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

$$1 - \mathbf{x} \qquad \textbf{(1)}$$

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 a definície:

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*.

Článok magického štvorca, je riadok, stĺpec alebo diagonála magického štvorca.

Magické číslo magického štvorca rádu *n* je číslo, ktorému sa budú rovnať súčty jeho jednotlivých článkov. Toto číslo sa dá vyjadriť explicitne vzorcom:

$$x / n \qquad \textbf{(2)}$$

kde *n* je rád magického štvorca a *x* je suma postupnosti 1,2,…,*k*:

$$x = 1 + 2 + \ldots + k$$

a táto je podľa známeho vzťahu zhodná s:

$$x = k * (k + 1) / 2$$

Vzorec **(2)** sa teda dá prepísať do tvaru:

$$(k * (k + 1)) / (2 * n) \qquad \textbf{(3)}$$

Dôkaz platnosti **(2)** resp. **(3)** sa dá previesť jednoducho sporom (pozrieme sa napr. na stĺpce štvorca): Nech magické číslo magického štvorca rádu *n* je napr. *x* / *n* + 1. Keďže súčet súčtov čísiel v jednotlivých stĺpcoch magického štvorca je vždy pevný a to *x*, tak by nutne v niektorom zo stĺpcov musel byť súčet *x* / *n* – 1 (ináč by sa muselo nejaké číslo v magickom štvorci opakovať, čo je v rozpore s definíciou). Ale toto je v spore s tým, že súčet čísiel v jednotlivých stĺpcoch je rovnaký.

Bod kríženia je náhodne vygenerované celé číslo z postupnosti 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 reálne číslo z intervalu <0,1). Operácia **random(k)** generuje náhodné celé číslo z 0, …, **k** - 1. Premenná **k** obsahuje dĺžku permutačného chromozómu **Ch**. Skúšal som aj nasledovnú (treba uviesť, že pomalšiu) verziu operátora mutácie, ktorá sa ukázala byť výhodná najmä v spojení s pricípom 2.) počítania sily chromozómu (viď. ďalej).

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, …, 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;
```

Skúšal som rôzne obmeny operátora kríženia (napr. aby sa bod kríženia generoval ako násobok rádu hľadaného švorca, a aby sa krížili – vymieňali iba celé riadky krížených štvorcov, obdobne so stĺpcami), avšak efektivita algoritmu v priemere vzrástla iba mierne. Dokonca som skúšal operátor kríženia aj úplne vylúčiť (algoritmus tak pripomínal niečo ako horolezecký algoritmus [1]), ale tu bolo vidieť rapídne zníženie efektivity.

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 nebude závisieť od vstupného parametra 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. Mal som viac nápadov, ako ohodnotiť silu chromozómov v prípade magických štvorcov. Ako najúspešnejšie sa ukázali nasledovné dva princípy:

1.) Algoritmus zistí počty jednotlivých článkov, v ktorých sa súčty rovnajú nejakému pevnému číslu. Najväčší z týchto počtov, si zapamätá do premennej **max**. Ďalej zistí počet tých článkov, ktorých súčet dáva magické číslo štvorca. Výsledná sila chromozómu bude daná súčtom týchto dvoch zistených hodnôt. Uvedené realizuje **Algoritmus 4**. **Szx** určuje veľkosť strany magického štvorca (rád).

Algoritmus 4

```
function TChromosome.Fitness:integer;
begin
  Clear(Counts);
  Clear(XSums);
  Clear(YSums);
  Clear(DSums);

  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]]);

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

Ako už bolo povedané, **Algoritmus 4** vypočíta sumy jednotlivých riadkov, stĺpcov a diagonál štvorca. Tieto uloží do polí **XSums**, **YSums** a **DSums**. Pole **Counts**, nazačiatku inicializované na samé nuly, bude obsahovať koľko článkov má súčet rovný jednotlivým indexom poľa (využil som princíp CountingSortu – pozri [2]). To znamená, že napríklad **Counts[10]** obsahuje počet všetkých tých článkov spolu, ktorých súčet dáva číslo 10. Vo finálnej fáze algoritmus uloží najväčší element poľa **Counts** do premennej **max**. Túto sčíta s hodnotou **Counts[MagicNumber]** (kde sa nachádza počet tých článkov, ktorých súčet dáva magické číslo) a priradí ju do výsledkovej premennej **result**. Toto je vlastne Fitness daného chromozómu. Tu je nutné uviesť, že keby sme ako silu chromozómu uvažovali len **Counts[MagicNumber]**, tak by algoritmus bol nepružný.

2.) Algoritmus zistí sumu odchýlok súčtov v jednotlivých článkoch štvorca od magického čísla a pripočíta ju k premennej **res**, ktorá je nazačiatku inicializovaná na 1. Výsledná sila chromozómu bude určená podielom 1 / **res**. Čiže magický štvorec bude mať Fitness rovnú jednej. Tento princíp sa ukázal byť efektívnejší ako 1.)

Algoritmus 5

```
function TChromosome.Fitness:Double;
begin
  Clear(XSums);
  Clear(YSums);
  Clear(DSums);

  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;

  res:=1;
  for x:=1 to szx do
  begin
    res:=res+abs(XSums[x]-MagicNumber);
    res:=res+abs(YSums[x]-MagicNumber);
  end;
  res:=res+abs(DSums[1]-MagicNumber);
  res:=res+abs(DSums[2]-MagicNumber);

  result:=1/res;
end;
```

Samotný genetický algoritmus realizuje nasledujúca procedúra:

Algoritmus 6

```
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 v rozmedzí 100 až 5000 jedincov. Pri vyšších číslach bol už algoritmus veľmi pomalý, a nazdávam sa, že na hľadanie magických štvorcov vyšších rádov, už zmeny veľkosti populácie rádovo v tisícoch a dokonca ani v desaťtisícoch nemajú výrazný vplyv.

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

PRepro – pravdepodobnosť reprodukcie chromozómov.

PMut – pravdepodobnosť mutácie.

MagicVal – rád magického štvorca ktorý hľadáme.

Nasledujúce dve konštanty sú závislé na predchádzajúcej a nie je ich preto nutné nastavovať (sú zadefinované ako funkcie nad **MagicVal**).

VectorSize – rád magického štvorca umocnený na druhú (veľkosť chromozómov).

MagicNumber – magické číslo magického štvorca.

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
```

Magické číslo je 34. Magických štvorcov rádu 4 už existuje viac (preklopené a transponované štvorce pokladáme za rovnaké – takže magický štvorec rádu 3 existuje práve 1).

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
```

Magické číslo je 65.

Magický štvorec rádu 6

```
 6 30  3 24 12 36
29 14 34 11 22  1
35 32  2 18  7 17
 8 21 26 25 16 15
28  4 19 20 31  9
 5 10 27 13 23 33
```

Magické číslo je 111.

Magický štvorec rádu 7

```
42  9 30 43 28 11 12
 7 35 17 32  1 46 37
25 23 14  2 21 49 41
47 29 22  6 27  8 36
 4 15 18 33 44 48 13
24 45 40 20 38  3  5
26 19 34 39 16 10 31
```

Magické číslo je 175.

Magický štvorec rádu 8

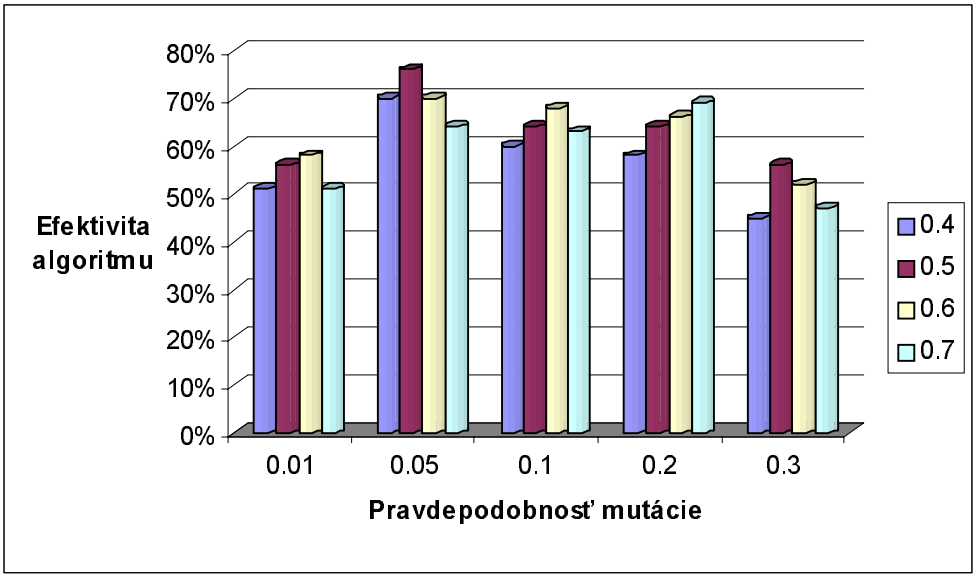
```
62 60 47  1 16 18 23 33
19 15 24 51 46 17 57 31
58 39 52 28  6  4 38 35
42 40  5 53 32 56  3 29
27 12  8 43 41 50 30 49
20  7 44 45 63 13 59  9
11 26 55  2 34 54 14 64
21 61 25 37 22 48 36 10
```

Magické číslo je 260.

Vplyv parametrov na efektivitu algoritmu:

Závislosť efektivity algoritmu od nastavenia konštánt pravdepodobnosti mutácie a pravdepodobnosti reprodukcie som sa rozhodol testovať nasledujúcim spôsobom. Na populácii veľkosti 5000 som nechal „bežať“ **x**-krát (zvolil som **x** = 50) genetický algoritmus, korý pracoval najviac 100 generácií a hľadal magický štvorec rádu 4. Vždy keď sa mu ho podarilo nájsť, zvýšilo sa počítadlo úspešnosti **n** o 1. Výsledná efektivita algoritmu bola daná podielom **n** / **x**. Operátor mutácie bol realizovaný **Algoritmom 2** a sila chromozómu **Algoritmom 5**. Výsledky pre nastavenia jednotlivých parametrov prezentuje **Graf 1, Graf 2** a **Tabuľka 1**.

Graf 1



Prvý stĺpec (fialová farba) z jednotlivých kolekcií stĺpcov označuje efektivitu algoritmu pri pravdepodobnosti reprodukcie 0.4, pričom prvá kolekcia označuje pravdepodobnosť mutácie 0.01. Obdobne druhý stĺpec z prvej kolekcie označuje efektivitu algoritmu pri pravdepodobnosti reprodukcie 0.5 a pravdepodobnosti mutácie 0.01, atď. Z grafu je vidno, že najväčšia efektivita algoritmu bola dosiahnutá pri týchto parametroch: **PMut** = 0.05, **PRepro** = 0.5 a to 76%.

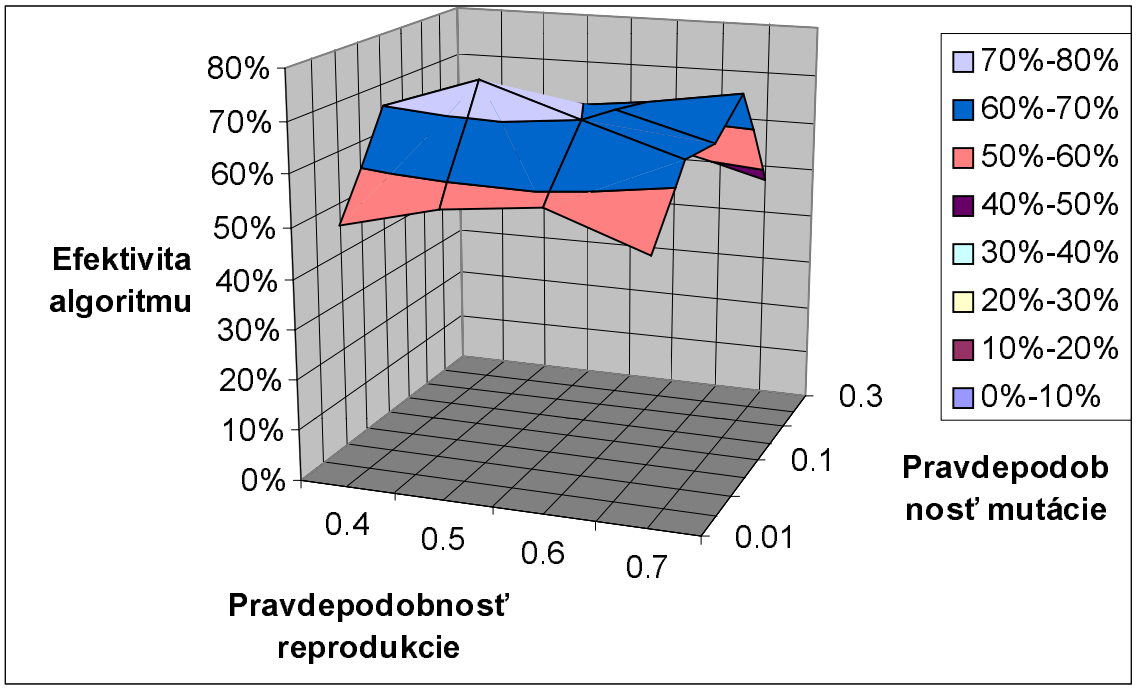
Tu sú konkrétne namerané hodnoty:

Tabuľka 1

	0.4	0.5	0.6	0.7
0.01	51%	56%	58%	51%
0.05	70%	76%	70%	64%
0.1	60%	64%	68%	63%
0.2	58%	64%	66%	69%
0.3	45%	56%	52%	47%

Rozloženie v priestore ukazuje nasledujúci graf:

Graf 2



Je vidieť, že keby sme zvyšovali pravdepodobnosť reprodukcie nad 0.7, prípadne ju znižovali pod 0.4, efektivita algoritmu prudko klesá. Takisto pravdepodobnosť mutácie, väčšia ako 0.3, robí algoritmus príliš náhodný, čo následne znižuje jeho efektivitu. Pravdepodobnosť mutácie, nižšia ako 0.01 zasa algoritmus príliš spomaluje.

Záver:

Podarilo sa mi nájsť magický štvorec rádu 8. Je vidieť, že použitie genetického algoritmu na daný problém malo svoj efekt, veď všetkých možných permutácií čísiel 1, …, 64 je až 64! = 1.268869322 * 10^{89}. Najväčší doteraz nájdený magický štvorec, hľadaný genetickým algoritmom bol rádu 10 (zdroj Internet).

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.