**Algoritmi genetici cu populații de lungime variabilă și eroare codificată în soluție**

Autori: Pelle Remus-Nicolae – [remusnicolaepelle@gmail.com](mailto:remusnicolaepelle@gmail.com)

Moise Ana-Ioana – [moiseana100@yahoo.com](mailto:moiseana100@yahoo.com)

Facultatea de Științe, Universitatea din Oradea

ABSTRACT:

În această lucrare, ne propunem diversificarea algoritmilor genetici clasici prin lungimea variabilă a genelor, a soluțiilor și a populațiilor din fiecare generație.

1. **Introducere**

Acest proiect este relaizat folosind programarea orientată pe obiecte, având clasele Gene, Solution și Engine. Un obiect de tipul Solution este format dintr-un array de obiecte Gene. În clasa Engine, există o listă de obiecte Solution care sunt prelucrate si determinăm care dintre acestea sunt mai apropiate de răspunsul căutat.

Proiectul are ca scop rezolvarea unor probleme cu soluție formată dintr-un număr oarecare de numere reale.

1. **Programare orientată pe obiecte**
   1. **Class Gene**

În primul rând, avem nevoie ca numerele să fie manipulate cu ușurință, în consecință am început proiectul cu conceptul de genă. Clasa Gene reprezintă un număr într-un vector de numere int astfel:

* primul element din vector ne va da semnul. Dacă este mai mic decât 5, semnul este negativ, iar in caz contrar, semnul este pozitiv;
* “digits” este un număr întreg reprezentând numărul de cifre ale numărului, iar următoarele “digits” elemente din vector reprezintă cifrele acestuia;
* “decimals” este similar lui “digits”, dar pentru zecimalele numărului.

Metoda ConverToDouble folosește pașii de mai sus pentru a returna valoarea numărului în format double.

Pentru inițializare, am creat un constructor care primește ca și parametri valorile ce vor fi atribuite câmpurilor digits și decimals, inițializăm vectorul de cifre cu lungimea digits+decimals+1, și apoi atribuim cifre aleatoare în vector.

|  |
| --- |
| public class Gene  {  public int digits, decimals, n;  public int[] number;  public Gene(int digits, int decimals)  {  this.digits = digits;  this.decimals = decimals;  n = digits + decimals + 1;  number = new int[n];  initRandom();  }  public void initRandom()  {  for (int i = 0; i < n; i++)  number[i] = Engine.random.Next(10);  }  public double ConvertToDouble()  {  double nr = 0;  for (int i = 1; i <= digits; i++)  nr = nr \* 10 + number[i];  for (int i = digits + 1; i < n; i++)  {  double s2 = number[i];  s2 /= Math.Pow(10, i - digits);  nr += s2;  }  if (number[0] < 5)  nr = nr \* -1;  return nr;  }  } |

* 1. **Class Solution**

După cum spuneam, problema va avea ca și soluție un număr oarecare de numere, deci am creat conceptul de soluție. În clasa Solution, avem un vector de gene numit genes, care este inițializat și populat în constructor: pentru fiecare element din vector, am apelat constructorul clasei Gene cu valori aleatoare pentru digits si decimals. Intervalul trebuie ales în funcție de preferințe sau de valorile necesare, de exemplu dacă problema necesită valori cu precizie foarte mare, intervalul de la decimals poate lua valori între 10 și 15.

Pentru a determina cât de apropiată de răspuns este soluția, trebuiesc aduse anumite calcule asupra genelor din soluție. Toate acestea sunt puse într-o funcție numită funcția de adecvare. Echivalentul acesteia în cod este Fadeq. Pentru acest exemplu, am decis să facem ceva simplu, și anume suma genelor aflate în soluție, care va fi comparată mai târziu cu un număr introdus de la tastatură.

Am realizat o funcție View pentru a vedea numerele componente soluției și funcția de adecvare.

|  |
| --- |
| public class Solution  {  public Gene[] genes;  public Solution()  {  genes = new Gene[Engine.solutionLength];  initRandom();  }  public void initRandom()  {  for (int i = 0; i < genes.Length; i++)  {  int digits = Engine.random.Next(2, 5);  int decimals = Engine.random.Next(3, 13);  genes[i] = new Gene(digits, decimals);  }  }  public double FAdeq()  {  double r = 0;  for (int i = 0; i < genes.Length; i++)  {  r += genes[i].ConvertToDouble();  }  return r;  }  public void View()  {  for (int i = 0; i < Engine.solutionLength; i++)  Console.Write(genes[i].ConvertToDouble() + ", ");  Console.WriteLine();  Console.WriteLine("FAdeq:\t" + FAdeq());  }  } |

* 1. **Class Engine**

Pentru a folosi teoria algoritmilor genetici, avem nevoie de: o populație inițială de soluții, de soluțiile ordonate în funcție de valoarea erorii și de o populație de părinți din care se va crea următoarea generație folosind operația de încrucișare. Algoritmii genetici simulează sistemele evolutive bazate pe selecție naturală.

Engine este o clasă statică în care vom inițializa toate câmpurile globale și în care vom realiza toate acțiunile la nivel de populație de soluții. Pentru inițializare, atribuim valori aleatoare în intervale alese de utilizator campurilor populationLength, parentsLength și solutionLength. După aceea, adăugăm în lista population soluții care se inițializează aleator.

|  |
| --- |
| public static class Engine  {  public static Random random = new Random();  public static int populationLength, parentsLength, solutionLength;  public static float requestedNumber;  public static List<Solution> population = new List<Solution>();  public static List<Solution> parents = new List<Solution>();  public static void initStats()  {  populationLength = random.Next(80, 100);  parentsLength = random.Next(2, 5);  solutionLength = random.Next(8, 10);  }  public static void initPopulation()  {  for (int i = 0; i < populationLength; i++)  population.Add(new Solution());  }  ...  } |

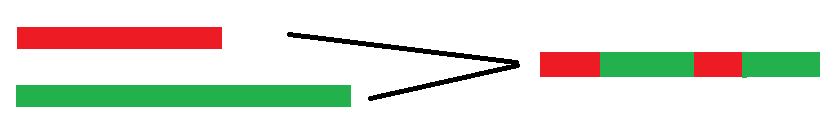
Având populația inițială, acum trebuie să ordonăm soluțiile în funcție de Fadeq. Funcția de adecvare returnează suma numerelor din soluție, iar noi dorim ca aceasta să fie cât mai apropiată de o valoare introdusă de la tastatură, salvată în câmpul requestedNumber. În concluzie, pentru ordonare, folosim metoda Sort, iar criteriul de comparare este modul din diferența dintre FAdeq și requestedNumber. Pentru a simula cât mai corect selecția naturală, am adăugat în clasa Solution un câmp long share, în care salvăm ponderea cu care determinăm dacă această soluție va fi aleasă ca și părinte pentru următoarea generație sau nu. Pentru a calcula share, ordonarea a fost făcută descrescător, iar share va fi egal cu indexul la care se află soluția în populație la puterea a 3-a.

|  |
| --- |
| public static void OrderPopulation()  {  population.Sort(delegate (Solution a, Solution b)  {  return Math.Abs(b.FAdeq() - requestedNumber)  .CompareTo(Math.Abs(a.FAdeq() - requestedNumber));  });  for (int i = 0; i < populationLength; i++)  population[i].share = (int)Math.Pow(i + 1, 3);  } |

Când trecem la urmatoarea generație, trebuie să avem aleși părinții acesteia. Pentru selecție, folosim metoda MonteCarloAlgorithm, care va returna indexul din population la soluția respectivă. Avem nevoie de suma ponderilor soluțiilor, iar apoi alegem un număr aleator “t” mai mic decât cel de la valoarea sumei. Folosind acest număr, parcurgem elementele populației și scădem ponderile acestora din valoarea lui “t”, până când acesta este negativ. Returnăm indexul ultimului element parcurs. Acum avem primul părinte, iar pentru a nu se repeta în listă, îl vom șterge din populație, astfel nu va mai intra la următoarea selecție. Când avem toți părinții, aceștia sunt ordonați crescător după același criteriu ca și populatia.

|  |
| --- |
| public static void SelectParents()  {  parents.Clear();  for (int i = 0; i < parentsLength; i++)  {  parents.Add(population[MonteCarloAlgorythm()]);  population.Remove(parents[i]);  }  parents.Sort(delegate (Solution a, Solution b)  {  return Math.Abs(a.FAdeq() - requestedNumber)  .CompareTo(Math.Abs(b.FAdeq() - requestedNumber));  });  }  public static int MonteCarloAlgorythm()  {  float sum = 0;  for (int i = 0; i < population.Count; i++)  sum += population[i].share;  float t = (float)random.NextDouble() \* sum;  int index = 0;  while (t > 0)  {  t -= population[index].share;  index++;  }  return index - 1;  } |

Înainte să trecem la următoarea generație, vom dori să vedem populația și părinții. Pentru asta am creat două metode de afișare. În metoda NextGeneration, ștergem elementele din population și atribuim valori aleatoare câmpurilor inițializate la început. Parcurgem cu un for până la populationLength, și trebuie să încrucișăm doi părinți pentru a avea o nouă soluție creată. Întâi, ne asigurăm că cei doi părinți sunt diferiți, după care apelăm metoda de încrucișare CrossNCuts. Încrucișarea se referă la combinarea celor două soluții pentru a obține una nouă. Cea mai simplă încrucișare este să alegem un număr aleator x, iar soluția finală va avea primele x elemente din primul părinte, iar restul din al doilea. Aceasta ar fi încrucișare cu două tăieturi, una din primul părinte, iar a doua din al doilea părinte. Când încrucișăm cu mai multe tăieturi, vom avea intercalat elemente din primul părinte, apoi din al doilea, iarăși din primul, ș.a.m.d., până când completăm N tăieturi.



Deoarece soluțiile nu sunt alcătuite dintr-un singur număr, avem nevoie de toate numerele din gene într-un singur vector. Pentru asta, am creat metoda GetAllValues în clasa Solution care returnează un vector de numere întregi. Pentru a determina ce lungime va avea fiecare tăietură de la cei doi părinți, avem length1 și length2, egali fiecare cu lungimea parintelui său împarțit la numărul de tăieturi. Folosind o variabilă counter care numără la a câta tăietură suntem, putem determina din care părinte trebuie să introducem valori folosind counter%2. La final, incrementam counter și se repetă procesul până când counter are valoarea lui N. Lista de valori întregi rezultată trebuie mai departe să poată crea o nouă soluție. Pentru aceasta, în clasa Solution am creat un nou constructor care primește List<int> ca parametru. Vom crea vectorul genes astfel: determinăm lungimea aleator, in funcție de câte gene mai trebuie să creăm. În funcție de această lungime, dăm valori aleatoare câmpului digits, iar diferența va fi valoarea lui decimals. După ce avem gena creată (aleator), parcurgem elementele necesare din listă și i le atribuim genei, iar apoi ștergem acele cifre pentru a trece la următorul pas. Această încrucișare se va folosi pentru crearea întregii populații noi.

|  |
| --- |
| public static class Engine  {  ...  public static void NextGeneration()  {  population.Clear();  initStats();  for (int i = 0; i < populationLength; i++)  {  int index1 = 0, index2 = 0;  Solution aux;  do  {  index1 = random.Next(parents.Count);  index2 = random.Next(parents.Count);  } while (index1 == index2);  aux = CrossNCuts(parents[index1], parents[index2]);  aux.Mutate();  population.Add(aux);  }  }  public static Solution CrossNCuts(Solution s1, Solution s2)  {  int N = 4; //number of cuts  int n1 = 0, n2 = 0;  for (int i = 0; i < s1.genes.Length; i++)  {  n1 += s1.genes[i].n;  n2 += s2.genes[i].n;  }  int length1 = n1 / N, length2 = n2 / N;  int[] solution1 = s1.getAllValues(n1), solution2 = s2.getAllValues(n2);  List<int> result = new List<int>();  int counter = 0;  do  {  if (counter % 2 == 0)  for (int i = counter\*length1; i < (counter + 1) \* length1; i++)  result.Add(solution1[i]);  else  for (int j = counter\*length2; j < (counter + 1) \* length2; j++)  result.Add(solution2[j]);  counter++;  } while (counter < N);  return new Solution(result);  }  public static void ViewPopulation()  {  foreach (Solution solution in population)  solution.View();  Console.WriteLine();  }  public static void ViewParents()  {  foreach (Solution solution in parents)  solution.View();  Console.WriteLine();  }  ...  } |

|  |
| --- |
| public class Solution  {  ...  public Solution(List<int> result)  {  genes = new Gene[Engine.solutionLength];  int count = Engine.solutionLength;  while (count > 0)  {  int n = Engine.random.Next(3, result.Count / count);  if (count == 1)  n = result.Count;  int digits = Engine.random.Next(n / 3);  genes[Engine.solutionLength - count] = new Gene(digits, n-digits-1);  for (int i = 0; i < n; i++)  genes[Engine.solutionLength - count].number[i] = result[i];  result.RemoveRange(0, n);  count--;  }  }  public int[] getAllValues(int length)  {  int[] result = new int[length];  int k = 0;  for (int i = 0; i < genes.Length; i++)  {  for (int j = 0; j < genes[i].n; j++)  {  result[k] = genes[i].number[j];  k++;  }  }  return result;  }  ...  } |

Introducem și operația de mutare, spunem că o genă trece prin procesul de mutare când un element aleator din vectorul de cifre primește o valoare aleatoare, iar pentru soluție este similar atunci când o genă aleatoare se mutează. Pentru aceasta, am introdus metoda Mutate atât în clasa Gene, cât și în clasa Solution.

|  |
| --- |
| public class Solution  {  ...  public void Mutate()  {  int i = Engine.random.Next(genes.Length);  genes[i].Mutate();  }  ...  }  public class Gene  {  ...  public void Mutate()  {  int index = Engine.random.Next(n);  int value;  do  {  value = Engine.random.Next(10);  } while (number[index] == value);  number[index] = value;  }  ...  } |

* 1. **Class Program**

În clasa Program, citim de la tastatură câmpul din Engine requestedNumber și apelăm metodele din Engine necesare programului. Deoarece dorim să observăm evoluția în timp a populației, vom itera la infinit cu apelarea metodelor de funcționare a programului și de afișare a valorilor. Putem să realizăm 100 de pași înainte să afișăm din nou ceva, pentru viteză mai mare.

|  |
| --- |
| static void Main(string[] args)  {  Console.Write("Please insert the number that you want the function of adequation to use: "); Engine.requestedNumber = float.Parse(Console.ReadLine());  Engine.initStats();  Engine.initPopulation();  do  {  Console.Clear();  Engine.ViewPopulation();  Engine.ViewParents();  for (int i = 0; i < 100; i++)  {  Engine.OrderPopulation();  Engine.SelectParents();  Engine.NextGeneration();  }  Console.ReadKey();  } while (true);  } |

1. **Concepte folosite**

Aplicația este realizată în limbajul de programare C#, folosind conceptele programării orientate pe obiect.

1. **Rezultate și concluzii**

Programul are ca scop găsirea soluțiilor pentru probleme cu rezolvare complicată, care necesita un timp de rulate mare pentru a rezolvate cerintele care necesita număr ridicat de numere. Prin acest program, nu folosim nici unul din acei algoritmi, însă începem cu valori aleatoare, iar din cele mai apropiate soluții, vom crea soluții noi. În acest fel, programul va evolua iar soluțiile vor fi din ce în ce mai apropiate de răspunsul corect.

Acesta a fost realizat cu lungime variabilă a genelor, soluțiilor și populațiilor pentru a nu se ajunge la un blocaj al evoluției. Acest lucru înseamnă că soluțiile ajung să se repete, iar la un moment dat toată populația de părinți va fi realizată din aceleași numere. Prin diversificarea unor valori, evităm acest lucru. Pe de altă parte, dacă valorile sunt prea diversificate, pierdem viteza de evoluție până în punctul în care soluțiile devin mai mult sau mai puțin aleatoare.

În concluzie, pentru rezultate cât mai bune, trebuie experimentat cu intervalul de valori, numărul de incrucișări ș.a. De exemplu, este posibil ca o populație cât mai mare și o populație de părinți cât mai mică să ducă la o evoluție mai rapidă din punct de vedere al numărului de generații parcurse.

1. **Bibliografie**

1. Laslo E, Ionescu V.S. Algoritmică C++, MatrixRom Bucureşti 518 pagini 2010, ISBN 978-973-755-640-0

2. Knuth D.E. “Arta programării calculatoarelor, vol. 1, Algoritmi fundamentali” Editura Teora, 1999 Bucureşti

3. Dan DUMITRESCU, “Algoritmi genetici şi startegii evolutive – aplicaţii ȋn inteligenţă artificială şi domenii conexe” Editura Albastră, ISBN V106, Cluj-Napoca

4. David WELLER, Alexandre Santos LOBÃO, Ellen HATTON, .NET Game Programming in C#, Apress ISBN(pbk) 1-59059-319-7, New-York, USA