Laborator 1

Introducere în sisteme evolutive – algoritmi genetici

1. Regulament de disciplină

- Prezența la orele de laborator este obligatorie.
- Studenții au obligația de a respecta orarul laboratorului și grupele/semigrupele din care fac parte
- Recuperarea orelor de laborator se face în conformitate cu regulamentul universității.
- Fiecare student va fi evaluat în urma activităților de laborator, nota minimă cu care studentul se consideră promovat fiind 5. Studenții care au la sfârșitul laboratorului situația neîncheiată sau încheiată cu nota sub 5 nu vor putea susține examenul de disciplină.

2. Obiectivele laboratorului

- Prezentarea regulamentului de disciplină
- Prezentarea uneltelor necesare pentru orele de laborator (JGAP / AForge.Net)
- Însușirea modului de lucru pentru soluționarea unor probleme simple folosind algoritmi genetici
- Evaluarea execuției algoritmului genetic: convergență, timpul de execuție, eficiența găsirii soluției, etc.

3. Prezentare teoretică a algoritmilor genetici

Algoritmii genetici fac parte din clasa algoritmilor evolutivi, care vizează căutarea soluției unei probleme, în mod iterativ, prin optimizarea la fiecare pas a unui indice de performanță asociat soluției.

Cu scop recapitulativ, se oferă în figura 1. schema logică a algoritmilor genetici. Modul cel mai comun de abordare implică următorii pași:

1. Stabilirea structurii cromozomului: numărul de gene și genotipul fiecăreia
2. Generarea populației inițiale
3. Construirea indivizilor din cromozomi cu ajutorul funcției de *mapping*
4. Evaluarea indivizilor folosind funcția de *fitness*
5. Selecția indivizilor
6. Crearea descendenților prin încrucișare (engl. *crossover*), mutație, etc.
7. Stabilirea populației care constituie generația următoare

După faza de inițializare, pașii 3-7 se efectuează într-o buclă repetitivă pentru un anumit număr de generații sau până când soluția găsită îndeplinește anumite criterii de performanță.

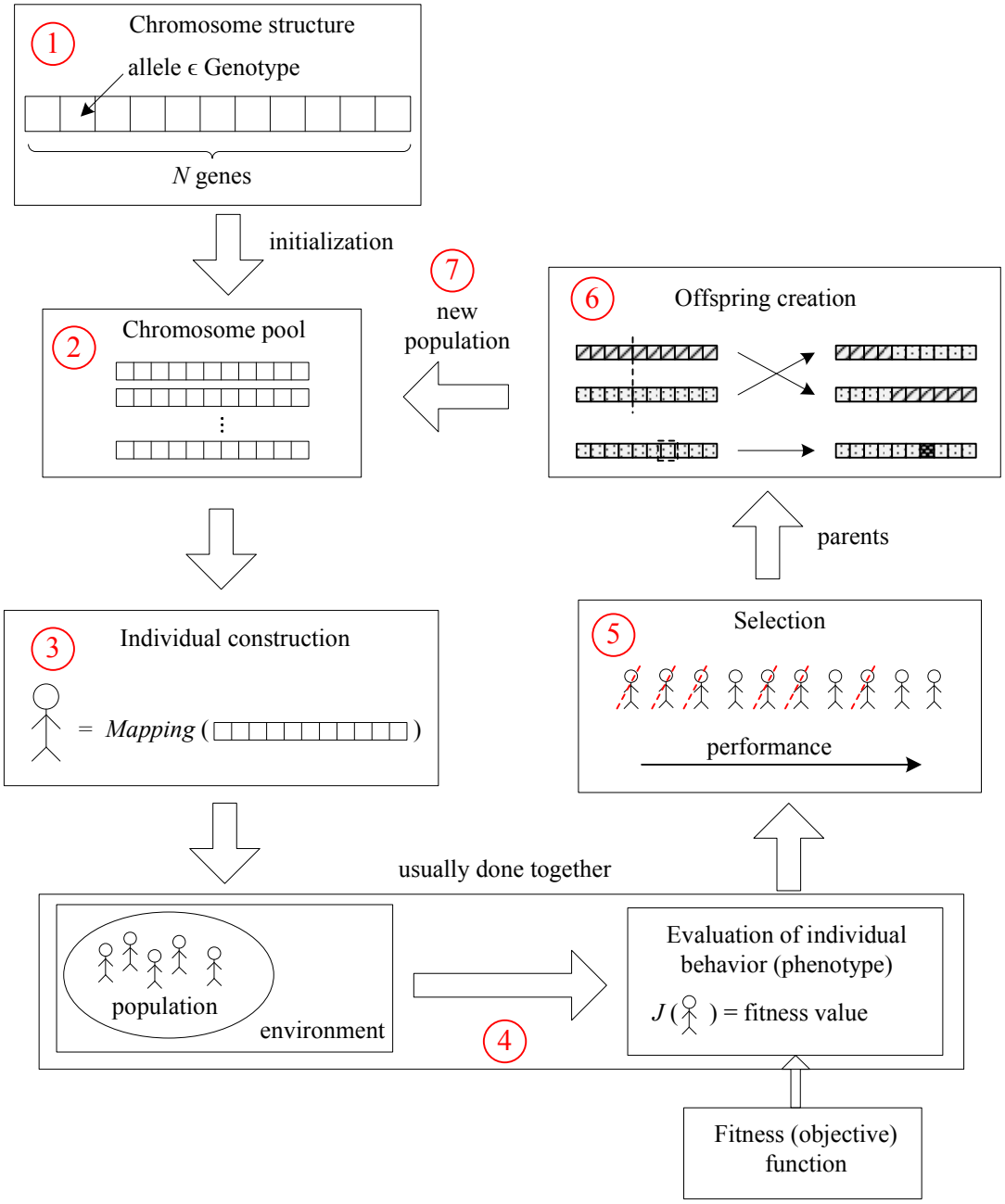

Figura 1. Schema logică a algoritmilor genetici

4. Unelte necesare pentru laborator

În majoritatea limbajelor de programare există biblioteci sau pachete care implementează o mare parte a funcționalităților la abordarea cu algoritmi genetici (generarea populației inițiale, modalitățile de selecție, operatorii de crossover și mutație), lăsând în grija programatorului doar structura cromozomului și funcțiile de mapping și de performanță. Dintre cele mai folosite amintim:

- pentru Java SE: JGAP (Java Genetic Algorithm Package), Jenetics, etc

- pentru C#: Aforge.Net, etc
- pentru Matlab: Genetic Algorithm Toolbox
- pentru Python: pygalib
- pentru Javascript: genetic-js

Pentru obiectivul acestui laborator se va folosi varianta Java SE, pachetul JGAP, pentru că pune la dispoziția programatorului mai multe exemple de programe care pot fi rulate direct, sau modificate relativ facil pentru a rezolva alte probleme. De asemenea, este open source, deci putem adapta de exemplu, operatorii de crossover și mutație la necesitățile noastre.

Este recomandat, dar nu obligatoriu, să se folosească mediul de dezvoltare Eclipse. Se va descărca versiunea *Eclipse IDE for Java Developers* de pe pagina http://www.eclipse.org/downloads/index.php. Se dezarhivează în locația dorită și se execută programul *eclipse.exe*. Este necesar să existe preinstalat și pachetul *JDK* (*java development kit*), de exemplu varianta *Java 2 SE 1.7*. Când se descarcă pachetul *JDK* trebuie avut grijă să se selecteze platforma și sistemul de operare corespunzătoare.

Se va descărca de la JGAP repository (https://sourceforge.net/projects/jgap/) ultima versiune a bibliotecii (de la secțiunea *Files*). În cadrul acestui laborator se va lucra pe exemplele distribuite în varianta *jgap_3.6.3_full*. Fiind practic o bibliotecă, pachetul JGAP nu necesită instalare, ci se dezarhivează fișierul descărcat și se referențiază ca și bibliotecă în Eclipse (pentru aceasta, click dreapta pe numele proiectului, se selectează opțiunea *Properties*, apoi *Java Build Path* din meniul din stânga, tabul *Libraries*, click pe *Add External JARs…*, apoi se selectează locația unde a fost dezarhivat pachetul JGAP și se selectează fișierul *jgap.jar*).

În figura 4 se prezintă diagrama UML a unor elemente esențiale din pachetul JGAP. Se observă că majoritatea funcționalităților sunt implementate deja în clasele existente. Contribuția programatorului este indicată cu contur roșu:

- definirea funcției de performanță (prin extinderea clasei `FitnessFunction` cu o clasă proprie care să implementeze metoda `evaluate()`)
- în programul principal, crearea obiectului de configurare, a populației inițiale, și apoi apelarea metodei `evolve()` din clasa `Genotype`.

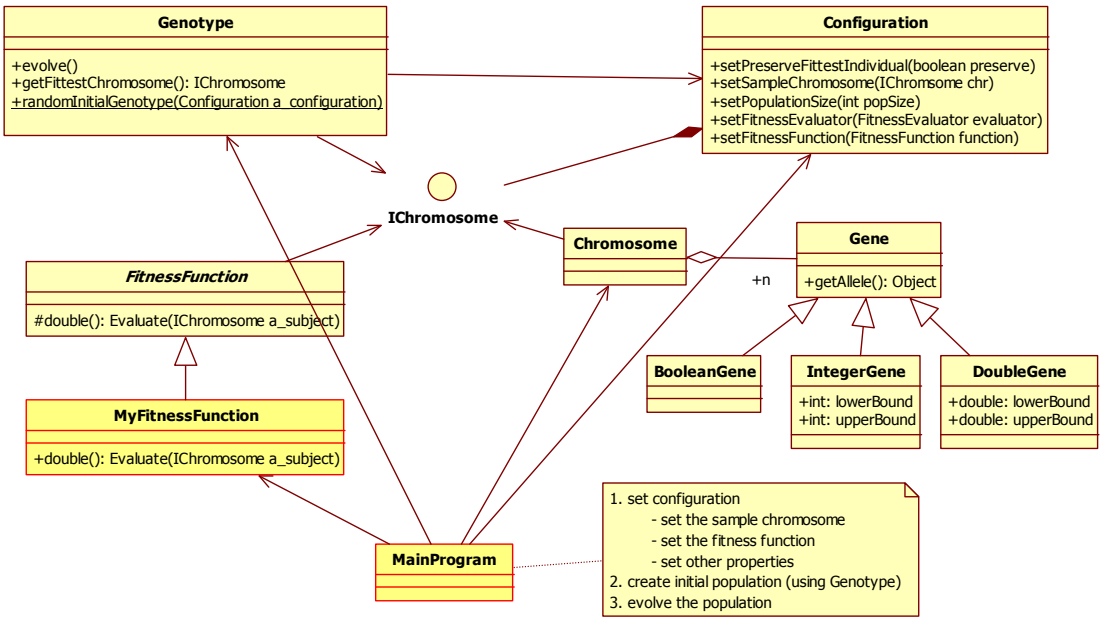

Figura 2. Structura de bază a pachetului JGAP

5. Problemă rezolvată - prezentarea modului de lucru

5.1. Formularea problemei

O problemă simplă care poate fi rezolvată cu algoritmi genetici este aceea de a calcula restul în monede, astfel încât suma să fie cât mai aproape de valoarea dorită, folosind cât mai puține monede. Aici, o soluție performantă va reprezenta un compromis între cele două cerințe. Există 4 tipuri de monede: Quarters (valoarea 0.25), Dimes (0.1), Nickels (0.05), Pennies (0.01). Sunt disponibile în total 20 de monede de tip Quarter, 30 Dimes, 50 Nickels, respectiv 80 Pennies.

5.2. Rezolvarea problemei

Necunoscuta problemei reprezintă informația despre numărul de monede din fiecare tip. Soluția problemei va fi deci un vector de 4 numere întregi. Această soluție trebuie codificată în cromozom, astfel încât să se acopere tot spațiul de căutare.

Prin urmare, vom considera un cromozom format din 4 gene cu alele numere întregi, iar valoarea alelei fiecărei gene va reprezenta câte monede din acel tip vor fi considerate. Genotipul fiecărei gene este o mulțime de numere întregi, diferită pentru fiecare genă. De exemplu, $G_1 = \{0, 1, 2, ..., 20\}$, pentru că sunt maxim 20 de monede de tip Quarter disponibile.

Algoritmul genetic va genera soluții care trebuie evaluate în funcție de obiectivele problemei, deci conform cu o funcție de performanță. Pentru evaluarea unei soluții posibile, vom evalua cele două obiective: suma sa fie cât mai aproape de suma dorită (S) și numărul total de monede să fie cât mai mic. Expresia funcției de performanță este deci:

$$J = \alpha \cdot \left| S - \sum_{i=1}^{4} g_i \cdot V_i \right|^2 + \sum_{i=1}^{4} g_i \,,$$

unde g_i reprezintă valoarea genei i (numărul de monedede tipul i), V_i este valoarea modei de tipul i, iar α este un factor de scalare. Cele două contribuții la funcția de performanță sunt însumate, performanța fiind cu atât mai bună cu cât J are valoare mai mică.

Pentru a stabili cât anume contează fiecare obiectiv la funcția globală de performanță se pot folosi diverse artificii:

- un factor liniar de scalare α pentru a stabili prioritatea fiecărei contribuții
- penalizarea progresivă a erorii folosind ridicarea la pătrat. Astfel, o eroare mică va fi penalizată puțin, iar o eroare mare va fi penalizată puternic.
- penalizarea diferențiată în funcție de diferite condiții logice, etc.

5.3. Implementare

În continuare vom parcurge codul sursă al programului *ConstraintExample* (*JGAP/examples/src/examples/constraint*). Există două fișiere:

- *ConstraintExample.java* – conține programul principal unde se setează toți parametrii necesari pentru a rula programul. Aici se află și bucla iterativă descrisă în capitolul 3.
- *SampleFitnessFunction*.java – conține implementarea funcției de performanță (funcția obiectiv)

Redăm mai jos secvența de cod simplificată a programului principal, unde am eliminat toate detaliile care nu sunt absolut necesare pentru rularea programului:

Secvența de cod 1: Exemplu simplificat

```
import org.jgap.*;
import org.jgap.impl.*;
public class ConstraintExample {
  private static final int NUM_EVOLUTIONS = 100;
  public static void main(String[] args) throws
InvalidConfigurationException{
    double targetAmount = 1.84;
    Configuration conf = new DefaultConfiguration();
     Configuration.resetProperty(Configuration.PROPERTY_FITEVAL_INST);
     //we use Delta evaluator (low value for fitness = better):
     conf.setFitnessEvaluator(new DeltaFitnessEvaluator());
     conf.setPreservFittestIndividual(true);
     conf.setKeepPopulationSizeConstant(true);

     FitnessFunction fitnessFunction = new
SampleFitnessFunction(targetAmount);
     conf.setFitnessFunction(fitnessFunction);

     Gene[] sampleGenes = new Gene[4];
     sampleGenes[0] = new IntegerGene(conf, 0, 20); // Quarters
     sampleGenes[1] = new IntegerGene(conf, 0, 30); // Dimes
     sampleGenes[2] = new IntegerGene(conf, 0, 50); // Nickels
     sampleGenes[3] = new IntegerGene(conf, 0, 80); // Pennies
     IChromosome sampleChromosome = new Chromosome(conf, sampleGenes);
     conf.setSampleChromosome(sampleChromosome);
```

```
     conf.setPopulationSize(80);
     Genotype population = Genotype.randomInitialGenotype(conf);

     for (int i = 0; i < NUM_EVOLUTIONS; i++) {
       population.evolve();
       IChromosome bestSolutionSoFar =
population.getFittestChromosome();
       DisplayIndividual(bestSolutionSoFar);
     }
   }

   public static void DisplayIndividual(IChromosome chr){
     System.out.print("Fitness value: " + chr.getFitnessValue());
     System.out.print(", Coins: ");
     for (int i = 0; i < 4; i++)
       System.out.print(SampleFitnessFunction.getNrCoinsAtGene(chr, i)
+ " ");
     System.out.println(", total change: " +
SampleFitnessFunction.amountOfChange(chr));
   }
}
```

Mai întâi se construiește obiectul de configurare `gaConf` în varianta `DefaultConfiguration`, care va conține toți parametrii necesari rulării algoritmilor genetici:

`DeltaFitnessEvaluator()` – se specifică faptul că o valoare numerică mică returnată de funcția de evaluare reprezintă o performanță mare a individului. Această setare a evaluatorului se folosește atunci când avem nevoie să minimizăm o anumită cantitate – în acest caz, diferența între obiectiv și valoarea curentă. Pentru a putea seta un nou `fitnessEvaluator` trebuie în prealabil resetată valoarea `PROPERTY_FITEVAL_INST`.

`setPreservFittestIndividual()` – se optează pentru păstrarea întotdeauna a celui mai bun individ (*ellite selection* cu un singur individ in elită)

`setKeepPopulationSizeConstant()` – mărimea populației este menținută constantă în fiecare generație, având valoarea setată cu instrucțiunea `gaConf.setPopulationSize(80)`

`setFitnessFunction()` – se specifică o instanță a clasei ce reprezintă funcția de fitness cu care se evaluează fiecare individ. Această clasă extinde `FitnessFunction` și suprascrie metoda `evaluate()`, care este folosită la evaluarea performanței.

`setSampleChromosome()` – structura cromozomului este setată folosind un obiect de tip `IChromosome`, pe baza căruia programul va genera cromozomii aleatori din populația inițială. În aces caz, cromozomul are 4 gene de tip `IntegerGene`, fiecare având setate valori minime și maxime între care se află valoarea alelei.

`randomInitialGenotype()` – se creează populația inițială aleatoare

În secvența de cod 1, bucla iterativă va executa un număr fix de 100 de iterații (`NUM_EVOLUTIONS`). Fiecare iterație reprezintă o întreagă generație a evoluției (mapping, evaluare, selecție, crearea descendenților), practic pașii 3-7 descriși în capitolul 3. Metoda care implementează ”trecerea” unei generații este `genotype.evolve()`. De

asemenea, se tipărește pe ecran valoarea funcției obiectiv pentru cel mai performant cromozom din fiecare generație și valorile alelelor cromozomului.

În secvența de cod 2 se prezintă funcția de evaluare, care în metoda `evaluate(…)` calculează performanța unui cromozom dat. În primul rând, se calculează valoarea reprezentată de cromozom (folosind funcția de mapping), iar apoi, conform cu cerința problemei, se penalizează diferența până la valoarea obiectiv `targetAmount`, precum și numărul de monede. Se folosește pătratul valorii `changeDifference` pentru a introduce penalizări progresive (dacă valoarea este foarte departe de obiectiv, se penalizează puternic, altfel mai puțin). De asemenea, există un factor de scalare (`300`), care specifică prioritatea mai mare a valorii în fața numărului de monede.

Secvența de cod 2: Funcția de performanță

```
package lab1;

import org.jgap.*;
public class SampleFitnessFunction extends FitnessFunction {
    private final double targetAmount;

    public SampleFitnessFunction(double targetAmount) {
     this.targetAmount = targetAmount;
    }

    public double evaluate(IChromosome chr) {
     double changeAmount = amountOfChange(chr); //mapping
     int totalCoins = getTotalNumberOfCoins(chr); //mapping

     double changeDifference = Math.abs(targetAmount - changeAmount);
     double fitness = 300 * changeDifference * changeDifference;
     fitness += totalCoins > 1 ? totalCoins : 0;
     return fitness;
    }

    public static double amountOfChange(IChromosome chr) {
      int numQuarters = getNrCoinsAtGene(chr, 0);
      int numDimes = getNrCoinsAtGene(chr, 1);
      int numNickels = getNrCoinsAtGene(chr, 2);
      int numPennies = getNrCoinsAtGene(chr, 3);
      return (numQuarters * 0.25) + (numDimes * 0.1) + (numNickels *
0.05) + (numPennies*0.01);
    }

    public static int getNrCoinsAtGene(IChromosome chr, int position) {
      Integer numCoins = (Integer)chr.getGene(position).getAllele();
      return numCoins.intValue();
    }

    public static int getTotalNumberOfCoins(IChromosome chr) {
      int totalCoins = 0;
      for (int i = 0; i < chr.size(); i++)
        totalCoins += getNrCoinsAtGene(chr, i);

      return totalCoins;
    }
  }
```

5.4. Testarea programului și măsurarea performanțelor

Convergența algoritmului

Atunci când se rulează programul de mai sus se poate observa că performanța afișată crește la fiecare pas (deci valoarea funcției de evaluare scade). Acest fapt este asigurat de păstrarea celui mai bun individ din fiecare generație prin apelul metodei `setPreservFittestIndividual()`. Faptul că la fiecare pas soluția poate fi doar îmbunătățită conferă, în acest caz, convergența algoritmului (în unele cazuri, este posibil ca soluția găsită să fie doar un optim local, nu global, acest caz fiind analizat în lucrările următoare). Este posibil ca, în timpul rulării programului, abaterea de la suma totală să crească de la o generație la alta. Aceasta se întâmplă numai în cazul în care performanța totală se îmbunătățește datorită scăderii numărului de monede.

Caracterul aleator al algoritmilor genetici este demonstrat de faptul că fiecare rulare produce, de cele mai multe ori, rezultate diferite. Caracterul aleator este dat de: generarea populației inițiale, operatorii de încrucișare și mutație și selecția indivizilor.

Eficiența algoritmului

Pentru a măsura timpul de execuție al algoritmului, se pot introduce niște instrucțiuni suplimentare. Timpul de execuție depinde, evident, de mașina pe care rulează programul, deci este mai relevant să analizăm eficiența algoritmului.

Eficiența algoritmului este dată de numărul de evaluări ale funcției de performanță raportat la mărimea spațiului de căutare. Spațiul de căutare este $S=G_1 \times G_2 \times G_3 \times G_4$, unde G_i este genotipul pentru gena i. Pentru problema dată, spațiul de căutare conține $|S|=20 \cdot 30 \cdot 50 \cdot 80 = 2\ 400\ 000$ soluții posibile. Evaluarea performanței se face pentru fiecare cromozom din fiecare generație, deci numărul de evaluări se poate calcula înmulțind numărul de generații cu mărimea populației (în acest caz 100 generații $\cdot 80$ indivizi = 8000 de evaluări). Putem deci concluziona că s-au evaluat 8000 / 2400000 = 0.33% dintre soluțiile posibile.

De fapt, la rularea programului se observă că după o anumită generație soluția nu se mai îmbunătățește deloc. Întrucât fiecare rulare produce rezultate diferite, putem considera o medie a numărului de generații în care este găsită soluția finală și calculăm eficiența algoritmului cu noua valoare.

6. Problemă propusă

În continuare se va aborda o problemă clasică de optimizare, numită în literatură problema rucsacului (engl. *knapsack problem*). Se dă un rucsac cu volum dat V și n obiecte care trebuie introduse în rucsac volumul o_1, o_2, ..., o_n. Fiecare obiect are un anumit volum și o valoare asociată. Fără a depăși rucsacului, se cere să se maximizeze valoarea totală a obiectelor în rucsac.

Pentru un caz particular, se consideră n=10, V=30, iar volumul și valoarea fiecărui obiect sunt date de: $vol_i = i$, $val_i = i^2$, unde i=1, 2, ..., n.

Modul de abordare prezentat anterior este potrivit și pentru această problemă. Soluția problemei conține informația despre fiecare obiect dacă este considerat sau nu, prin urmare cromozomul va fi format n gene de tip `BooleanGene`. Pentru a calcula performanța J a unui cromozom, vom calcula valoarea totală a obiectelor și o penalizare

diferențiată J_{pen}: dacă volumul rucsacului este depășit se va penaliza performanța cu o valoare foarte mare și progresivă în funcție de cât este depășirea:

$$J = \sum_{i=1}^{4} g_i \cdot val_i + J_{pen}, \text{ unde } J_{pen} = if\left(V > V_{tot}\right) then\ 0\ else\ \alpha \cdot \left(V - V_{tot}\right)^2, \text{ iar } V_{tot}$$

este volumul total al obiectelor: $V_{tot} = \sum_{i=1}^{4} g_i \cdot vol_i$.

7. Exerciții

1. Explicați modul de funcționare al următorilor operatori: încrucișare (crossover), mutație. Căutați și înțelegeți diferite variante pentru operatorul de selecție. Explicați modul de funcționare al selecției de tip turneu și ruletă. Căutați și înțelegeți și alte variante de selecție.
2. Rulați programul prenzentat în capitolul 5. Observați de la ce generație nu se mai îmbunătățește soluția. Estimați câte evaluări s-au făcut până în acel moment. Rulați programul de mai multe ori și observați diferențele.
3. În cadrul programului prenzentat în capitolul 5, ajustați factorii care influențează modul de funcționare al algoritmilor genetici (mărimea populației, numărul de generații, parametrii funcției de performanță – scalarea diverselor contribuții de penalizare, etc.), și observați efectul acestor modificări asupra vitezei de convergență a algoritmului.
4. Implementați programul pentru problema prezentată în capitolul 6. Pentru implementare este necesar să modificați structura cromozomului și funcția de evaluare. Rulați programul, modificați parametrii de funcționare și testați diferențele.
5. Modificați implementarea pentru problema rucsacului: există *n* tipuri de obiecte, cu maxim 5 obiecte disponibile din fiecare tip. Problema este de a afla câte obiecte din fiecare tip trebuie puse în rucsac, cu aceleași obiective de a nu depăși volumul maxim și de a maximiza valoarea totală.