# Evolučné Algoritmy

**Simulácia konvergencie genetického algoritmu.**

**Peter Satury, 5inf (5satury@st.fmph.uniba.sk, peter.satury@corinex.sk)**

1 ZADANIE "CASE STUDY"

2 KOMENTÁR K ZADANIU

3 OBJASNENIE PODSTATY GENETICKÉHO ALGORITMU

4 KONKRÉTNE VYSVETLENIE POUŽITÉHO ALGORITMU

5 VÝSLEDKY ALGORITMU ZNÁZORNENÉ GRAFMI A KOMENTÁR

6 ZÁVER

7 KÓD ALGORITMU

## 1 Zadanie "case study"

Zadania boli zverejnené na internetovskej stránke. Ja som si vybral tému od Profesora D. Goldberga, ktorá skúma konvergenciu genetického algoritmu na základe viacerých parametrov.

(This case study is taken from the home page of Prof. D. Goldberg). Simulate the convergence of a genetic algorithm with population sizes ***n* = 25, 50, 100, 200** members under **proportionate selection** (i.e. a probability of selection is proportional to fitness) with simple replacement (no other operators like mutation and/or crossover are involved). The population should be initialized with one individual with fitness $f = f_{max}$ and the remaining individuals with fitness **f = 1**. Take $f_{max}$ **= 1.5, 3, 9**. Observe the time (in generations) required for the population to converge to n **-** 1 members of higher fitness. Repeat each run a minimum of **3 times** with different random seeds. Repeat for **tournament selection** (with replacement). For the first set of runs use **pairwise tournament selection (s =2)** with $p_{winner} = 1$ (i.e. winner with a greater functional value is selected deterministically for the next generation) and run for all fitness cases. Thereafter run experiments with ***s* = 1.5, 3, 9**, using only $f_{max}$ **= 1.5**.

Blišie sa môžete s prácou Profesora D. Goldberga zoznámiť na internetovskej stánke: http://www.engr.edu/OCEE/webcourses/ge485.

## 2 Komentár k zadaniu

Pokúsim sa vysvetliť niektoré pojmy, ktoré by nemuseli byť jasné:

**Proportionate selection**: Je spôsob výberu jedinca do ďalšej generácie na základe náhodného výberu, pričom ale pravdepodobnosť výberu je priamo úmerná veľkosti fitness tohto jedinca. Preto je vlastá pravdepodobnosť výberu jedinca do ďalšej generácie rovná podielu fitness tohto jedinca a sumy fitness vsetkých jedincov v tejto generácii.

**Simple replacement**: Táto podmienka obmedzuje možnosti použité v genetickom algoritme iba na kopírovanie jedincov, pričom ich fitness ostáva nezmenená. Preto nie je umožnené kríženie a ani mutácie jedincov.

Pri každej simulácii je úvodná generácia zložená z (n – 1) prvkov s fitness 1 a s jedným prvkom s fitness $f_{max}$.

Mojou úlohou bude určiť počet krokov potrebých na to, aby buď všetky prvky alebo aspoň všetky okrem

jedného mali fitness rovné $f_{max}$. Simuláciu mám zopakovať aspoň tri krát pre každé hodnoty atribútov.

**Tournament selection**: Je ďalší spôsob výberu prvku do ďalšej generácie založený na úplne náhodnom vybratí **s** prvkov a potom vyberiem do ďalšej generácie prvok s najvyššou fitness. Toto ale platí iba pre $p_{winner}$ rovné jednej. $p_{winner}$ je vlastne pravdepodobnosť s akou dám do ďalšej generácie prvok s najvyššou fitness. Špeciálny prípad pre **s = 1,5** budem riešiť tak, že budem striedať tournament selection pre **s = 2** a úplne náhodný výber prvku do ďalšej generácie.

Experiment budem postupne spúšťať pre **n = 25, 50, 100 a 200**. Pričom budem ešte meniť aj parameter **s = 0** (to znamená bez použitia **tournamen selection), 2 , 3 , 9 a 1,5** a parameter $f_{max}$ **= 1,5 , 3 , 9**.

# 3 Objasnenie podstaty genetického algoritmu

Genetický algoritmus je založený na simulovaní vývoja skupiny jedincov, ktorá sa mení na základe dopredu daných pravidiel. Základnými predpokladmi pre simuláciu genetického algoritmu je reprezentovanie generácie jedincov.

Počet jedincov sa môže počas simulácie meniť, alebo môže byť konštantný, ako je to v našom prípade. Potom si musíme reprezentovať jedica, najlepšie pomocou nejkého kódovania. Je vítané, aby jednotlivé nezávislé vlastnosti mali samostatné kódovanie a aby boli zakódované iba jedným znakom.

Potom simulujeme evolúciu pre našu skupinu jedincov. Princíp je vo vytvorení nasledujúcej generácie jedincov z danej generácie. Vo všeobecnosti musí existovať funkcia, ktorá určí fitness jedinca na základe jeho vlastností zapísaných v kóde. Podľa tejto fitness väčšinou vyberieme jedinca do ďalšej generácie. Spôsobov je veľa, viaceré sú dokonca použité v tomto experimente.

Iný spôsob tvorby jedinca do ďalšej generácie je kríženie resp. mutácia jedincov resp. jedinca. Vo všeobecnosti je dopredu daná pravdepodobnosť mutácie, kríženia a kopírovania. Mutácie a kríženie používa kód jedinca, čo je vlastne dôvod, prečo je dôležité vhodné kódovanie vlastností jedinca.

Algoritmus skončí v prípade, že všetci jedinci majúrovnaký kód. Ale nie vždy sa do tohoto stavu musí dostať. Výsledkom algoritmu je vlastne jedinec resp. jeho kód, ktorý má vysokú hodnotu fitness. Preto je dôležité aj správne napísanie funkcie, ktorá z kódu jedinca určí jeho fitness.

# 4 Konkrétne vysvetlenie použitého algoritmu

Ako programovací jazyk pre algoritmus som si vybral JAVU, pretože v tomto jazyku momentálne najviac pracujem a algoritmus nie je náročný na implementáciu. Zdrojový kód je súčasťou tohoto textu.

Najprv som si zadefinoval globálne premennné n, s a Fmax.

Použil som pomocné polia p a pnew, v ktorých som si ukladal jednotlivé generácie prvkov resp. ich hodnoty fitness. Ptom som použil pomocné pole c, ktoré som použil pri tournament selection na uloženie jednotlivých prvkov, ktoré boli vybrané do tournamentu.

Ďalej používam pomocnú prmennú krok, v ktorej počítam počet krokov algoritmu, teda vlastne počet nových generácií.

Teraz popíšem jednotlivé metódy použité v algoritme:

**public Evol_alg()** :

Toto je základná časť programu, ktorá volá jednotlivé metódy. Vnútri tejto časti prebieha cyklus while, v ktorom sa vytvára nova generácia a testuje sa, či počet prvkov s hodnotou fitness Fmax dosiahol n – 1. Zároveň zapisuje výsledky do filu.

**public void log(String c, boolean x)** :

Táto metóda inicializuje zapisovanie do filu vystup.txt. Povoluje zapisovat String c na koniec súboru, pričom podľa hodnoty x určí, či sa má zapísať celý riadok, alebo iba časť textu.

**public void writegen()** :

Táto metóda vypíše počet prvkov s hodnotuo fitness Fmax v danej generácii.

**public void init()** :
Táto metóda načíta do poľa p úvodnú generáciu. Prvému prvku priradí hodnotu Fmax a ostatným hodnotu 1. Načíta iba n prvkov.

**public int number_fmax()** :
Táto metóda vráti počet prvkov v poli p, ktorých hodnota fitness je rovná Fmax.

**public void nextGen()** :
Táto metóda zapíše do poľa p novú generáciu prvkov, pričom použije pomocné pole pnew. Rozhoduje sa podľa s, či požije proportionate selection v prípade, že s je rovné nule, alebo tournament selection ak je s nenulové. Zvlášť rieši prípad pre s rovné 1,5 a to tak, že do polovice poľa p dá prvky podľa náhodného výberu a do druhej polovice podľa tournament selection pre s rovné dvom.

**public double suma()** :
Táto metóda vracia sumu fitness všetkých prvkov poľa p.

**public int find_one_roulete()** :
Táto mtóda vracia poradové číslo prvku v poli p, ktorý bol vybratý podľa spôsobu v proportionate selection tzv. Roulette algoritmus.

**public int find_any()** :
Táto metóda vráti poradové číslo prvku v poli p, ktorý bol vybratý na základe náhodného výberu.

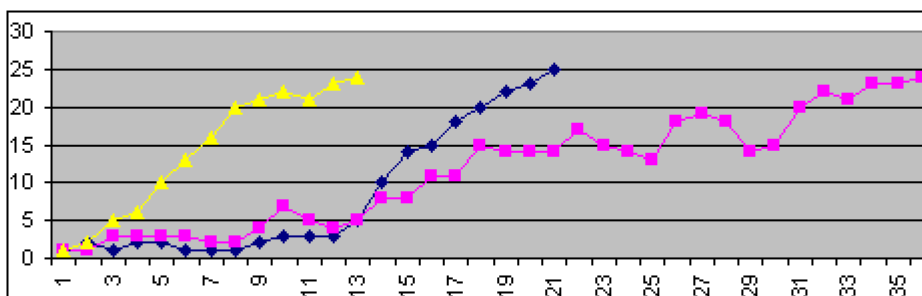
**public int find_next(double ss)** :
Táto metóda vráti poradové číslo prvku v poli p, ktorý "vyhral" tournament podľa tournament slection algoritmu.

# 5 Výsledky algoritmu znázornené grafmi a komentár

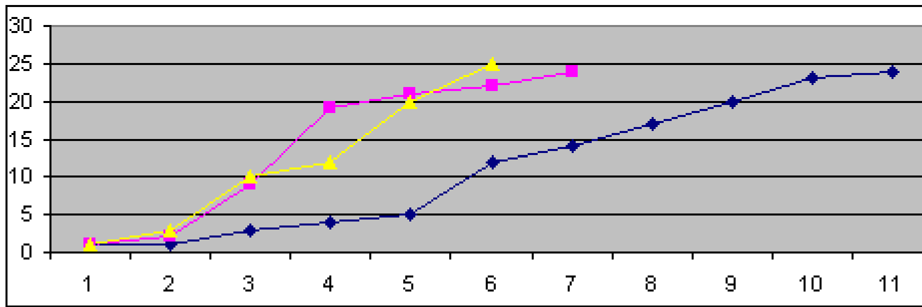
Jednotlivé výsledky som usporiadal podľa hodnoty **s** ( **poradie: 0 (proportionate selection), 2, 3, 9, 1.5** ), potom podľa počtu prvkov v populácii čiže **n** ( **poradie: 25, 50, 100, 200** ) a nakoniec podľa hodnoty $f_{max}$ ( **poradie: 1.5, 3, 9** ).

## Proportionate selection

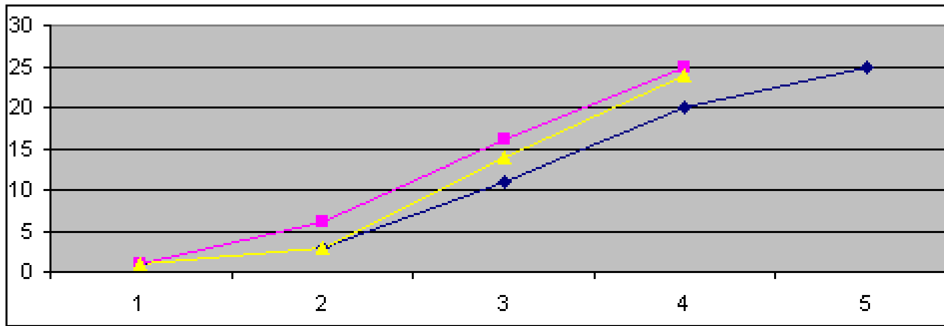
**n = 25 , s = 0 ,** $f_{max}$ **= 1.5**




**n = 25 , s = 0 ,** $f_{max}$ **= 3**

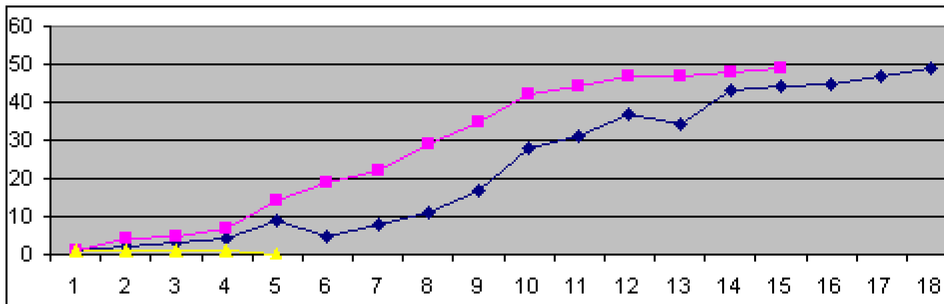



**n = 25 , s = 0 , $f_{max}$ = 9**

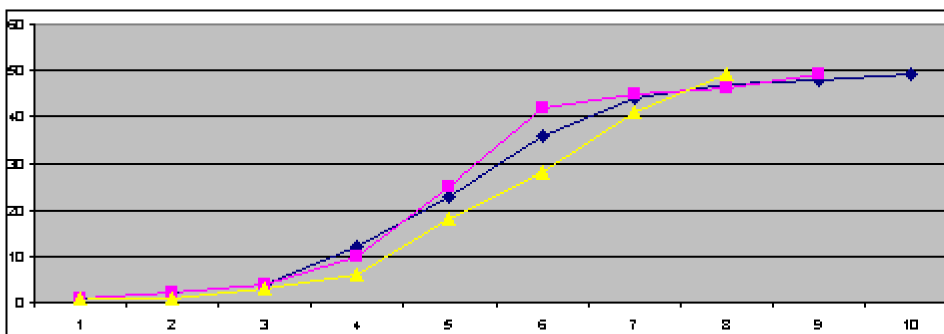



Ako je vidno z grafov, zvyšovanie $f_{max}$ veľmi urýchlilo konvergovanie algoritmu, pretože pravdepodobnosť výberu prvku s vyšším fitness sa veľmi zvyšovala.

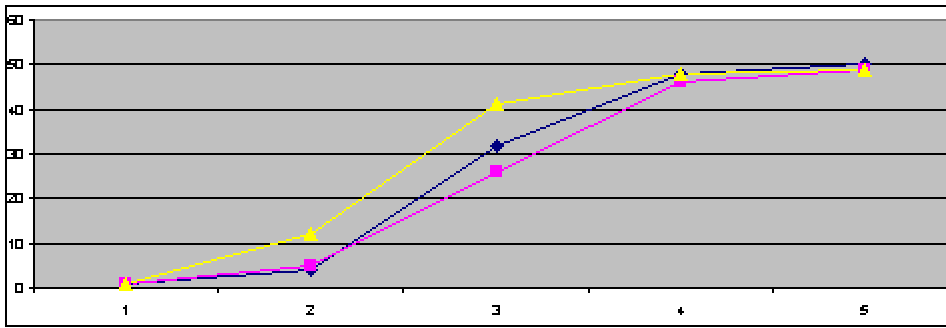
**n = 50, s = 0 , $f_{max}$ = 1.5**




**n = 50 , s = 0 , $f_{max}$ = 3**

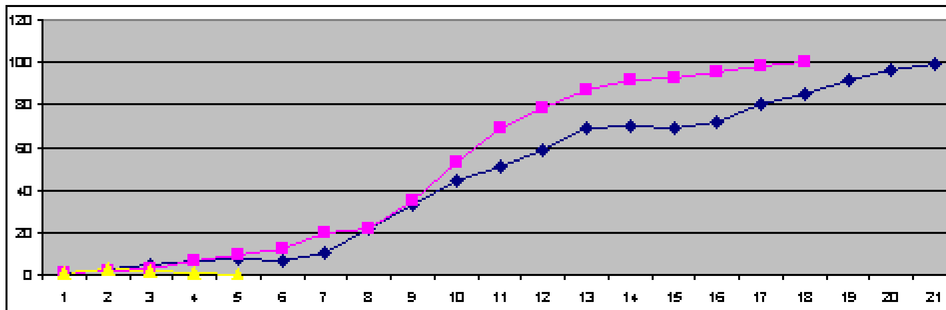



**n = 50 , s = 0 , $f_{max}$ = 9**

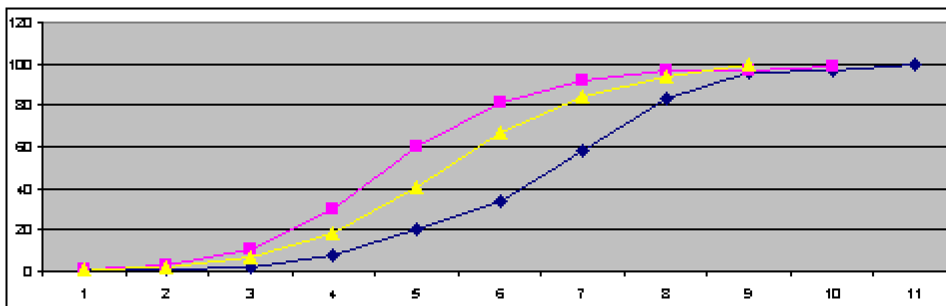



Podobne aj pri n = 50 vidno vplyv $f_{max}$ na konvergenciu. Dokonca v prípade $f_{max}$ = 1,5 algoritmus skončil "neúspechom", keď v piatej generácii mali všetky prvky fitness 1.

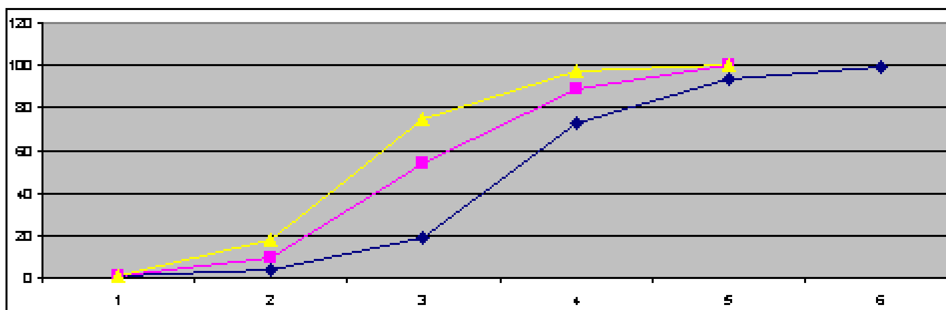
**n = 100 , s = 0 , $f_{max}$ = 1.5**



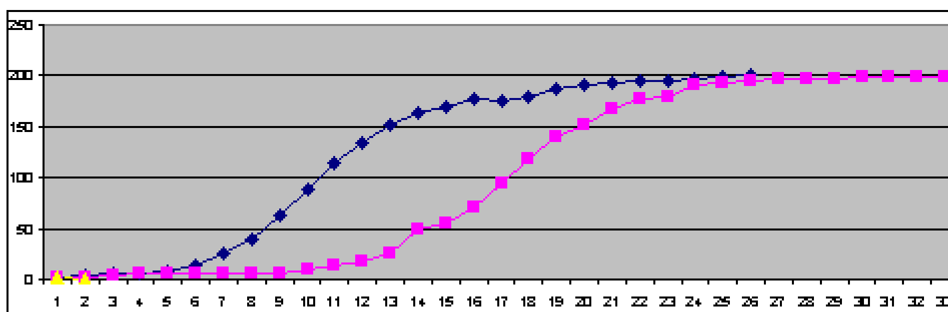

**n = 100 , s = 0 , $f_{max}$ = 3**



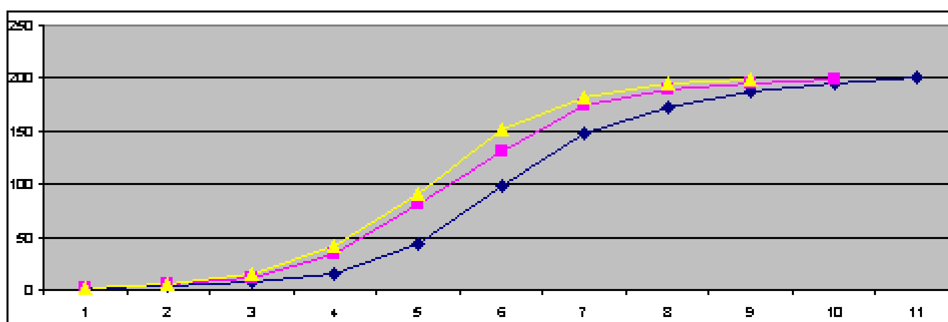

**n = 100 , s = 0 , $f_{max}$ = 9**



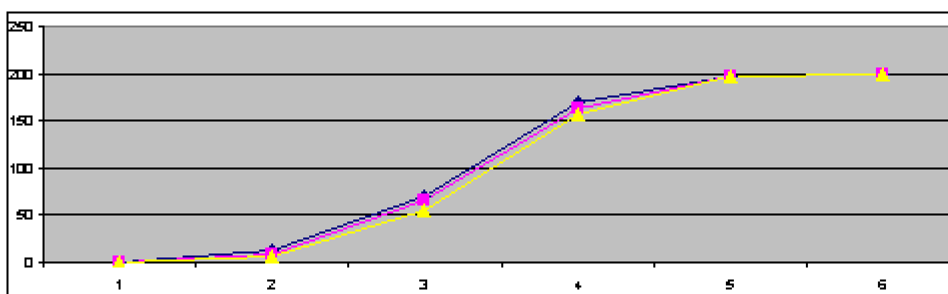

**n = 200 , s = 0 , $f_{max}$ = 1.5**




**n = 200 , s = 0 , $f_{max}$ = 3**



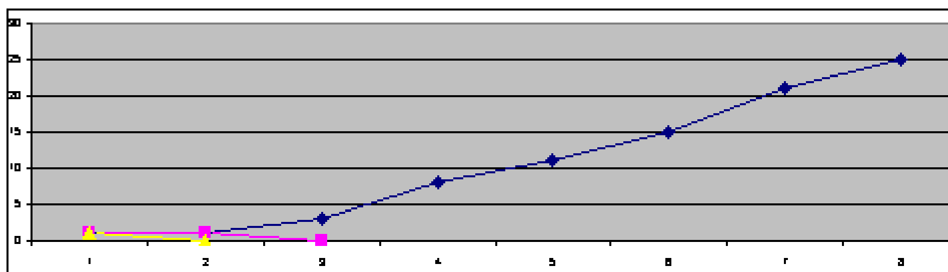

**n = 200 , s = 0 , $f_{max}$ = 9**



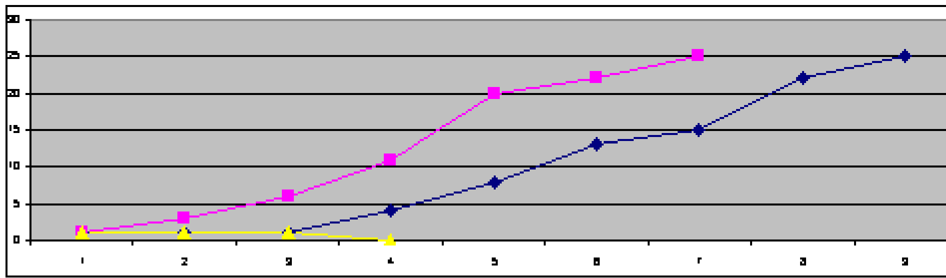

Rovnaké závery môžeme vidie aj pre n = 100 a n = 200. Takže je zrejmé, že pri proportionate selection je hlavným faktorom urýchlenia konvergencie genetického algoritmu $f_{max}$ a až rádovo rozdielne n podstatnejšie vplýva na rýchlosť konvergencie.

## Tournament selection

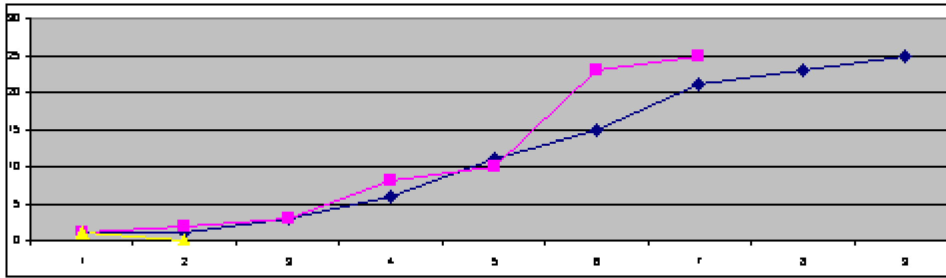
**n = 25 , s = 2 , $f_{max}$ = 1.5**



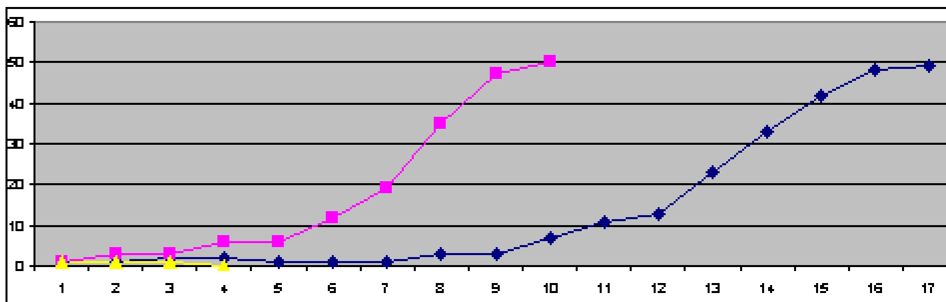
**n = 25 , s = 2 , $f_{max}$ = 3**



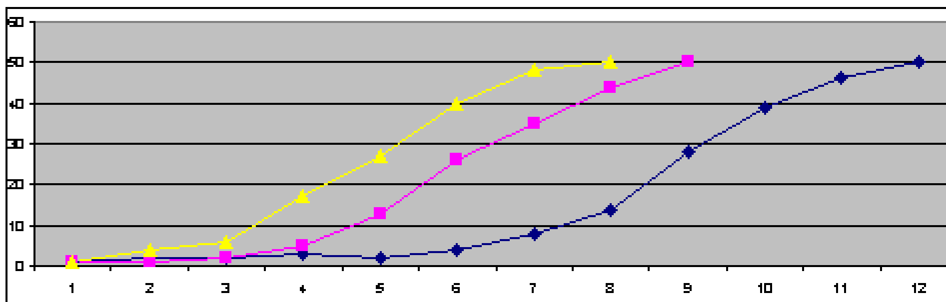
**n = 25 , s = 2 , $f_{max}$ = 9**



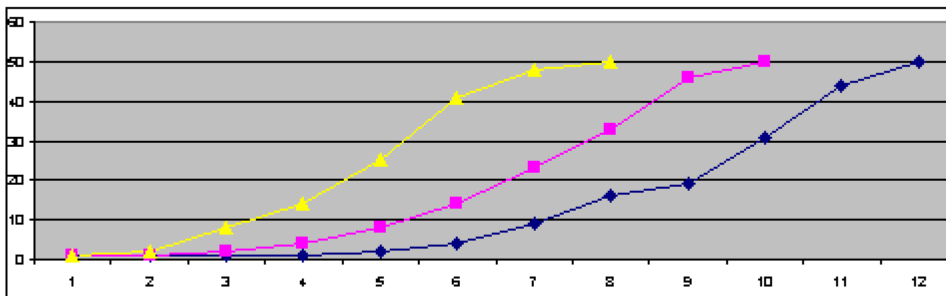
**n = 50 , s = 2 , $f_{max}$ = 1.5**



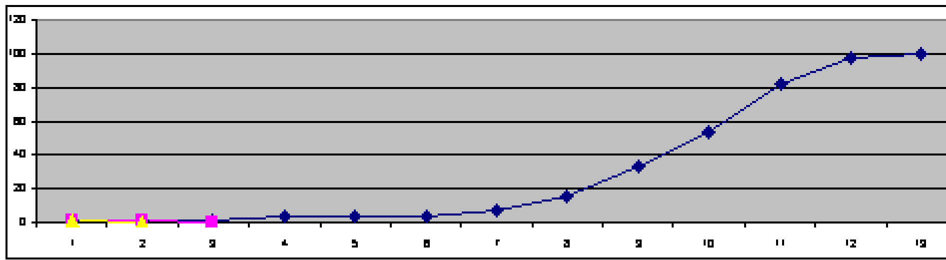

**n = 50 , s = 2 , $f_{max}$ = 3**



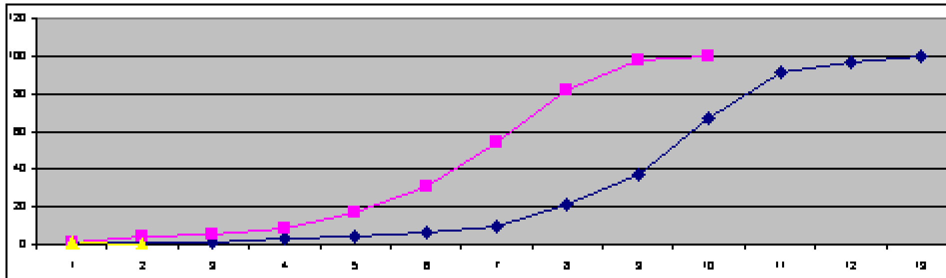

**n = 50 , s = 2 , $f_{max}$ = 9**



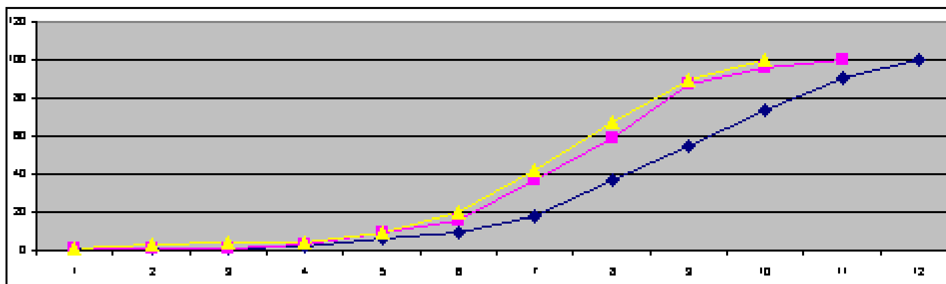

**n = 100 , s = 2 , $f_{max}$ = 1.5**



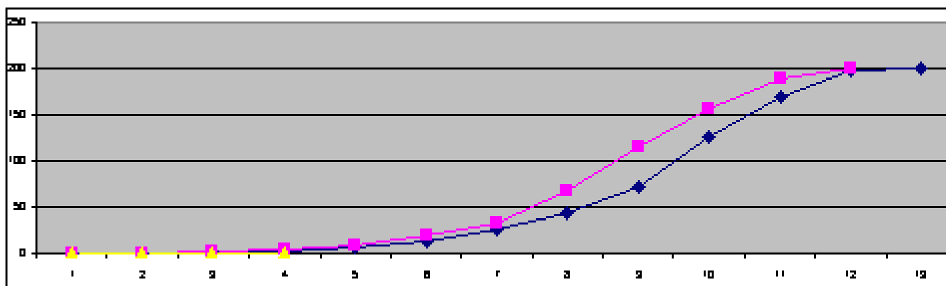
**n = 100 , s = 2 ,** $f_{max}$ **= 3**



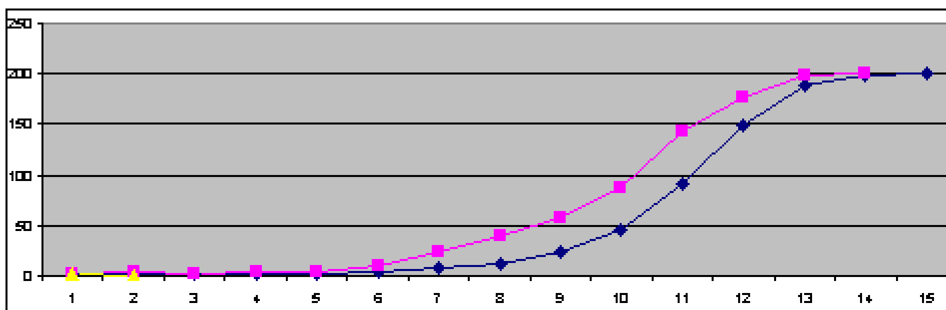
**n = 100 , s = 2 ,** $f_{max}$ **= 9**



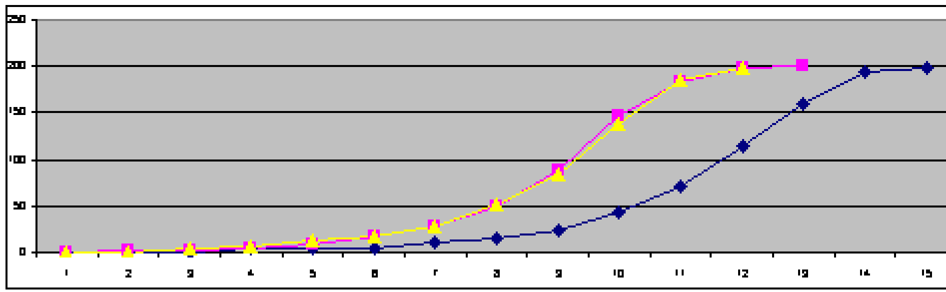
**n = 200 , s = 2 ,** $f_{max}$ **= 1.5**



**n = 200 , s = 2 ,** $f_{max}$ **= 3**

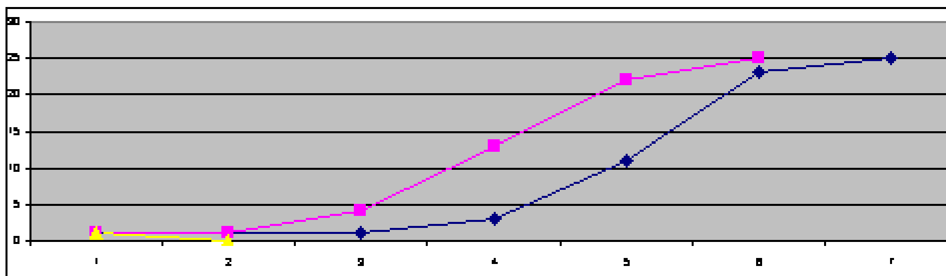



**n = 200 , s = 2 ,** $f_{max}$ **= 9**

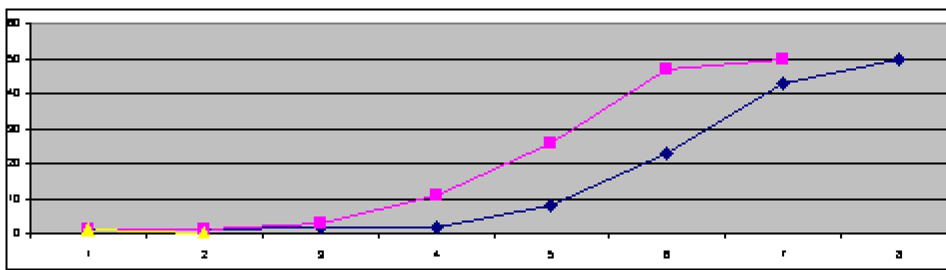


Pri tornament selection je zrejmé, že $f_{max}$ nemá žiadny vplyv na rýchlosť konvergencie. Premenná n má vplyv iba minimálny, takže ho vidieť iba pri veľkých rozdieloch.

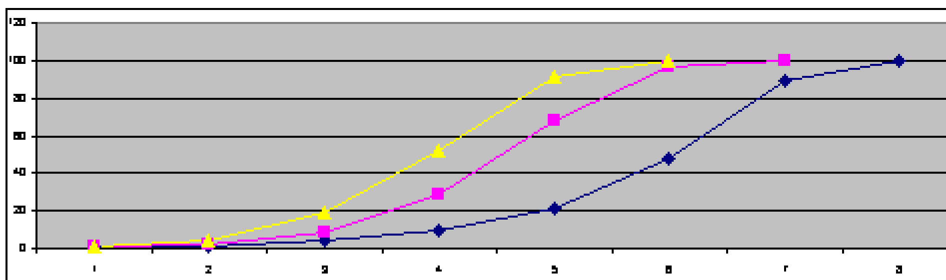
**n = 25 , s = 3 , $f_{max}$ = 1.5**



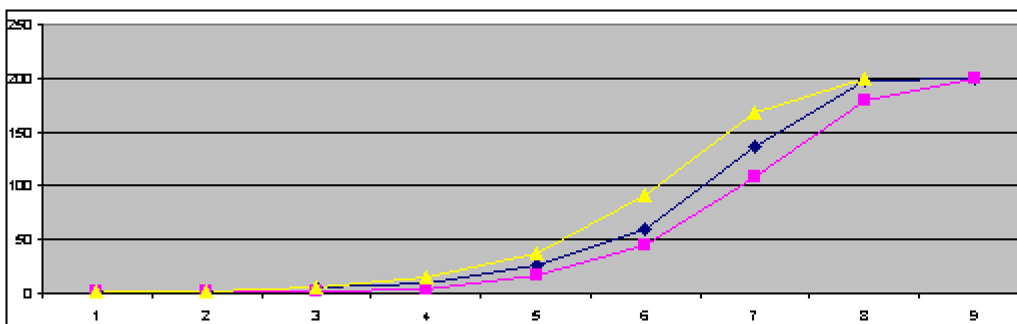
**n = 50 , s = 3 , $f_{max}$ = 1.5**



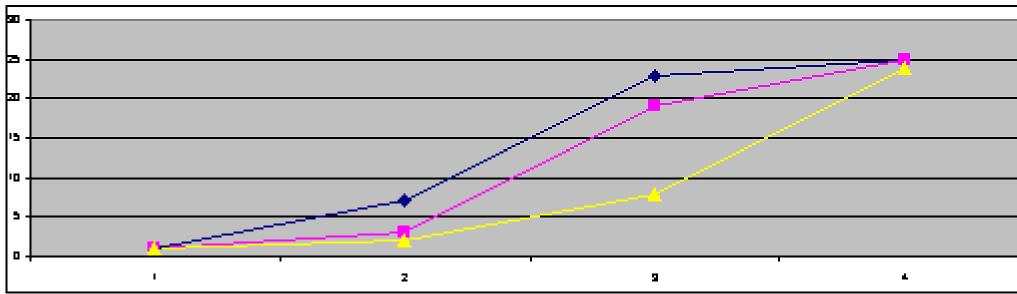
**n = 100 , s = 3 , $f_{max}$ = 1.5**



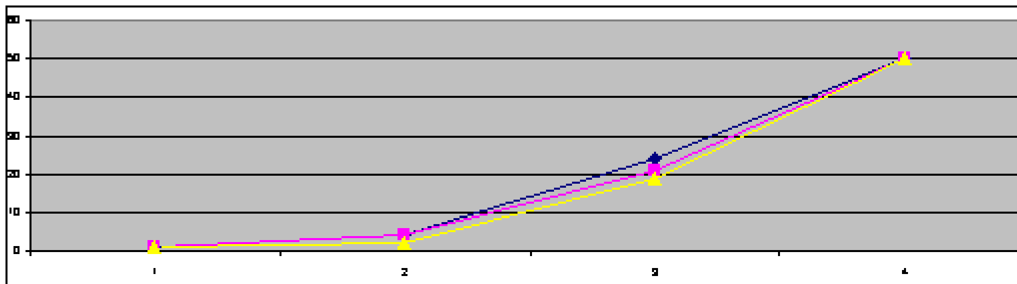
**n = 200 , s = 3 , $f_{max}$ = 1.5**



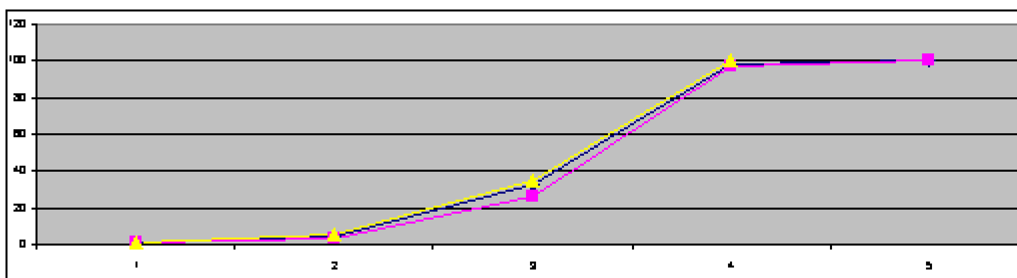

**n = 25 , s = 9 , $f_{max}$ = 1.5**



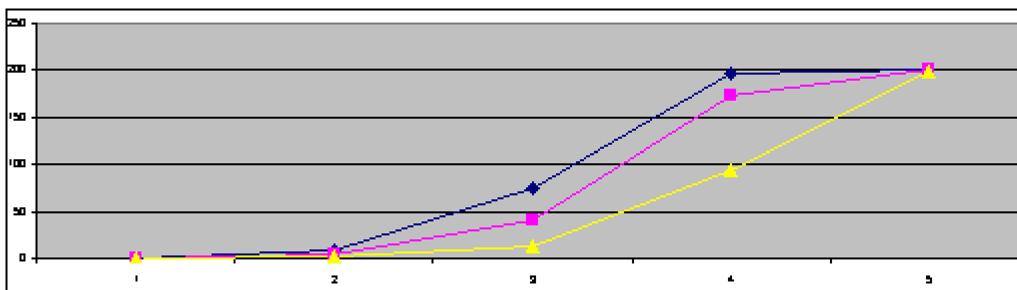
**n = 50 , s = 9 , $f_{max}$ = 1.5**



**n = 100 , s = 9 , $f_{max}$ = 1.5**

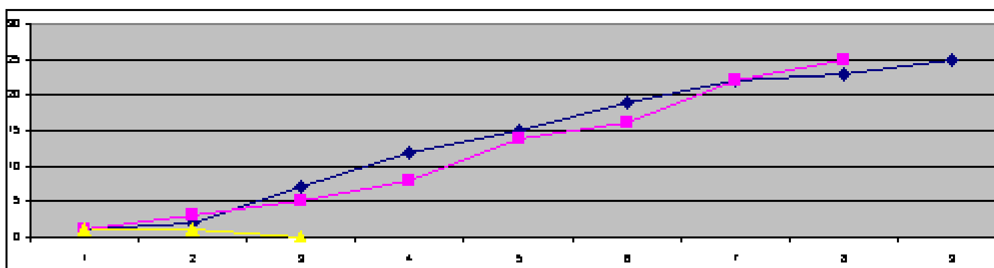


**n = 200 , s = 9 , $f_{max}$ = 1.5**

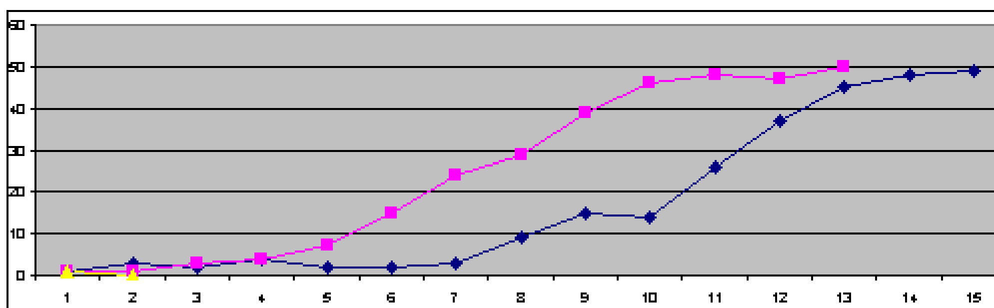


Pri zvýšení parametra s sa potvrdilo, že n má iba malý vplyv na konvergenciu, pričom ale zvýšenie s podstatne urýchlilo konvergenciu.

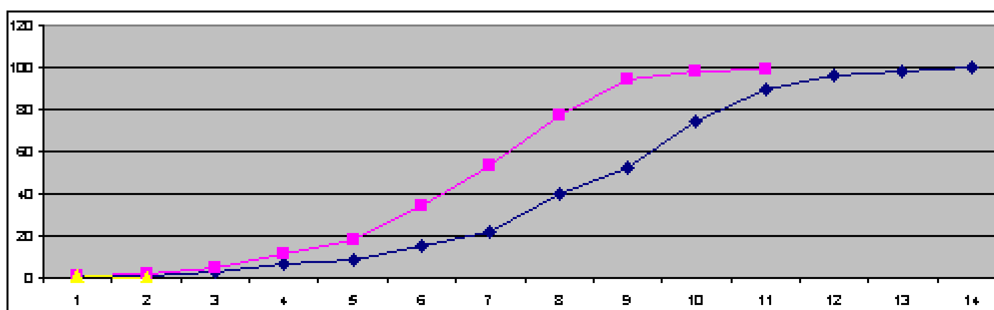
**n = 25 , s = 1.5 , $f_{max}$ = 1.5**



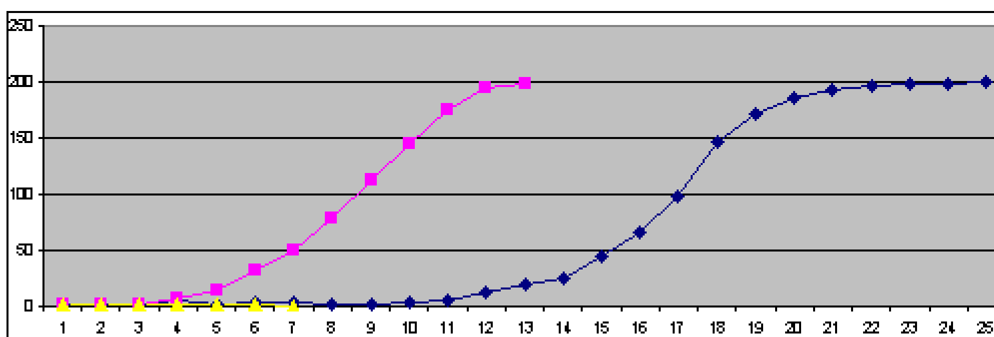
**n = 50 , s = 1.5 , $f_{max}$ = 1.5**




**n = 100 , s = 1.5 , $f_{max}$ = 1.5**




**n = 200 , s = 1.5 , $f_{max}$ = 1.5**




Pri poslednej zmene s na 1,5 sa striedal náhodný výber s tournament selection pre s = 2. V takomto prípade je veľké riziko toho, že algoritmus skončí neúspechom. Rozdiel pri zmene n je väčší ako pri klasickom tournament selection, ale nie podstatný.

# 6 Záver

Na záver by som zdôraznil niektoré výsledky, ktoré vyplývajú z experimentu.

### Proportionate selection

1. Pri proportionate selection je veľmi dôležitá veľkosť $f_{max}$ , ktorá podstatne ovplyvňuje rýchlosť konvergencie.
2. Pocet prvkov v generácii n ovplyvňuje konvergenciu len čiastočne. Zmena rýchlosti konvergencie je viditeľná až pri rádoej zmene n.
3. Ak je $f_{max}$ príliš nízke, hrozí skutočnosť, že algoritmus skončí s neúspechom, tj. že vznikne generácia iba s prvkami, ktoré majú fitness 1.

### Tournament selection

4. Naopak pri tournament selection zmena $f_{max}$ nemá žiadny vplyv na rýchlosť konvergencie, čo priamo vyplýva z definície tournament selection.
5. Ale rovnako ako pri proportionate selection má n iba malý vplyv na konvergenciu.
6. Podstatný vplyv na konvergenciu má s a to už aj pri malej zmene s, hlavne pri vysokom n.
7. Veľmi nízke s naopak znamená nebezpečenstvo, že algoritmus skončí s neúspechom.

# 7 Kód algoritmu

```
import java.math.*;
import java.io.*;

public class Evol_alg {

  public double Fmax = 9;
  public int n = 25;
  public int s = 0;
  public double[] p = new double[n + 1];
  public double[] pnew = new double[n + 1];
  public int[] c = new int[10];
  public int krok;
  PrintWriter logger;

  public void log(String c, boolean x) throws Exception {
    logger = new PrintWriter(new FileOutputStream("vystup.txt",true));
    if (x)
      logger.print(c);
    else
      logger.println(c);
    logger.flush();
    logger.close();
  }

// evol_alg je zakladna cast algoritmu
  public Evol_alg() {
  try {
    String Title = "n=" + String.valueOf(n) + "  Fmax=" + String.valueOf(Fmax) + "  s=" + String.valueOf(s);
    System.out.println(Title);
    log(Title, false);
    init();
    krok = 0;
    while ((number_fmax() < (n - 1)) && (number_fmax() != 0)) {
      writegen();
      krok++;
      nextGen();
    }
    writegen();
    log("krokov=" + String.valueOf(krok), false);
    System.out.println("krokov=" + String.valueOf(krok));
    }
    catch (Exception ex) {}
  }

// writegen vypise stav
  public void writegen() {
    try {
      String output = (String.valueOf(number_fmax()) + "  ");
      log(output, true);
      System.out.print(output);
    }
    catch (Exception ex)  {}
  }

// init nacita do pola p prvu generáciu
  public void init()  {
```

```
    p[1] = Fmax;
    for (int i = 2; i <= n; i++)
      p[i] = 1;
  }

// number_max vrati pocet prvkov generacie (pola p) s fitness fmax
  public int number_fmax() {
    int number = 0;
		for (int i = 1; i <= n; i++)
			if (p[i] == Fmax)
				number++;
		return number;
}

// nextGen nacita do pola p dalsiu generaciu
  public void nextGen() {
    for (int i = 1; i <= n; i++) {
			if (s == 0) {
				pnew[i] = p[find_one_roulete()];
			}
			else {
				pnew[i] = p[find_next(s)];
			}
		}
		if (s == 1.5) {
			int nn = Math.round(n / 2);
      for (int j = 1; j <= nn; j++)
				pnew[j] = p[find_next(2)];
			for (int l = (nn + 1); l <= n; l++)
				pnew[l] = p[find_one_roulete()];
		}
		for (int m = 1; m <= n; m++)
			p[m] = pnew[m];
  }

// suma vrati sucet fitness vsetkych prvkov pola p
  public double suma() {
		double sum = 0;
		for (int i = 1; i <= n; i++)
			sum = sum + p[i];
		return sum;
  }

// find_one_roulete vyberie poradove cislo jedneho clena do dalsej genneracie podla metody roulete
  public int find_one_roulete() {
		int k = 0;
		double x = Math.random();
		x = x * suma();
		double s = 0;
		while (s < x) {
		      k++;
		      s = s + p[k];
		}
		if (k > n)
			k = n;
    if (k < 1)
      k = 1;
		return k;
  }

// find_any nahodne vyberie poradove cislo jedneho clena do dalsej generacie
  public int find_any() {
		double x = Math.random();
```

```
                x = (x * n)  + 1;
                int k = (int)Math.round(x);
                if (k > n)
                        k = n;
    if (k < 1)
      k = 1;
                return k;
  }

// find_next vyberie nahodne s prvkov pola p a urobi na nich tornament selection a vrati poradie vitaza
  public int find_next(double ss) {
                int k;
                for (int i = 1; i <= ss; i++)
                        c[i] = find_any();
                k = c[1];
                for (int j = 2; j <= ss; j++)
                        if (p[k] < p[c[j]])
                                k = c[j];
                return k;
  }


  public static void main(String[] args) {
    try {
      Evol_alg ea = new Evol_alg();
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```