CASE STUDY

pre predmet:
Evolučné algoritmy

vypracoval:
Daniel Ferák
MFF UK, Bratislava, 4.ročník

muflon@pobox.sk
www.muflon.miesto.sk

Úloha č. 5:

Generujte magické štvorce pomocou genetického algoritmu. "Magický štvorec" rádu *n* je *n*n* matica obsahujúca všetky celé čísla od 1 po *n*n*, každé z nich práve raz, tak, že súčty všetkých riadkov, stĺpcov a diagonál sú tie isté. Inými slovami, je to permutácia čísel 1 po *n*n*, usporiadaná do pravouhlej mriežky, s vlastnosťou že každý súčet riadku, stĺpca, alebo diagonály matice sa rovná tomu istému celému číslu.

Napríklad, magický štvorec rádu 3 je:

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

Riešenie:

Úloha generovania magických štvorcov sa zaraďuje medzi klasické kombinatorické optimalizačné problémy, medzi ktoré patrí napríklad aj známy problém obchodného cestujúceho (hamiltonovská kružnica v grafe). Úlohu som riešil genetickým algoritmom s črtami charakteristickými pre riešenie kombinatorických optimalizačných problémov.

Najprv trocha teórie:

Genetické algoritmy (pozri [1]) patria medzi stochastické optimalizačné algoritmy, využívajúce silu a princíp evolúcie v živej prírode (tak ako ho popísal Darwin) s ktorou majú mnoho spoločného. Pracujú nad populáciou jedincov (pri vyššej abstrakcii sa za jedinca bude považovať nie množina ale iba jeden chromozóm reprezentovaný binárnym alebo celočíselným vektorom - ďalej teda budem pojmy jedinec, chromozóm a vektor voľne zamieňať). Populácia sa časom (postupnosťou generácií) mení, pričom do ďalšej generácie prechádzajú s väčšou pravdepodobnosťou tí jedinci, ktorí sú spomedzi ostatných najlepšie prispôsobení na „prežitie". V reči genetických algoritmov – majú najväčšiu Fitness – číslo, popisujúce silu jedinca (čím väčšie je toto číslo, tým lepšie). Inými slovami, sú to jedinci, ktorí sa najviac ponášajú na ideál, ktorý buď poznáme, alebo sme ho nejakým spôsobom schopní opísať. Pravidlo výberu schopnejšieho jedinca sa realizuje napríklad pomocou princípu rulety. To znamená, že Fitness (reprezentovanú intervalmi) jednotlivých jedincov usporiadame vedľa seba na priamku medzi body A a B (často je vhodné priradenie A=0 a B=1). Vygenerujeme náhodné číslo z intervalu <A,B> a ďalej pracujeme s jedincom, ktorému bod na priamke „padol" do jeho Fitness. Jedinec môže do ďalšej generácie prejsť bez zmeny, avšak s pravdepodobnosťou P_{repro} môže spolu s nejakým iným jedincom prejsť procesom reprodukcie, ktorý zahŕňa proces kríženia a proces mutácie. Kríženie dvoch chromozómov spočíva vo vygenerovaní bodu kríženia (indexu zložky vektora), a následnom skopírovaní zložiek vektorov do nových chromozómov, pričom zložky týchto nových vektorov sú pred bodom kríženia identické so svojimi „rodičmi" a po bode kríženia sú navzájom vymenené (niektoré genetické algoritmy môžu operovať aj s viacerými bodmi kríženia). Nasleduje proces mutácie nových chromozómov. Algoritmus po zložkách prejde novými vektormi a s veľmi malou pravdepodobnosťou P_{mut} pri každej zložke túto môže zmeniť. V prípade binárnych vektorov sa zložka zamení za novú podľa vzorca

1-**x**

(1)

kde **x** je pôvodná zložka.

Pri kombinatorických optimalizačných problémoch (čo je náš prípad), sa bude postupovať mierne odlišným spôsobom. Vektory totiž budú nie binárne, ale celočíselné. Ďalej bude nutné dodržať podmienku, že vo vektore dĺžky *n* sa každé číslo z 1,...,*n* bude vyskytovať práve raz. Inými slovami, chromozómy budú vlastne permutáciami *n* celých čísiel. Pri takto konštruovaných chromozómoch samozrejme vzorec **(1)** zlyháva, preto bude potrebné vymyslieť nejakú inú operáciu mutácie. Takisto proces kríženia popísaný vyššie nám nezaručuje, že potomkovia pôvodných chromozómov budú opäť permutácie. Budeme teda postupovať nasledovným spôsobom:

Rovnako ako pri mutácii binárneho vektora, budeme po zložkách prechádzať celým chromozómom, avšak s pravdepodobnosťou P_{mut} vymeníme danú zložku so zložkou s indexom *i*, kde *i* bude náhodné celé číslo z 1,..,*n*. Je zrejmé, že takto definovaný operátor mutácie nám zachováva našu požiadavku byť permutáciou. Operátor kríženia bude spočiatku zhodný s operátorom kríženia definovaným pre binárne vektory, avšak bude pridaný takzvaný opravný proces, ktorý nám novovzniknuté chromozómy upraví na permutácie. Jeho princíp vysvetlím až neskôr.

Je vidieť, že operácie kríženia aj mutácie sú oproti klasickým operátorom trocha umelé. Namiesto mutácie jednej zložky vektora sa nám vymieňajú dve zložky. Okrem prostého kríženia je nutné výsledné vektory ešte opravovať.

Nasleduje formálny opis:

Nech *permutačný chromozóm* je vektor dĺžky *k* – permutácia čísiel 1,..,*k*:

$$P = (p_1,\ldots,p_k)$$

Magický štvorec rádu *n* bude reprezentovaný permutačným chromozómom dĺžky *k*, kde *k*=*n***n*.

Bod kríženia je náhodne vygenerované číslo z intervalu <1,*k*>. Bude označovať index zložky permutačného chromozómu.

Operáciu *mutácie* definujem nasledovným algoritmom (v pascalovskom pseudokóde):

Algoritmus 1

```
procedure Mutation(var Ch:TChromosome);
begin
  for i:=1 to k do
    if random<Pmut then
    begin
      j:=random(k)+1;
      v:=Ch[i];
      Ch[i]:=Ch[j];
      Ch[j]:=v;
    end;
end;
```

Operácia **random** generuje náhodné číslo v intervale <0,1>. Premenná **k** obsahuje dĺžku permutačného chromozómu **Ch**. Skúšal som aj nasledovnú verziu operátora mutácie, avšak výsledky neboli presvedčivé, skôr naopak.

Algoritmus 2

```
procedure Mutation2(var Ch:TChromosome);
begin
  for i:=1 to k do
    if random<Pmut then
    begin
      v:=Ch[i];
      if random<0.5 then
      begin
        dec(Ch[i]);
        if Ch[i]=0 then Ch[i]:=2;
        end
        else begin
          inc(Ch[i]);
          if Ch[i]=k+1 then Ch[i]:=k-1;
        end;
      for j:=1 to k do
        if (j<>i) and (Ch[i]=Ch[j]) then Ch[j]:=v;
    end;
end;
```

Idea bola, aby mutácia zložky chromozómu bola jemnejšia, t.j. aby sa táto nevymenila úplne náhodne s nejakým iným číslom (napr. 2 s 10), ale aby sa zmenila len o jednotku (náhodne +1 alebo –1).

Procedúru *kríženia* reprezentuje **Algoritmus 3**. Tu je namieste niekoľko poznámok k procesu opravy skrížených chromozómov. Majme permutačné chromozómy P a Q a bod kríženia c.

$$P = (p_1,\ldots,p_{c-1},p_c,\ldots,p_k)$$
$$Q = (q_1,\ldots,q_{c-1},q_c,\ldots,q_k)$$

Po klasickej operácii kríženia dostaneme nové vektory:

$$P' = (p_1,\ldots,p_{c-1},q_c,\ldots,q_k)$$
$$Q' = (q_1,\ldots,q_{c-1},p_c,\ldots,p_k)$$

Je zrejmé, že tieto chromozómy už nemusia byť permutačné (napr. ak $q_c = p_2$). Je nutné aplikovať proces opravy (alebo tiež čiastočné priradenie - partial matching), ktorý nám „permutačnosť“ obnoví. Z viacerých možností som si vybral nasledovnú (bližšie pozri aj [1] a [3]):

Definujme zobrazenie f_{PQ} tak, že:

$$f_{PQ}(p_i) = 0 \text{ (pre } i<c)$$
$$f_{PQ}(p_i) = q_i \text{ (pre } i>=c)$$

a zobrazenie f_{QP} inverzné ku f_{PQ}. **Algoritmus 3** pre tento účel pracuje s poľami **Fi** a **Fi1**. Externá procedúra **Clear** použitá v tomto algoritme tieto polia vynuluje. Pomocou definovaných zobrazení budeme nové vektory opravovať nasledovným iteračným princípom:

Opravme najprv vektor P'. Ak zložka p_i (pre $0 < i < c$) nepatrí do definičného oboru zobrazenia f_{QP} tak táto zostáva nezmenená. V opačnom prípade je hodnota p_i zhodná s niektorou z hodnôt $q_c,\ldots,q_k$. Bude preto musieť byť nahradená za prvú hodnotu iteračnej schémy $x_{k+1} = f_{QP}(x_k)$, ktorá sa už nenachádza v definičnom obore zobrazenia f_{QP}. Iterácia sa inicializuje priradením $x_0 = p_i$. Obdobne opravíme aj vektor Q', avšak využijeme pritom zobrazenie f_{PQ}. Nasleduje algoritmus riešiaci daný problém. Fragment č. 1 zabezpečuje

klasické kríženie chromozómov, fragment č. 2 zabezpečuje ich opravu horeuvedeným spôsobom.

Algoritmus 3

```
procedure TChromosome.CrossOver(Ch:TChromosome;var
Child1,Child2:TChromosome);

begin
  Clear(Fi);
  Clear(Fi1);

  //fragment c.1

  CrossPoint:=Random(k)+1;

  if CrossPoint>1 then
  begin
    for i:=1 to CrossPoint-1 do
    begin
      Child1.Value[i]:=Value[i];
      Child2.Value[i]:=Ch.Value[i];
    end;
  end;

  for i:=CrossPoint to k do
  begin
    Child1.Value[i]:=Ch.Value[i];
    Fi[Value[i]]:=Ch.Value[i];
    Child2.Value[i]:=Value[i];
    Fi1[Ch.Value[i]]:=Value[i];
  end;

  //fragment c.2

  if CrossPoint>1 then
  begin
    for i:=1 to CrossPoint-1 do
    begin
      j:=Child1.Value[i];
      while Fi1[j]<>0 do j:=Fi1[j];
      Child1.Value[i]:=j;

      j:=Child2.Value[i];
      while Fi[j]<>0 do j:=Fi[j];
      Child2.Value[i]:=j;
    end;
  end;
end;
```

Pri evolučných algoritmoch (obvzlášť pri kombinatorických optimalizačných problémoch) je najdôležitejším elementom návrh procedúry, ktorá ohodnotí daný chromozóm jeho silou – tzv. Fitness funkcia. Poďme sa bližšie pozrieť na zadanú úlohu. Magický štvorec rádu *n* je reprezentovaný chromozómom dĺžky n*n. Najjednoduchší algorimus hľadajúci magický štvorec rádu n by vygeneroval všetky možné vektory dĺžky *n*n* (je ich až (*n*n*)!) a postupným prehľadávaním by našiel vyhovujúci. Zložitosť tohto algoritmu by bola faktoriálna, čo pri väčších hodnotách *n* znamená praktickú neriešiteľnosť.

Môj genetický algoritmus (tak ako každý iný genetický algoritmus) bude pracovať s obmedzenou veľkosťou populácie, napríklad 1000. Počet chromozómov teda nezávisí od n. Ak hľadáme napríklad magický štvorec rádu 5, je počet všetkých rôznych permutačných

chromozómov až 25! = 1.551121004 * 10^{25}, čo je celkom slušné číslo. Je preto potrebné zabezpečiť, aby sa do populácie veľkosti 1000 dostali len naozaj schopní jedinci. Práve na to slúži Fitness funkcia. Môj nápad, ako ohodnotiť silu chromozómov v prípade magických štvorcov bol nasledovný. Algoritmus zistí počty jednotlivých riadkov, stĺpcov a diagonál, v ktorých sa súčty rovnajú nejakému pevnému číslu. Najväčší z týchto počtov, bude určovať silu daného chromozómu. Uvedené realizuje **Algoritmus 4**. Szx určuje veľkosť strany magického štvorca (rád).

Algoritmus 4

```
function TChromosome.Fitness:integer;
begin
  Clear(Counts);
  Clear(XSums);
  Clear(YSums);
  DSums[1]:=0;
  DSums[2]:=0;

  for y:=1 to szx do
  begin
    for x:=1 to szx do
    begin
      YSums[x]:=YSums[x]+Value[(y-1)*szx+x];
      XSums[y]:=XSums[y]+Value[(y-1)*szx+x];
      if x=y then DSums[1]:=DSums[1]+Value[(y-1)*szx+x];
      if y=szx-x+1 then DSums[2]:=DSums[2]+Value[(y-1)*szx+x];
    end;
  end;

  for x:=1 to szx do
  begin
    inc(Counts[XSums[x]]);
    inc(Counts[YSums[x]]);
  end;
  inc(Counts[DSums[1]]);
  inc(Counts[DSums[2]]);

  result:=1;
  for x:=1 to CountsMax do
  begin
    if Counts[x]>result then result:=Counts[x];
  end;
end;
```

Ako už bolo povedané, **Algoritmus 4** vypočíta sumy jednotlivých riadkov, stĺpcov a diagonál magického štvorca. Tieto uloží do polí **XSums**, **YSums** a **DSums**. Pole **Counts** nazačiatku inicializované na samé nuly, bude obsahovať koľko riadkov, stĺpcov a diagonál má súčet rovný jednotlivým indexom poľa (využil som princíp CountingSortu – pozri [2]). To znamená, že napríklad Counts[10] obsahuje počet riadkov, stĺpcov a diagonál (spolu) ktorých súčet dáva číslo 10. Vo finálnej fáze algoritmus uloží najväčší element poľa Counts do výsledkovej premennej **result**. Toto je vlastne Fitness daného chromozómu. Pri experimentovaní som skúšal výslednú hodnotu umocniť na druhú resp. tretiu, avšak popisovacia sila algoritmu sa strácala.

Samotný genetický algoritmus realizuje nasledujúca procedúra:

Algoritmus 5

```
procedure Evolution(var AOpt:TChromosome);
begin
  StopCriterion:=false;
```

```
  GenerateRandomPopulationOfChromosomes;

  t:=0;
  while (t<NumberOfCycles) and (not StopCriterion) do
  begin
    t:=t+1;

    ComputeFitnessesForAllChromosomesInPopulation;

    TestStopCriterion;

    NextPopulationSize:=0;
    NextPopulation:={};
    while NextPopulationSize<PopulationSize do
    begin
      ChooseTwoChromsomesByRoulette(Ch1,Ch2);

      if Random<PRepro then
      begin
        Reproduction(Ch1,Ch2,NewCh1,NewCh2);
      end
      else begin NewCh1:=Ch1; NewCh2:=Ch2; end;
      NextPopulation:=NextPopulation ∪ {NewCh1,NewCh2};
      inc(NextPopulationSize,2);
    end;
    Population:=NextPopulation;
  end;
  AOpt:=BestChromosomeFromPopulation;
end;
```

Implementácia:

Súčasťou tohto dokumentu je aplikácia nazvaná Magic, naprogramovaná v prostredí Borland Delphi 5 (platforma Windows 9x). Priložené sú aj zdrojové kódy. Aplikácia využíva prostriedky objektového programovania. Trieda **TVector** zapuzdruje celočíselný vektor s veľkosťou **Size** a hodnotami jeho zložiek v poli **Value,** spoločne so štandardnými procedúrami na prácu s vektormi. Trieda **TChromosome**, potomok triedy **TVector**, zapuzdruje permutačný chromozóm definovaný vyššie spolu s procedúrami kríženia (**CrossOver**), mutácie (**Mutation**) a ohodnotenia jeho sily (**Fitness**). Trieda **TPopulation** zapuzdruje populáciu chromozómov. Jednotlivé chromozómy sú uložené v poli **Chromosomes**. Procedúra **ComputeFitnesses** vypočíta sily všetkých chromozómov a upravené ich uloží do poľa **Fitnesses**. Procedúra **RouletteWheel** vracia index do poľa **Chromosomes**, pričom s väčšou pravdepodobnosťou vracia indexy chromozómov s väčšou silou. Procedúra **Reproduction** implementuje operátor reprodukcie a procedúra **Evolution** samotný genetický algoritmus nad danou populáciou.

Aplikácia pracuje s konštantami, ktoré je potrebné nastaviť pred samotným spustením. Tu je ich zoznam:

PopulationSize – počet chromozómov v populácii. Štandardne som pracoval s veľkosťou 1500.

NumberOfCycles – maximálny počet generácií, cez ktoré má populácia prejsť.

PRepro – pravdepodobnosť reprodukcie chromozómov. Empiricky sa mi podarilo zistiť, že optimálne vyzerá byť číslo 0.6.

PMut – pravdepodobnosť mutácie. Tu je naozaj potrebné veľké množstvo experimentovania. Avšak v zásade platí, čím väčšie je toto číslo, tým sa algoritmus stáva náhodnejší a stráca

črty charakteristické pre genetické algoritmy. Štandardne som pracoval v intervale 0.05 – 0.15.

VectorSize – rád magického štvorca ktorý hľadáme, umocnený na druhú.

Po spustení programu sa objaví okno. Tlačidlo **Start** spustí evolučný proces. Na formulár sa postupne vykresľuje graf. Zelenou farbou je vyznačená sila optimálneho jedinca. Červenou farbou – minimálna sila (rovná jednej). Čiernou farbou je označená sila najsilnejšieho jedinca v populácii aký bol zatiaľ nájdený (jeho hodnota sa súčasne vykresľuje do okna v podobe magického štvorca) – algoritmus pracuje s elitizmom (viď [1]). Olivovou farbou sa vykresľuje momentálne najsilnejší jedinec v populácii. Pri väčších hodnotách **PopulationSize** sa tieto dve farby zväčša prekrývajú. Tlačidlo **Stop** slúži na predčasné zastavenie evolúcie. Algoritmus po nájdení magického štvorca zadaného rádu automaticky zastaví a upozorní zvukovým signálom.

Výsledky:

Tu sú jednotlivé magické štvorce, ktoré sa mi podarilo nájsť:

Magický štvorec rádu 4

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

Súčty všetkých riadkov, stĺpcov a diagonál sú rovné číslu 34. Je zaujímavé, že sa mi podarilo nájsť veľa rôznych magických štvorcov rádu 4, ale každého súčet bol vždy 34. (Preklopené a transponované štvorce považujeme za rovnaké. Takže magický štvorec rádu 3 existuje práve jeden, ale magických štvorcov rádu 4 už existuje viac.)

Magický štvorec rádu 5

```
 3  6 23 25  8
 4 22 11 21  7
20  9 12  5 19
24 10  2 13 16
14 18 17  1 15
```

Súčty sú rovné číslu 65.

Magický štvorec rádu 6 sa mi už bohužiaľ nepodarilo nájsť. Avšak našiel som jeho blízkeho „kolegu“, ktorý mal okrem jedného riadku a jedného stĺpca všade rovnaké súčty, a to 101. V treťom stĺpci a štvrtom riadku boli však súčty rovné číslu 161. Tu je:

```
19  9 21 22 19 17
25  6 32  8 16 14
 1 30 34 11  7 18
31  2 24 35 33 36 •
20 28 23 15  3 12
 5 26 27 10 29  4
      •
```

Bodkou sú označené nevyhovujúce súčty.

Záver:

Genetický algoritmus, ktorý hľadá magické štvorce sa mi podarilo naprogramovať, je však otázne, či by sa nedala navrhnúť lepšia Fitness funkcia, ktorá by pomohla v rovnakom

čase nájsť väčšie magické štvorce. Bohužiaľ sa mi nepodarilo na internete nájsť doteraz najväčšie nájdené magické štvorce.

Literatúra:

[1] V. Kvasnička, J.Pospíchal, P. Tiňo: *Evolučné algoritmy*, Vydavateľsvo STU, Bratislava, 2000.

[2] J. Procházka: *Algoritmy a dátové štruktúry*, Vysokoškolské skriptá - MFF UK Bratislava, 1998.

[3] D.E. Goldberg: *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison Wesley, Reading, MA 1989.