Algoritmi EVOLUTIVI

Crina Groșan, Mihai Oltean

Începând cu anii '70 s-a manifestat un interes sporit pentru algoritmii care se bazează pe un principiu evolutiv. Un termen comun adoptat care să se refere la tehnicile folosite este acela de metode de calcul evolutiv.

Calculul evolutiv

În general, orice sarcină abstractă care trebuie îndeplinită poate fi privită ca fiind rezolvarea unei probleme, care, la rândul ei, poate fi percepută ca o căutare în spațiul soluțiilor potențiale. Deoarece, de obicei, căutăm cea mai bună soluție, putem privi acest proces ca fiind unul de optimizare. Pentru spații mici, metodele clasice exhaustive sunt suficiente; pentru spații mari, pot fi folosite tehnicile speciale ale inteligenței artificiale. Metodele calculului evolutiv se numără printre aceste tehnici; ele folosesc algoritmi ale căror metode de căutare au ca model câteva fenomene naturale: ***moștenirea genetică*** și ***lupta pentru supraviețuire***. Cele mai cunoscute tehnici din clasa calculului evolutiv sunt algoritmii genetici, strategiile evolutive, programarea genetică și programarea evolutivă. Există și alte sisteme hibride care încorporează diferite proprietăți ale paradigmelor de mai sus; mai mult, structura oricărui algoritm de calcul evolutiv este, în mare măsură, aceeași. Un model ar fi următorul:

```
procedura algoritm_evolutiv
   t ← 0
   creare P(t)
   evaluare P(t)
   cât timp nu condiția de terminare
      t ← t + 1
      selectare P(t) din P(t-1)
      modificare P(t)
      evaluare P(t)
   sfârșit cât timp
sfârșit procedura
```

În cele ce urmează vom explica algoritmul general propus mai sus.

Algoritmii evolutivi mențin o populație $P(t) = \{x_1^t, \ldots, x_n^t\}$ de indivizi la fiecare iterație t. O populație poate fi privită ca fiind un vector de valori. Fiecare individ (element al vectorului) reprezintă o soluție potențială a problemei și este implementată sub forma unei structuri de date S. Un individ este uneori numit și ***cromozom***.

Fiecare soluție este evaluată ca fiind o măsură a "***fitness***-ului" său (***speranței de viață***). Acest *fitness* reprezintă calitatea individului. De obicei, cu cât individul este mai promițător, cu atât *fitness*-ul său este mai mare. Există unele probleme în cazul cărora *fitness*-ul trebuie să fie minimizat.

O nouă populație (iterația $t + 1$) se formează alegând cei mai promițători indivizi (pasul de selecție) din populația curentă. O parte din membri populației nou formate suferă transformări (pasul de modificare) sub acțiunea ***operatorilor genetici***, pentru a obține noi soluții.

Operatorii genetici sunt, de fapt, proceduri care operează asupra elementelor vectorului populație. Există doi operatori genetici principali:

- un operator unar m_i de transformare numit ***mutație***, care creează un nou individ printr-o mică modificare a unui individ ales ($m_i: S \to S$);
- un operator mai puternic c_j numit ***încrucișare***, care creează noi indivizi combinând părți din doi sau mai mulți indivizi ($c_j: S \times \ldots \times S \to S$) (de cele mai multe ori se folosesc doi părinți).

După un anumit număr de generații algoritmul ***converge***: se speră că cel mai promițător individ ajunge la o valoare cât mai apropiată de soluția optimă. În ciuda similarităților puternice între diferitele tehnici de calcul evolutiv, există și multe diferențe. Acestea sunt date, în principal, de structurile de date folosite pentru a reprezenta un individ și de ordinea în care se aplică operatorii genetici. De exemplu, cele două linii din algoritmul de mai sus:

```
selectare P(t) din P(t-1)
modificare P(t)
```

pot apărea în ordine inversă: (în strategiile evolutive, întâi se modifică populația și apoi este formată o nouă populație prin procesul de selecție, în timp ce într-un algoritm genetic întâi se aplică selecția, iar apoi intră în acțiune operatorii genetici de transformare).

Există, de asemenea, și alte diferențe între metode. Una dintre acestea ar fi cea referitoare la metodele de selecție care includ:

- ***selecția proporțională***, unde șansa (probabilitatea) ca un individ să fie selectat este proporțională cu *fitness*-ul lui;
- ***metoda rangului***, în care toți indivizii din populație sunt sortați în funcție de *fitness*, iar probabilitatea (șansa) ca ei să fie selectați este fixată de întreg procesul de evoluție (de exemplu, probabilitatea de selecție a celui mai promițător individ este 0.15, a individului următor este 0.14, etc.; cel mai promițător individ are cea mai mare probabilitate și suma totală a probabilităților indivizilor este 1);
- selecția prin ***turnir*** (prin concurs) unde un anumit număr de indivizi (de obicei doi) luptă pentru a fi selectați în noua generație. Această competiție (***turnir***) este repetată până sunt selectați un număr de indivizi egal cu dimensiunea populației.

Pentru fiecare dintre aceste categorii de selecție există și alte detalii importante. Câteva exemple sunt:

- ***selecția proporțională*** poate necesita folosirea unor metode de trunchiere;
- există diferite moduri pentru stabilirea probabilității în metoda ***rangului***;
- dimensiunea mulțimii alese pentru concurs poate juca un rol semnificativ în metoda selecției prin ***turnir***.

Trecerea de la o generație la alta poate fi efectuată în două variante:

- algoritm *generațional* (noua populație este formată doar din descendenți ai vechii generații);
- algoritm *non-generațional* (în noua populație sunt introduși, de obicei, cei mai promițători indivizi din cele două populații, cea a părinților și cea a descendenților). Este posibil, de asemenea, să creăm puțini (în particular, unul singur) descendenți, care înlocuiesc câțiva (cei mai puțin promițători) indivizi.

Ca o regulă generală trebuie reținut faptul că, în majoritatea cazurilor, este de preferat să se utilizeze un model elitist, care păstrează cei mai promițători indivizi dintr-o generație și îi adaugă automat generației următoare (aceasta înseamnă că, dacă cel mai promițător individ din generația curentă este pierdut datorită selecției sau operatorilor genetici, sistemul forțează apariția lui într-o generație următoare). Un astfel de model este foarte folositor pentru rezolvarea multor probleme de optimizare.

Totuși, structurile de date folosite pentru probleme particulare, împreună cu o mulțime de operatori genetici, constituie componenta esențială a oricărui algoritm evolutiv.

Acestea sunt elementele cheie care ne permit să distingem între variatele paradigme ale metodelor evolutive.

Principalele paradigme ale calculului evolutiv

Există câteva paradigme importante ale tehnicilor de calcul evolutiv. Le vom trata în continuare pe fiecare în parte.

Algoritmi genetici

Începuturile algoritmilor genetici se situează undeva în jurul anului 1950, când mai mulți biologi au folosit calculatoarele pentru simularea sistemelor biologice. Rezultatele muncii au început să apară după 1960, când la Universitatea din *Michigan*, sub directa îndrumare a lui ***John Holland***, algoritmii genetici au apărut în forma în care sunt cunoscuți astăzi.

După cum sugerează și numele, algoritmii genetici folosesc principii din genetica naturală. Câteva principii fundamentale ale geneticii sunt împrumutate și folosite artificial pentru a construi algoritmi de căutare care sunt robuști și cer informații minime despre problemă.

Algoritmii genetici au fost inventați folosind modelul procesului de adaptare. Ei operează, în principal, cu șiruri binare și folosesc un operator de recombinare și unul de mutație.

Prin mutație se schimbă un element (genă) dintr-un cromozom, iar prin încrucișare se schimbă material genetic între doi părinți; dacă părinții sunt reprezentați prin șiruri de cinci biți, de exemplu (0, 0, 0, 0, 0) și (1, 1, 1, 1, 1), încrucișarea celor doi vectori poate duce la obținerea descendenților (0, 0, 1, 1, 1) și (1, 1, 0, 0, 0) (acesta este un exemplu al așa-numitei încrucișări cu un punct de tăietură).

Fitness-ul unui individ este atribuit proporțional cu valoarea funcției criteriu corespunzătoare individului; indivizii sunt selectați pentru generația următoare pe baza *fitness*-ului lor.

Vom ilustra modul de lucru al algoritmilor genetici cu ajutorul unei probleme simple: proiectarea unei cutii de conserve. Considerăm o cutie de conserve cilindrică, cu numai doi parametri: diametrul d și înălțimea h (evident, pot fi considerați și alți parametri, cum ar fi grosimea, proprietăți ale materialului, forma, dar sunt suficienți doar cei doi parametri pentru a ilustra lucrul cu algoritmii genetici).

Să considerăm că această conservă trebuie să aibă un volum de cel puțin 300 ml și obiectivul proiectului este de a minimiza costul materialului folosit la fabricarea conservei. Putem formula problema noastră astfel: să se minimizeze valoarea funcției $f(d,h) = c\left(\frac{\pi d^2}{2} + \pi dh\right)$

unde c reprezintă costul materialului conservei per cm², iar expresia din paranteză reprezintă suprafața conservei.

Funcția f se mai numește și ***funcție criteriu*** (sau ***funcție obiectiv***). Mai trebuie îndeplinită și condiția ca volumul cutiei să fie cel puțin 300 ml și vom formula aceasta astfel:

$$g(d,h) \equiv \frac{\pi d^2 h}{4} \geq 300.$$

Parametrii d și h pot varia între anumite limite:

$$d_{min} \leq d \leq d_{max} \text{ și } h_{min} \leq h \leq h_{max}.$$

Reprezentarea soluției

Primul pas în utilizarea unui algoritm genetic este stabilirea unei codificări a problemei. Codificarea binară este cea mai obișnuită dintre tehnicile de codificare; ea este

simplu de manipulat și conferă robustețe problemei. Reprezentarea binară poate codifica aproape orice situație, iar operatorii nu includ cunoștințe despre domeniul problemei. Este motivul pentru care un algoritm genetic se poate aplica unor probleme foarte diferite. În cazul codificării binare, fiecare valoare se reprezintă printr-un șir de lungime specificată care conține valorile `0` și `1`. În anumite situații este necesar să utilizăm codificarea "naturală" a problemei, în locul reprezentării binare. Un exemplu de codificare naturală ar fi codificarea reală, care utilizează numere reale pentru reprezentare.

Pentru a folosi algoritmii genetici la găsirea unor valori optime pentru parametri *d* și *h*, care să satisfacă condiția prezentată sub forma funcției *g* și care să minimizeze funcția *f*, vom avea în primul rând nevoie de reprezentarea valorilor parametrilor în șiruri binare (vom folosi, așadar, o codificare binară a problemei). Să presupunem că folosim cinci biți pentru a codifica fiecare dintre cei doi parametri *d* și *h*. De exemplu, următorul șir reprezintă o conservă cu diametrul de 8 cm și înălțimea de 10 cm:

$$\underbrace{01000}_{d}\underbrace{01010}_{h}.$$

Pentru aceste valori ale lui *d* și *h*, costul conservei este de 23 de unități.

Marginea inferioară a celor doi parametri este `0`, iar marginea superioară este `31`.

Folosind această reprezentare a parametrilor pe cinci biți, există exact 2^{10} = `1024` soluții posibile. Folosind marginile considerate mai sus, algoritmii genetici ne permit să alegem numai valori din intervalul [`0`, `31`].

Algoritmii genetici nu ne impun numai valori întregi din acest interval; în general, putem folosi orice altă valoare întreagă sau reală, prin schimbarea lungimii șirului binar și a celor două margini: inferioară și superioară.

Atribuirea fitness-ului

Am afirmat anterior că algoritmii genetici lucrează cu șiruri de biți reprezentând valorile parametrilor și nu cu parametrii înșiși. După ce a fost creat un nou șir (o nouă soluție) prin operatori genetici, trebuie să-l evaluăm. În majoritatea cazurilor, *fitness*-ul este chiar valoarea funcției criteriu pentru soluția respectivă. De exemplu, *fitness*-ul conservei reprezentat prin șirul de zece biți este:

`F = 0.0654 · (π(8)² / 2+ π(8)(10)) = 23`,

presupunând că avem `c = 0.0654`.

Dacă obiectivul nostru este de a minimiza funcția criteriu, atunci vom spune că o soluție este mai bună decât alta, dacă *fitness*-ul celei de-a doua este mai mare.

Structura unui algoritm genetic

Vom descrie în continuare structura algoritmilor genetici. Pentru început vom stabili următoarele:

- cromozomii utilizați au lungime constantă;
- populația (generația) $P(t + 1)$ de la momentul $t + 1$ se obține reținând toți descendenții populației $P(t)$ și șterg ând ulterior cromozomii generației precedente ($P(t)$);
- numărul cromozomilor este constant.

Putem prezenta acum structura algoritmului genetic fundamental:

***Pasul* 1:** `t ← 0`.

***Pasul* 2:** Se inițializează aleator populația $P(t)$.

***Pasul* 3:** Se evaluează cromozomii populației $P(t)$. În acest scop se utilizează o funcție de performanță ce depinde de problemă.

***Pasul* 4:** Cât timp nu este îndeplinită condiția de terminare se execută pașii următori:

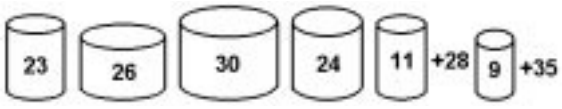
***Pasul* 4.1:** Se selectează cromozomii din $P(t)$ care vor contribui la formarea noii generații. Fie P_1 mulțimea cromozomilor selectați (P_1 reprezintă o populație intermediară).

***Pasul* 4.2:** Se aplică cromozomilor din P_1 operatorii genetici. Cei mai utilizați sunt operatorii de mutație și încrucișare. În funcție de problemă se pot alege și alți operatori (inversiune, reordonare, operatori speciali). Fie P_2 populația astfel obținută (descendenții populației $P(t)$). Se șterg din P_1 părinții descendenților obținuți. Cromozomii rămași în P_1 sunt incluși în populația P_2. Se construiește noua generație, astfel: $P(t + 1) \leftarrow P_2$; se șterg toți cromozomii din $P(t)$; se execută atribuirea $t \leftarrow t + 1$; se evaluează $P(t)$.

Condiția de terminare se referă, de regulă, la atingerea numărului de generații specificate.

Dacă numărul maxim admis de generații este *N*, atunci condiția de oprire este `t > N`.

Se admite că rezultatul algoritmului este dat de cel mai promițător individ din ultima generație. În realitate, nimic nu ne garantează că un individ mai performant nu a fost obținut într-o generație anterioară. De aceea, este normal ca la fiecare pas (la fiecare generație *t*) să reținem cel mai promițător individ care a fost generat până atunci. Acest proces se numește ***elitism***. Revenind la exemplul nostru, să considerăm o populație aleatoare de șase indivizi, fiecare având marcat *fitness*-ul corespunzător.



Două dintre conservele considerate nu au volumul interior de cel puțin 300 ml și, prin urmare, vor fi penalizate cu suma scrisă lângă cutia respectivă. Această penalizare este destul de mare pentru a face ca toate soluțiile inacceptabile să devină mai puțin promițătoare decât oricare dintre soluțiile acceptabile.

Selecția

Un rol important în cadrul unui algoritm genetic îl ocupă operatorul de selecție. Acest operator decide care dintre indivizii unei populații vor putea participa la formarea po-

pulației următoare. Scopul selecției este de a asigura mai multe șanse de reproducere celor mai performanți indivizi dintr-o populație dată. Prin selecție se urmărește maximizarea performanței indivizilor. În continuare vom prezenta succint cele mai importante mecanisme de selecție.

Selecția proporțională
În cazul selecției proporționale, probabilitatea de selecție a unui individ depinde de valoarea performanței acestuia. Să presupunem că avem o mulțime de cromozomi x_1, x_2, …, x_n. Pentru fiecare cromozom x_i vom calcula performanța sa $f(x_i)$. Se impune condiția ca $f(x_i) \geq 0$. Suma performanțelor tuturor cromozomilor din populație va constitui performanța totală și o vom nota cu F. Probabilitatea de selecție p_i a cromozomului x_i este dată de relația:

$$p_i = \frac{f(x_i)}{F} .$$

Selecția bazată pe ordonare
Această modalitate de selecție constă în a calcula (pentru fiecare generație) valorile funcției de *fitness* și de a aranja indivizii în ordinea descrescătoare a acestor valori. Se va atribui fiecărui individ i o probabilitate de selecție p_i care depinde de rangul său în șirul stabilit. Probabilitățile depind acum doar de poziția cromozomului. Cel mai promițător individ are probabilitatea 1.

Selecția prin concurs
Selecția prin concurs sau *selecția turnir* se bazează pe compararea directă a câte doi cromozomi și selectarea celui mai performant. Operațiile implicate sunt următoarele:
- se aleg în mod aleator doi cromozomi;
- se calculează performanțele cromozomilor selectați;
- cromozomul mai performant este selectat (copiat în populația intermediară asupra căreia se aplică operatorii genetici).

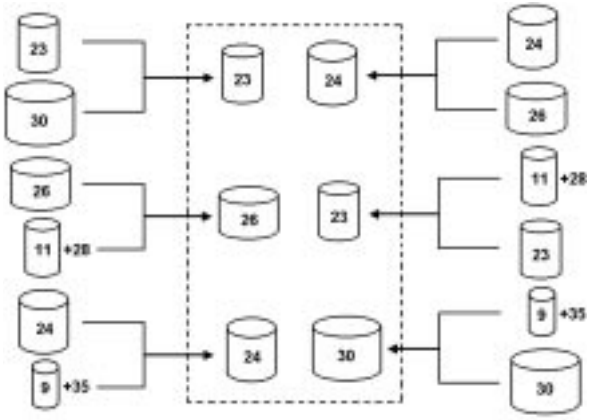
Alte mecanisme de selecție
Un alt tip de selecție este *selecția elitistă*. În acest caz, la fiecare generație se păstrează cel mai promițător sau cei mai promițători indivizi. O altă idee ar fi ca, la fiecare generație, să fie înlocuită doar o parte restrânsă a populației.

Operatorii genetici
Descriem în continuare operatorii genetici folosiți, de obicei, într-un algoritm genetic.

Operatorul de reproducere
Rolul operatorului de reproducere este de a menține soluțiile promițătoare din populație și de a le elimina pe cele mai puțin promițătoare, păstrând constantă dimensiunea populației. Aceasta se realizează astfel:
- se identifică soluțiile promițătoare din populație;
- se creează mai multe copii ale soluțiilor promițătoare;
- se elimină soluțiile mai puțin promițătoare din populație astfel încât multiplele copii ale soluțiilor promițătoare să poată fi plasate în populație.

Există mai multe moduri de a realiza acest lucru. Cele mai uzuale metode sunt selecția proporțională, selecția prin turnir și selecția prin ordonare.

Pentru exemplul nostru, vom folosi selecția prin concurs. Vom ilustra în figura de mai jos modul de selecție.



După cum se poate observa și în figură, avem șase perechi a câte două cutii; dintre perechea de cutii cu *fitness*-ul 23 respectiv 30, o vom alege pe cea cu *fitness*-ul 23 și o vom include în populația intermediară; dintre cutia cu *fitness*-ul 24 și cea cu *fitness*-ul 11 și penalizarea 28, o vom alege pe cea cu *fitness*-ul 24 și o vom include în populația intermediară ș.a.m.d. Astfel, am format o populație intermediară care are tot șase elemente. Este ușor de observat că soluțiile promițătoare au mai mult de o copie în populația intermediară (de exemplu, cutia cu *fitness*-ul 23 și cea cu *fitness*-ul 24 au câte două copii).

Operatorul de încrucișare
Operatorul de încrucișare este aplicat asupra indivizilor din populația intermediară. În exemplul nostru, va fi aplicat asupra reprezentării binare a celor șase elemente pe care le avem în populația intermediară. Operatorul de încrucișare acționează în felul următor: sunt aleși aleator doi indivizi din populația intermediară (care se mai numește și ***piscină de încrucișare***) și anumite porțiuni din cei doi indivizi sunt interschimbate. Operatorul imită încrucișarea intercromozomială naturală. De regulă, se utilizează operatori de încrucișare de tipul (2, 2), adică doi părinți dau naștere la doi descendenți. Încrucișarea realizează un schimb de informație între cei doi părinți. Descendenții obținuți prin încrucișare vor avea caracteristici ale ambilor părinți. Dată fiind importanța majoră a încrucișării, au fost propuse mai multe modele de încrucișare. Vom enumera aici câteva dintre cele utilizate atunci când se folosește codificarea binară.

Încrucișarea cu un punct de tăietură
Fie r lungimea cromozomilor. Un punct de tăietură este un număr întreg $k \in \{1, 2, \ldots, r - 1\}$. Numărul k indică poziția din interiorul cromozomului unde secvența cromozomială se rupe pentru ca segmentele obținute să se recombine cu alte segmente provenite de la alți cromozomi.

Considerăm doi cromozomi:

$x = x_1x_2\ldots x_kx_{k+1}\ldots x_r$ și

$y = y_1y_2\ldots y_ky_{k+1}\ldots y_r$.

În urma recombinării se schimbă între cei doi cromozomi secvențele aflate în dreapta punctului de tăietură *k*. Cromozomii fii vor fi:

$x' = x_1x_2\ldots x_ky_{k+1}\ldots y_r$ și

$y' = y_1y_2\ldots y_kx_{k+1}\ldots x_r$

De exemplu, dacă avem o reprezentare mai sugestivă a celor doi cromozomi:

descendenții vor fi:

Încrucișarea cu mai multe puncte de tăietură

În cazul utilizării mai multor puncte de tăietură, segmentele obținute se combină după o regulă dată. Considerăm încrucișarea cu două puncte de tăietură. Acest tip de încrucișare se realizează conform schemei de mai jos.

Din cromozomii:

vor rezulta doi descendenți de tipul:

În cazul a trei puncte de tăietură, descendenții vor fi de forma:

Revenind la exemplul nostru, vom considera încrucișarea cu un singur punct de tăietură. De exemplu, din încrucișarea a două soluții reprezentate prin cutia care are *fitness*-ul 23, $h = 8$ și $d = 10$, respectiv cutia cu *fitness*-ul 26, $h = 14$ și $d = 6$, vor rezulta doi descendenți care vor avea *fitness*-ul 22, $h = 10$ și $d = 6$, respectiv *fitness*-ul 38, $h = 12$ și $d = 10$ după modelul de mai jos:

Trebuie reținut faptul că încrucișarea nu generează descendenți aleatori. Deși este improbabil ca fiecare încrucișare între două soluții din populație să genereze soluții fii mai promițătoare decât soluțiile părinte, totuși în scurt timp devine clar că șansa de a crea soluții mai promițătoare este mai mare decât în cazul căutării aleatoare. Din încrucișarea cu un singur punct de tăietură a unei perechi de șiruri binare, se pot crea doar două șiruri pereche diferite care vor avea în componență biți combinați din ambii părinți; soluțiile fiu create sunt, probabil, șiruri cel puțin la fel de promițătoare. Prin urmare, nu fiecare încrucișare poate crea soluții la fel de promițătoare, dar nu vor fi mai puțin promițătoare decât părinții. Dacă a fost obținută o soluție mai puțin promițătoare, atunci aceasta nu va mai apărea când se va aplica următorul operator de reproducere și astfel va avea o viață scurtă. Dacă este creată o soluție mai promițătoare, atunci este probabil ca ea să aibă mai multe copii la următoarea aplicare a operatorului de reproducere. Pentru a păstra o astfel de selecție a șirurilor promițătoare în timpul aplicării operatorului de reproducere, nu toate șirurile din populație sunt folosite pentru încrucișare.

Operatorul de mutație

Operatorul de încrucișare este, în principal, responsabil cu aspectul de căutare al algoritmilor genetici, în timp ce operatorul de mutație este folosit pentru alte scopuri. Mutația este cel de-al doilea operator genetic în ordinea importanței și folosirii sale. Efectul acestui operator este schimbarea valorii unei singure poziții dintr-un cromozom. Prin mutație se introduc în populație indivizi care nu ar fi putut fi obținuți prin alte mecanisme.

Operatorul de mutație acționează asupra biților indiferent de poziția lor în cromozom. Fiecare bit al cromozomului poate suferi o mutație. Într-un cromozom pot exista, așadar, mai multe poziții care suferă o mutație.

Mutația este un operator probabilist (adică nu se aplică cu siguranță). Considerăm o populație de *n* indivizi (cromozomi), fiecare având lungimea *r*. Fiecare bit are aceeași probabilitate p_m de a suferi mutația.

Există mai multe variante ale operatorului de mutație. Una dintre ele ar fi ***mutația în forma tare***. În această situație se procedează în felul următor: se generează un număr aleator *q* în intervalul [0, 1). Dacă $q < p_m$, atunci se execută mutația poziției respective schimbând 0 în 1 sau 1 în 0. În caz contrar, poziția respectivă nu se schimbă.

Revenind la exemplul nostru, dacă aplicăm operatorul de mutație unei soluții obținute în urma procesului de încrucișare, și anume soluției care are *fitness*-ul 22, vom obține o altă soluție care va avea *fitness*-ul 16.

Soluția obținută este mai promițătoare decât soluția originală.

În consecință, operatorul de reproducere selectează cele mai promițătoare șiruri, operatorul de încrucișare com-

bină subșiruri din două șiruri promițătoare pentru a forma șiruri mai promițătoare, iar operatorul de mutație schimbă șirurile local, de asemenea, pentru a îmbunătăți soluția.

Strategii evolutive

Strategiile evolutive au fost dezvoltate ca metode de rezolvare pentru problemele de optimizare a parametrilor. Prima strategie evolutivă a fost bazată pe o populație constând dintr-un singur individ. De asemenea, este folosit un singur operator în procesul de evoluție: mutația. Aceasta este în concordanță cu conceptul biologic potrivit căruia modificări mici au loc mai frecvent decât o modificare mare. De obicei această strategie conform căreia un părinte dă naștere prin mutație unui singur descendent este cunoscută sub numele de strategie evolutivă 1 + 1. Felul în care se aplică practic acest algoritm este simplu: se generează o soluție aleatoare pe domeniul de căutare și se efectuează mutații asupra ei. Este acceptat cel mai bun dintre părinte și descendent. Operatorul de mutație se aplică repetat până când se ajunge la soluție.

Un alt tip de strategie este strategia $(\mu + \lambda)$: μ părinți produc λ descendenți. Noua populație (temporară) de $(\mu + \lambda)$ indivizi este redusă din nou - printr-un proces de selecție - la μ indivizi. Pe de altă parte, în strategia (μ, λ), μ indivizi produc λ descendenți ($\lambda > \mu$) și prin procesul de selecție se alege o nouă populație de μ indivizi numai din mulțimea celor λ descendenți. Astfel, viața fiecărui individ este limitată la o generație.

Programare evolutivă

Tehnicile programării evolutive originale au fost dezvoltate de *Lawrence Fogel*. El urmărea o dezvoltare a inteligenței artificiale în sensul dezvoltării abilității de a prezice schimbările într-un mediu înconjurător.

Mediul înconjurător a fost descris ca o secvență de simboluri, iar evoluarea algoritmului presupunea obținerea unui nou produs, și anume a unui nou simbol. Simbolul obținut va maximiza funcția finală care măsoară acuratețea predicției. De exemplu, putem considera o serie de evenimente notate $a_1, a_2, \ldots, a_n$; un algoritm va determina următorul simbol (a_{n+1}), bazându-se pe simbolurile cunoscute $a_1, a_2, \ldots, a_n$.

Ideea care stă la baza programării evolutive este de a evolua un algoritm. Ca și în strategiile evolutive, și în tehnica programării evolutive se creează mai întâi descendenții și apoi se vor selecta indivizii pentru generația următoare. Fiecare părinte produce un singur descendent; deci dimensiunea populației intermediare se dublează (ca în strategia evolutivă (n, n), unde n este dimensiunea populației). Descendentul este creat printr-o mutație aleatoare a părintelui (este posibil să se aplice mai mult de o mutație unui individ). Un număr de indivizi (cei mai promițători) egal cu dimensiunea populației sunt reținuți pentru noua generație. În versiunea originală acest proces este repetat până se obține un nou simbol care este disponibil. După ce s-a obținut un nou simbol, acesta este adăugat listei simbolurilor cunoscute și întregul proces se repetă.

Recent, tehnicile de programare evolutivă au fost folosite pentru rezolvarea problemelor de optimizare numerică precum și în numeroase alte scopuri.

Programarea genetică

O altă abordare interesantă a fost descoperită relativ recent de ***John Koza*** (vezi [5]). *Koza* sugerează că programul dorit va evolua el însuși pe parcursul unui proces de evoluție. Cu alte cuvinte, în loc de a rezolva o problemă și în loc de a construi un program evolutiv care să rezolve problema, vom încerca să găsim un cod sursă care să o rezolve. *Koza* a dezvoltat o nouă metodologie care furnizează un mod de a efectua această căutare.

De exemplu, se dorește obținerea unui program *Pascal* sau *C++* care să rezolve problema drumului *hamiltonian* sau problema ieșirii dintr-un labirint. Deci, nu ne interesează să obținem o soluție pentru un set oarecare de date, ci, mai degrabă, ne interesează să obținem un program sursă care să genereze o soluție corectă pentru orice intrare dată. Cu alte cuvinte, ne interesează să obținem ca rezultat un program asemănător cu cel pe care l-am fi putut scrie noi dacă am fi știut să rezolvăm problema.

Din punct de vedere evolutiv abordarea unor astfel de probleme se face generând o mulțime (populație) aleatoare de coduri sursă care apoi sunt selectate pe baza funcției de *fitness* și evoluate cu ajutorul unor operatori genetici specifici.

În primul rând trebuie să atribuim o funcție de calitate (funcția *fitness*) fiecărui program generat. Această funcție de *fitness* trebuie să reflecte performanțele programului căruia îi este atașată.

De obicei atașarea unei funcții de *fitness* se face rulând programul respectiv și măsurând calitatea soluției în raport cu o soluție care se cunoaște a fi optimă. Un program va avea o calitate mai mare dacă soluția generată va fi mai asemănătoare cu cea a soluției corecte. Nu este grav dacă o soluție optimă nu se cunoaște anterior, deoarece noi dorim să obținem soluții cu un *fitness* cât mai mare (sau cât mai mic).

Evoluarea programelor sursă se realizează prin operatori genetici specifici. De exemplu, un operator de recombinare poate însemna alipirea secvențelor dintr-un cod sursă cu secvențe din alt cod sursă. Un operator de mutație ar putea însemna inserarea de noi instrucțiuni în codul sursă, ștergerea de instrucțiuni, transformarea de instrucțiuni. Evident, în urma aplicării acestor operatori genetici se generează cod sursă care conține greșeli de sintaxă. De asemenea, sunt generate secvențe de cod sursă nefolositoare. Exemple în acest sens sunt secvența de instrucțiuni:

```
i := i + 1;
i := i - 1;
```

sau secvența:

```
a := 0;
b := c / a;
```

De obicei, se evoluează reprezentări mai simple ale programelor de calculator și anume reprezentările arbo-

rescente. Există limbaje de programare (de exemplu *LISP*) în care programele sunt scrise sub forma unor liste ușor transformabile în arbori.

Aplicație

În cele ce urmează vom prezenta rezolvarea unei probleme folosind algoritmii genetici

Enunț (Submulțime de sumă dată)

Se consideră o mulțime *M* de *n* numere și un număr *S*. Să se determine o submulțime a mulțimii *M* care are suma elementelor cât mai apropiată de numărul *S*.

Rezolvare

Determinarea unei submulțimi de sumă dată este o problemă *NP*-completă (vezi [4]). Aceasta înseamnă că nu se știe dacă există sau nu un algoritm de complexitate polinomială pentru rezolvarea acestei probleme. Până în prezent, algoritmii folosiți au complexitate exponențială, iar pentru anumite cazuri particulare au complexitate pseudopolinomială. De exemplu, putem rezolva rezonabil această problemă, dacă datele de intrare îndeplinesc următoarele condiții:

- sunt cel mult 100 de numere naturale;
- suma numerelor nu depășește 500 (mai exact, produsul dintre numărul numerelor și suma acestora nu trebuie să depășească dimensiunea maximă admisă pentru alocarea unei matrice (presupunem că aceasta este alocată static).

Dacă aceste condiții ar fi îndeplinite am putea rezolva ușor această problemă prin metoda programării dinamice, folosind un algoritm de complexitate $O(n \cdot S)$ (vezi [6]).

Însă, dacă numerele nu ar fi întregi ci reale, sau suma lor ar fi mai mare decât 500, sau diferențele între ele ar fi mari etc., atunci algoritmul prin programare dinamică nu mai poate fi folosit. Am enumerat aici doar cazurile importante, dar pot fi imaginate și alte dificultăți.

Din aceste motive vom rezolva această problemă cu ajutorul unui algoritm genetic. Va trebui să găsim o reprezentare a soluției și, de asemenea, o funcție de *fitness*.

Modul în care vom reprezenta soluția ne este dat chiar în enunțul problemei: se cere o submulțime a unei mulțimi *M* cu *n* elemente. Deci, o soluție a problemei este o submulțime. Vom codifica o submulțime printr-un șir de lungime *n* care conține doar valorile 0 și 1. Dacă o poziție *k* va avea valoarea 1, atunci submulțimea respectivă va conține elementul M_k (al *k*-lea element din mulțimea *M*), iar dacă pe poziția *k* este valoarea 0, atunci elementul respectiv nu aparține submulțimii. Această reprezentare a unei submulțimi este specifică tipului `set` din *Turbo Pascal*.

Modul de calcul al *fitness*-ului (calității) unei soluții (submulțimi) este simplu. Calculăm suma elementelor submulțimii, iar *fitness*-ul va fi diferența (în valoare absolută) dintre suma obținută și numărul dat *S*. În aceste condiții *fitness*-ul va trebui minimizat, deoarece noi dorim să determinăm o submulțime pentru care suma elementelor este cât mai apropiată de valoarea dată *S*.

Structura algoritmului genetic propus pentru rezolvarea acestei probleme a fost prezentată mai sus. Vom folosi selecția turnir pentru obținerea populației intermediare. Operatorii genetici folosiți sunt specifici codificării binare (încrucișare cu un singur punct de tăietură, mutație cu probabilitate $p_m = 0.1$).

Codul sursă al implementării este disponibil pentru *download* la www.ginfo.ro/revista/11_8/.

Concluzii și sfaturi practice

În acest articol au fost prezentate principalele direcții ale algoritmilor evolutivi. Aplicațiile practice ale acestor algoritmi sunt nenumărate. Ei sunt folosiți în domenii tot mai neașteptate cum ar fi proiectarea aripilor de avion sau la proiectarea formei stațiilor orbitale. Dacă ați ales să rezolvați o problemă evolutiv, trebuie să țineți cont de câteva sfaturi.

- Pentru a rezolva o problemă cu algoritmi evolutivi trebuie să o transformați mai întâi într-o problemă de optimizare, adică să se minimizeze sau să se maximizeze o valoare (cel mai scurt lanț hamiltonian, cea mai mare componentă intern stabilă etc.).
- Algoritmii evolutivi sunt algoritmi euristici, adică soluția găsită de ei nu este întotdeauna cea mai bună, dar se află într-o vecinătate a soluției optime. Deci, dacă aveți de ales între un algoritm polinomial care rezolvă sigur problema și un algoritm evolutiv, ar fi de preferat să folosiți algoritmul polinomial.
- Algoritmii evolutivi, de obicei, au complexitate polinomială. De aceea ei sunt foarte des utilizați pentru a rezolva problemele dificile (*NP*-complete). Rezultatele obținute sunt foarte apropiate de cele obținute de algoritmii siguri, dar care au rulat mii de ore.
- Dacă problema este complexă folosiți un algoritm genetic și nu o strategie evolutivă. De obicei mutația este un operator de căutare slab, deci, dacă se folosește doar acesta, există șanse mari să se obțină o soluții locale și nu globale.

Bibliografie

1. Beasley D., Bull D.R., Martin R.R., *An Overview of Genetic Algorithms, Part 1, Foundations*, University Computing, Vol.15, No.4, pp. 170-181, 1993;
2. Dumitrescu D., *Algoritmi genetici și strategii evolutive - Aplicații în inteligența artificială și în domenii conexe*, Editura Albastră, Cluj-Napoca, 2000;
3. Goldberg D.E., *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison - Wesley, Reading, MA,1989;
4. Garey M.R., Johnson D.S., *Computers and Intractability: A Guide to NP-completeness*, W.H. Freeman and Company, New York, 1978.
5. Koza J.R., *Genetic Programming*, MIT Press, Cambridge, MA, 1992;
6. Oltean M., *Proiectarea și implementarea algoritmilor*, Computer Libris Agora, Cluj-Napoca, 2000.