációs grafikonokon nem igazán mutatkozik korreláció. Talán az az oka, hogy a STAGE-nek nem sikerül felhasználnia a múlt tapasztalatait: ha beindul valamin, az nem segíti abban, hogy rátaláljon az optimumhoz vezető útra. Valószínűleg az optimumra csak "véletlenül" bukkan rá. Ez ugyanis pont ilyen futásiő-eloszlást eredményez.

Javaslat: indítsuk el a STAGE-et párhuzamosan több gépen. Érdemes elgondolkodni azon, hogy milyen információ lehetne amit érdemes megosztaniuk egymással a külön futó szálaknak?

Egy ilyen gondolatból született a Stagenis algoritmus is. A Stagenis alg. tk. több párhuzamos STAGE-számítást szimulál, amelyek a fapp-jukat osztják meg egymással. Mivel a fapp közös, nem érdemes mindenkinek a fapp alapján választania újraindító állapotot. A többiek mehetnének egyszerű random választással, ehelyett mi a GA heurisztikáját vettük be a játékba.

Az eredmény, mint az 1. Tézisnél már mondtuk, nem egyértelmű: párhuzamosítás nélkül semmiképpen sem jobb a GAwS, párhuzamosítással viszont gyakran igen.

5.3 Az állapotjellemzők megválasztása

- bevezetés - A tapasztalataink azt mutatják, hogy a STAGE tud nagyon hatékony lenni (lásd satisfiability eredmények). Valószínűleg ez a probléma sem nehezebb annál (hiszen az NP-teljes). Tehát nem valószínű, hogy a problémánk volna extra nehéz, amit a STAGE nem tudna kezelni. Nem a problémával és nem is a STAGE-dzsel van a baj, hanem inkább az állapotjellemzőkkel: nem alakítanak ki a térben szép tendenciákat. (indoklás: látszik az LS-grafikonokból és a futásidő-eloszlásból).

A STAGE heurisztikája szemlélhető az "előítélettel" való hasonlatosság felől is. Ha így nézzük, akkor a saját tapasztalatainkból tudhatjuk, hogy nagyon fontos az állapotjellemzők gondos megválasztása.

Beszéljünk tehát az állapotjellemzők megválasztásáról: - példák - Szükséges feltétel az állapotjellemzők jóságára lineáris és kvadratikus regresszió fapp esetén - Hogyan lehet ellenőrizni ezt a feltételt: egyszerűbben mint futtatni sok STAGE-et aztán nézni a futási eredményeket Javaslat: HA vki feature-t keres, akkor érdemes ezt a tesztet elvégezni, mert kevesebb számolással megvan mint a STAGE kipróbálása.

5.4 Összefoglalás

Javaslat: a jelen probléma megoldására, hatékony alg. találására érdemes az állapotjellemzők tájékán is keresgélni. Hiszen a jelenlegi állapotjellemzők biztosan nem elég jók. Általánosan azonban, más problémáknál, érdemes lehet egy megerősítéses tanulást alkalmazni arra, hogy mit éri meg csinálni jobban: - elkezdeni jobb feature-t keresni, s ezzel tölteni az időt - futtatni többször a STAGE-et a meglévő állapotjellemzőkkel - STAGE helyett más algoritmussal próbálkozni inkább (pl. RR, GA, GAwRR, Stagenis stb.)

6. Jelölések

$\Re$ a valós számok halmaza

$\tilde{\Re}$ valós értékű valószínűségi változók halmaza. Tetszőleges nem üres H halmaz esetén $\tilde{H}$ azoknak a valószínűségi változóknak a halmazát jelöli, amelyek értékkészlete H-nak nem üres részhalmaza.

$\mathcal{X}$ keresési tér, állapottér, a probléma lehetséges megoldásainak halmaza

$\mathcal{Y}$ tulajdonságvektorok tere, $\mathcal{Y} \subseteq \Re^d \quad (0 < d$ egész)

$\mathcal{V}$ az $\mathcal{X} \to \Re$ és $\mathcal{X} \to \tilde{\Re}$ típusú függvények összessége (állapotértékelő függvények). Ha $f \in \mathcal{V}$, akkor az $E(f)$ a következő $\mathcal{X} \to \Re$ függvény:

$$E(f)(x) = \begin{cases} f(x) & \text{ha } f \;\; \mathcal{X} \to \Re \text{ típusú} \\ E(f(x)) & \text{ha } f \;\; \mathcal{X} \to \tilde{\Re} \text{ típusú} \end{cases} \qquad (x \in \mathcal{X})$$

o a jósági függvény, amelyet optimalizálni (minimalizálni vagy maximalizálni) akarunk, $o \in \mathcal{V}$

ω felső korlát a jósági függvény értékére (minimalizálási problémánál alsó korlát), $\omega \in \Re$. Ha a korlát nem ismert, $\omega = \infty$ is lehet.

N szomszédsági struktúra $\mathcal{X}$ fölött, $N : \mathcal{X} \to 2^{\mathcal{X}}$. Valamely $x \in \mathcal{X}$ állapot szomszédainak halmazát $N(x)$ jelöli.

$\mathcal{M}$ mozgási operátorok halmaza, elemei egy-egy mozgási irányt testesítenek meg: $\mathcal{M} = \{\, m_k : \mathcal{X} \to \mathcal{X} \mid m_k$ a k-dik irányba való elmozdulás (lépés) hatásfüggvénye $\}$. A mozgási operátorok szomszédsági struktúrát definiálnak:

$$N(x) = \bigcup_{m \in \mathcal{M}} \begin{cases} \{m(x)\} & \text{ha } x \in D_m \\ \emptyset & \text{különben} \end{cases} \qquad (\forall x \in \mathcal{X})$$

$\mathcal{X}^*$ — állapotsorozatok (az $\mathcal{X}$ elemeiből képzett *véges* sorozatok) halmaza

τ — állapotsorozat, pályagörbe, $\tau = x_1 x_2 \ldots x_{|\tau|} \in \mathcal{X}^*$

$|\tau|$ — a τ sorozat hossza

π — lokális keresési módszer, $\pi : \mathcal{V} \times \mathcal{X} \to \tilde{\mathcal{X}}^*$.
Ha az $o \in \mathcal{V}$ célfüggvényre alkalmazott, $x \in \mathcal{X}$ kezdőállapotból indított π lokális keresés kimenetele a $\tau = x_1 x_2 \ldots x_{|\tau|} \in \mathcal{X}^*$ pályagörbe, azt így jelöljük: $\tau = \pi(o, x)$.

$\xi_i^{\pi(o,x)}$ — valószínűségi változó, értéke az az állapot, amelyben az o-t optimalizáló, x-ből indított π lokális keresési módszer az i-edik lépés megtétele előtt van (vagyis a $\pi(o, x)$ pályagörbék i-edik tagja), $\xi_i^{\pi(o,x)} \in \tilde{\mathcal{X}}$. Így tehát $\pi(o, x) = \xi_1^{\pi(o,x)} \xi_2^{\pi(o,x)} \ldots$ Természetesen $\xi_1^{\pi(o,x)}$ értéke mindig x. A keresés leállását e ξ-változók szempontjából úgy értelmezzük, hogy a pályagörbe minden további állapota a végállapot ($x_{|\tau|}$).

$\mathcal{P}$ — populáció, $\mathcal{P} \subset \mathcal{X}$

Γ — a gének alaphalmaza, tetszőleges véges vagy végtelen szimbólumhalmaz

γ — gén, $\gamma \in \Gamma$

$\vec{\gamma}$ — génsorozat, $\vec{\gamma} = \gamma_1 \gamma_2 \ldots \gamma_{|\vec{\gamma}|}$

$|\vec{\gamma}|$ — génsorozat hossza

i — a populáció egy egyede, génsorozat, $i \in \mathcal{P}$, $i = \gamma_1 \gamma_2 \ldots \gamma_{|i|}$

$|\mathcal{P}|$ — a populáció mérete, az egyedek száma a populációban

F — tulajdonságfüggvény, $F : \mathcal{X} \to \mathcal{Y}$, $F = (F_1, F_2, \ldots, F_d)$.
Az $F_k : \mathcal{X} \to \Re$ komponensfüggvények az állapotjellemzők ($k = 1 \ldots d$).

Φ — függvény-approximátor, $\Phi : \mathcal{Y} \to \Re$

$f(i)$ — az i egyed fitnessze (a genetikus algoritmus számára), illetve az o jósági függvénynek valamely $\tau = \pi(o, i)$ pályagörbén előforduló legjobb értéke (a STAGE algoritmus számára). Formálisan $f \in \mathcal{V}$, $f(i) = \max_{k \in \{1 \ldots |\tau|\}} o(i_k)$, ahol i_k a τ

állapotsorozat k-dik tagját jelöli. (Ha a jósági függvény minimalizálása a cél, akkor max helyett természetesen min áll.) Természetesen az $f(i)$ érték a lokális keresés aktuális τ kimenetelén múlik. Amikor ez nem determinisztikus, akkor $f(i)$ sem az: $f(i) \in \tilde{\Re}$.

f optimális állapotértékelő függvény, $f : \mathcal{X} \to \Re$, $f = E(f)$

$\hat{f}$ értékbecslő függvény, $\hat{f}(x) = \Phi(F(x)) \approx f(x) \qquad (x \in \mathcal{X})$

p_i az i egyed kiválasztásának valószínűsége az új generáció származtatása során

χ a keresztezés-operátor alkalmazásának valószínűsége

μ a mutáció valószínűsége

r véletlen állapotot generáló módszer, $r \in \tilde{\mathcal{X}}$

Irodalom

[1] W. H. Press, S. A. Teukolsky, W. T. Vetterling és B. P. Flannery: „Numerical Recipes in C: the art of scientific programming"
Cambridge University Press, Cambridge, Nagy-Britannia, második kiadás, 1992

[2] D. Beasley, D. R. Bull és R. R. Martin: „An overview of Genetic Algorithms: Part 1, fundamentals"
University Computing, 1993, vol.15., no.2., 58-69. oldal

[3] G. Rudolph: „Convergence analysis of canonical genetic algorithm"
IEEE Transactions on Neural Networks, 1994, vol.5., no.1., 96-101. oldal

[4] J. Tóth Gábor, Kovács Szabolcs és Lőrincz András: „Genetic Algorithm with Alphabet Optimization", *Biological Cibernetics*, 1995

[5] S. Russel és P. Norvig: „Artifical intelligence: A Modern Approach"
Prentice Hall 1995, 111. oldal

[6] W. Zhang: „Reinforcement Learning for Job-Shop Scheduling"
PhD. Dissertation, Oregon State University, 1996

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest: „Algoritmusok"
magyar nyelvű kiadás, Műszaki Könyvkiadó, Budapest, 1997, 662. oldal

[8] B. Selman, H. Kautz és D. McAllester: „Ten challenges in propositional reasoning and search"
In Proceedings of the Fifteenth International Joint Conference on Artifical Intelligence (IJCAI), 1997, 96. oldal

[9] R. Moll, A. Barto, T. Perkins és R. Sutton: „Reinforcement and local search: A case study”
Technical Report UM-CS-1997-044, University of Massachusetts Amherst, Computer Science, October, 1997, 64. és 106. oldal

[10] D. McAllester, H. Kautz, B. Selman: „Evidence for invariants in local search”, *In Proceedings of AAAI-97*, 1997

[11] Richard S. Sutton és Andrew G. Barto: „Reinforcement Learning: An Introduction” *MIT Press* ISBN 0262193981, Cambridge, MA, 1998

[12] J. A. Boyan: „Learning Evaluation Functions for Global Optimization”
School of Computer Science, Carnegie Mellon University Pittsburgh, PA, 1998 CMU-CS-98-152

[13] D. V. Buonomano, M. M. Merzenich: „Cortical Plasticity: From Synapses to Maps” *Annual Reviews Neuroscience, AR050-07* 1998, 21:149-86

[14] Moshe Shipper: „A Brief Introduction To Genetic Algorithms”
http://lslwww.epfl.ch/~moshes/ga.html, 2000

[15] Hévízi György: „Kvantumrendszerek állapot-optimalizációja eredményesség-visszacsatolás segítségével” *SZTE fizikus szak*, Szeged, 2000

[16] *NP-teljes problémák esetén az eloszlás gyakran heavy-tailed*

[17] *Palotai Zsolt satisfyability eredményei*