**TEORIA GRAFURILOR ŞI PROGRAMARE ORIENTATᾸ PE AGENŢI CU COMPORTAMENT ÎN FUNCŢIE DE NECESITᾸŢI**

**Remus-Nicolae PELLE1, Eugen LASLO2**

**1**Student anul III, Informatică, Facultatea de Informatică şi Științe, Universitatea din Oradea, remusnicolaepelle@gmail.com

**2**Lect. univ. dr. Departamentul de Matematică și Informatică, Facultatea de Informatică şi Științe, Universitatea din Oradea, [lasloeugen@yahoo.com](mailto:lasloeugen@yahoo.com)

|  |
| --- |
| ***Abstract****:*  *Pentru această lucrare, ne-am propus să construim un model de agenţi al căror comportament să fie influenţat de diferite necesităţi ce trebuiesc satisfăcute. Aceste necesităţi pot fi schimbate oricând, sau se pot adăuga mereu mai multe. Pentru ca aplicaţia să fie vizuală, am reprezentat locurile şi conexiunile dintre acestea ca grafuri.* |

***Cuvinte cheie****: agenţi, necesităţi, grafuri*

**INTRODUCERE**

Acest proiect este relaizat folosind programarea orientată pe obiecte, având clasele Place, Person şi Engine. Obiectele de tip Place pot avea tipurile “Casă”, “Loc de muncă” şi “Restaurant”, reprezentate simplu printr-un caracter: H, W, respectiv R. Un obiect de tipul Person are, în proiectul ales, necesităţile bani, mâncare şi relaxare, care sunt satisfăcute de locurile W, R respectiv H. În clasa Engine, Există o listă cu toate persoanele şi locurile, şi matricea de valori pentru graful locurilor.

**PROGRAMARE ORIENTATĂ PE OBIECTE**

* 1. **Clasa Place**

În primul rând, avem nevoie de o hartă pe care să se plimbe agenţii pentru satisfacerea necesităţilor. Această hartă este un graf ai cărui noduri sunt reprezentate ca fiind obiecte ale clasei Place. Aceasta poate avea 3 tipuri, Home, Work sau Restaurant, are locaţie pe ecran şi are câştig şi pierdere, care funcţionează astfel: tipul care oferă câştig la necesitatea money, oferă o pierdere la necesitatea relaxation; cel care oferă câştig la relaxation, oferă pierdere la food; iar cel care oferă câştig la food, oferă pierdere la money. Pierderea este mereu mai mică decât câştigul.

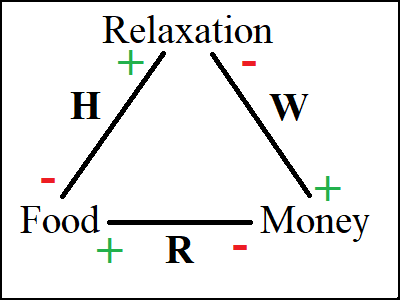


Fig. 1: Câştigul şi pierderea pentru fiecare tip de obiect Place pentru necesităţile date.

Constructorul acestei clase primeşte ca parametru un string de unde se extrag toate informaţiile necesare. Acesta arată astfel: W 350 50 0.01 0.004 , şi vor fi citite mai multe astfel de stringuri dintr-un fişier pentru mai multe obiecte. Se face split după caracterul spaţiu, type va fi primul string din vectorul rezultat, următoarele două valori reprezintă punctul de pe hartă unde se află obiectul curent, iar ultimele două valori reprezintă câştigul şi costul acestui obiect.

|  |
| --- |
| public class Place  {  public string type;  public Point location;  public float gain, cost;  public Place(string data)  {  string[] dataSplit = data.Split(' ');  type = dataSplit[0];  location = new Point(int.Parse(dataSplit[1]), int.Parse(dataSplit[2]));  gain = float.Parse(dataSplit[3]);  cost = float.Parse(dataSplit[4]);  }  } |

* 1. **Clasa Person**

Fiecare persoană are ca şi necesităţi variabilele reale money, food şi relaxation, care vor avea valori între 0 şi 1. Vor avea viteză de mişcare pe ecran în pixeli, un obiect Place care îi va fi casă şi altul pentru a şti care a fost ultimul loc de pe hartă vizitat, o listă de obiecte Place pentru a şti ce traseu trebuie să parcurgă pentru a ajunge la locul dorit, şi un PictureBox pentru a putea fi reprezentat grafic pe ecran.

Aceste valori sunt iniţializate în constructor: cele trei necesităţi primesc valori aleatoare, viteza este un număr întreg între 5 şi 8, casa o primeşte ca şi parametru, iar locaţia curentă este iniţial acasă, iar pentru PictureBox se alege părintele pentru a fi afişat unde trebuie, se ia dimensiunea din clasa Engine, şi se atribuie locaţia în funcţie de locul curent.

|  |
| --- |
| public class Person  {  public float money, food, relaxation;  public int speed;  public Place home;  public Place currentLocation;  public List<Place> destination = new List<Place>();  public PictureBox pB;  public Person(Place home)  {  money = (float)(Engine.rnd.NextDouble() / 3);  food = (float)(Engine.rnd.NextDouble() / 2 + 0.3);  relaxation = (float)(Engine.rnd.NextDouble() / 2 + 0.3);  speed = Engine.rnd.Next(5, 8);  this.home = home;  currentLocation = home;  pB = new PictureBox();  pB.Parent = Engine.form.pictureBox1;  pB.Size = new Size(Engine.size, Engine.size);  pB.Location = new Point(home.location.X - 7, home.location.Y - 7);  }  [...]  } |

În metoda Do, se decide întreg comportamentul persoanei, deci aceasta stă la baza componentei de agent a acestei clase. Această metodă este apelată la fiecare tick al timer-ului aplicaţiei. Fiecare dintre necesităţi scade puţin, dar nu poate să scadă sub 0. Se calculează fericirea persoanei prin media aritmetică a necesităţilor, iar în funcţie de aceasta, persoana îşi schimba culoarea cu un gradient între roşu şi verde. Dacă persoana se află la destinaţie, atunci se verifică tipul acesteia şi se modifică necesităţile corespunzătoare. Dacă necesitatea a ajuns la valoarea 1, se alege o nouă destinaţie.

Dacă persoana nu se află la destinaţie, se calculează traiectoria pe care trebuie să se mişte persoana pentru a ajunge la următorul punct din graf folosind arctangentă, se calculează x şi y folosind viteza şi cosinus, respectiv sinus din unghiul rezultat. Se păstrează partea fracţionară în două variabile, left şi top, deoarece poziţiile pictureBox-ului nu pot fi decât întregi, şi nu dorim să schimbăm traiectoria din cauza erorii. La final, dacă ajunge suficient de aproape de următorul punct din graf, este aşezat pe acesta şi se scoate din lista destinaţie.

|  |
| --- |
| public void Do()  {  money -= 0.001f; food -= 0.001f; relaxation -= 0.001f;  if (money < 0)  money = 0;  if (food < 0)  food = 0;  if (relaxation < 0)  relaxation = 0;  happiness = (money + food + relaxation) / 3;  pB.BackColor = Engine.colors[(int)(happiness \* 100)];  if (destination.Count == 0)  {  if (currentLocation.type == "H")  {  relaxation += currentLocation.gain;  food -= currentLocation.cost;  if (relaxation >= 1)  {  relaxation = 1;  GetNewDestination();  }  if (food < 0)  food = 0;  }  [...]  }  else  {  double angle = Math.Atan2(destination[0].location.Y - currentLocation.location.Y, destination[0].location.X - currentLocation.location.X);  double x = left + speed \* Math.Cos(angle);  double y = top + speed \* Math.Sin(angle);  pB.Left += (int)x;  pB.Top += (int)y;  left = x - (int)x;  top = y - (int)y;  if (Engine.Distance(pB.Location,destination[0].location)<Engine.size)  {  currentLocation = destination[0];  pB.Location = new Point(destination[0].location.X - 7,  destination[0].location.Y- 7);  destination.RemoveAt(0);  }  }  } |

Metoda GetNewDestination găseşte cel mai apropiat şi eficient loc care îi va satisface agentului necesitatea cea mai mică. Pentru asta, se ia o listă de indecşi pentru toate locurile din tipul necesar. Se calculează vectorul de distanţe faţă de locaţia curentă folosind algoritmul lui Dijkstra, după care se parcurge lista de indecşi. Pentru fiecare dintre locuri, se calculează eficienţa locului respectiv şi se caută minimul şi indexul minim în mod clasic.

Acum că ştim locul la care trebuie să ajungem, mai trebuie doar să determinăm cel mai scurt mod de a ajunge din locaţia curentă la acel loc. Întâi determinăm toate drumurile dintre acestea folosind parcurgerea în adâncime, după care se alege drumul cu distanţa minimă.

|  |
| --- |
| public void GetNewDestination()  {  List<int> indexes = new List<int>();  if (money < food && money < relaxation)  for (int i = 0; i < Engine.n; i++)  if (Engine.places[i].type == "W")  indexes.Add(i);  if (food < money && food < relaxation)  for (int i = 0; i < Engine.n; i++)  if (Engine.places[i].type == "R")  indexes.Add(i);  if (relaxation < food && relaxation < money)  indexes.Add(Engine.places.FindIndex(x => x.Equals(home)));  int minIndex = 0;  float min = Int32.MaxValue;  if (indexes.Count > 1)  {  float[] distances = Engine.Dijkstra(Engine.places.FindIndex(x => x.Equals(currentLocation)));  for (int i = 0; i < indexes.Count; i++)  {  float efficiency = Engine.places[indexes[i]].gain / Engine.places[indexes[i]].cost;  if (distances[indexes[i]] \* efficiency < min)  {  min = distances[indexes[i]] \* efficiency;  minIndex = i;  }  }  }  Engine.DFS(Engine.places.FindIndex(x => x.Equals(currentLocation)), indexes[minIndex], new List<Place>(), new bool[Engine.n]);  minIndex = 0;  min = Int32.MaxValue;    for (int i = 0; i < Engine.paths.Count; i++)  {  float distance = 0;  for (int j = 1; j < Engine.paths[i].Count; j++)  distance += Engine.Distance(Engine.paths[i][j - 1].location, Engine.paths[i][j].location);  if (distance <= min)  {  min = distance;  minIndex = i;  }  }  destination = Engine.paths[minIndex];  Engine.paths.Clear();  } |

* 1. **Clasa Engine**

Clasa Engine se ocupă cu acţiunile elaborate ale aplicaţiei. De asemenea, are o multitudine de variabile încapsulate: numărul de noduri din graf, dimensiunea persoanelor, variabila aleatoare, lista de persoane şi locuri din aplicaţie şi matricea de adiacenţă a grafului. Pentru a = ne ajuta la partea vizuală, avem un obiect Bitmap şi un obiect Graphics pentru a desena harta, avem vectorul cu culori ale persoanelor şi avem instanţa curentă a formularului.

|  |
| --- |
| public static class Engine  {  public static int n;  public static int size = 15;  public static Random rnd = new Random();  public static List<Person> people = new List<Person>();  public static List<Place> places = new List<Place>();  public static float[,] matrix;  public static Graphics grp;  public static Bitmap bmp;  public static Color[] colors;  public static Form1 form;  [...]  } |

Metoda Init este apelată la începutul aplicaţiei şi se ocupă cu iniţializarea unor valori: se citesc locurile şi conexiunile grafului din fişier, se adaugă nişte persoane în aplicaţie, se iniţializează vectorul de culori ca şi un gradient de la roşu la verde, cu 100 de culori, se iniţializează bmp şi grp, după care se desenează conexiunile şi punctele grafului, apoi se atribuie bmp ca şi imaginea picturebox-ului principal. Când se desenează nodurile, întâi se umple un cerc alb pentru a şterge liniile conexiunilor acelui loc, apoi se scrie tipul de loc care este acolo, pentru a observa ce necesităţi se vor satisface.

|  |
| --- |
| public static void Init(Form1 f)  {  form = f;  ReadFromFile(@"..\..\TextFile1.txt");  Populate();  InitializeColours();    bmp = new Bitmap(f.pictureBox1.Width, f.pictureBox1.Height);  grp = Graphics.FromImage(bmp);  DrawStreets();  DrawPlaces();  f.pictureBox1.Image = bmp;  }  [...] |

La fiecare tick al timer-ului aplicaţiei, se apelează metoda Tick din Engine, care apelează comportamentul tuturor persoanelor şi actualizează necesităţile afişate. Pentru asta, pentru fiecare persoană se scriu toate necesităţile sale.

|  |
| --- |
| public static void Tick()  {  foreach(Person person in people)  person.Do();  UpdateStats();  }  private static void UpdateStats()  {  form.listBox1.Items.Clear();  for(int i=0; i<people.Count;i++)  {  form.listBox1.Items.Add("Person" + (i + 1));  form.listBox1.Items.Add("money: "+people[i].money.ToString("0.00"));  form.listBox1.Items.Add("food: " + people[i].food.ToString("0.00"));  form.listBox1.Items.Add("relaxation: " +  people[i].relaxation.ToString("0.000"));  form.listBox1.Items.Add("");  }  }  [...] |

Algoritmul lui Dijkstra nu este modificat faţă de cel clasic. Însă, pentru parcurgerea în adâncime, de fiecare dată a trebuit să fie salvat noul drum de la locul curent la destinaţia dorită. Vom avea o nouă variabilă, o listă de liste de locuri, în care se vor adăuga toate drumurile găsite. De asemenea, deoarece funcţia este recursivă, avem nevoie de mai mulţi parametri: indexul de start şi de sfârşit, lista cu drumul creat până acum şi lista de noduri deja vizitate. Se adaugă nodul de start în listă, se setează ca vizitat, iar dacă nodul acesta este nodul final, atunci drumul se adaugă în lista de drumuri şi se pune nodul ca fiind nevizitat. În caz contrar, pentru fiecare alt nod, dacă există drum spre acela şi nu a fost încă vizitat, se face o copie a listei curente, se reapelează metoda recursiv şi se setează nodul curent ca fiind nevizitat.

|  |
| --- |
| public static List<List<Place>> paths = new List<List<Place>>();  public static void DFS(int start, int end, List<Place> list, bool[] visited)  {  visited[start] = true;  list.Add(places[start]);  if (start == end)  {  paths.Add(list);  visited[end] = false;  }  else  for (int i = 0; i < n; i++)  if (matrix[start, i] > 0 && !visited[i])  {  List<Place> t = new List<Place>(list);  DFS(i, end, t, visited);  visited[i] = false;  }  } |

1. **Rezultate și concluzii**

Programul are ca scop crearea unui model de agent care acţionează în funcţie de necesităţi definite de programator, oricare ar fi acestea. Prin acest program, am ales un model al necesităţilor care satisface o necesitate la costul alteia, satisfacţiile şi costurile avănd o formă circulară precum se poate observa şi în Fig.1, pentru a nu avea o necesitate favorată sau vreo necesitate nefavorată.

Deoarece aplicaţia este vizuală, rezultatele se pot observa imediat: se începe cu valori mici pentru fiecare necesitate, deci culoarea persoanelor tinde spre roşu, iar după un timp de rulare, persoanele au culoare verzuie. Dacă aceste persoane nu ar reuşi să ajungă la o culoare ce tinde spre verde, atunci ar înseamna că programul nu funcţionează corect. De asemenea, mai pot apărea probleme la alegerea nodurilor destinaţiei sau a ordinei acestora, ceea ce se poate observa dacă persoanele iau un drum lung până să ajungă la destinaţie sau dacă nu respectă legăturile grafului.

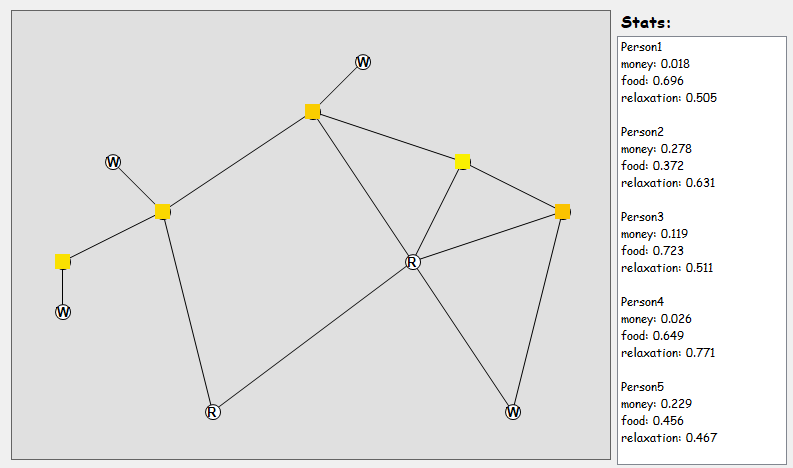


Fig. 2: Agenţii la începutul apliaţiei.

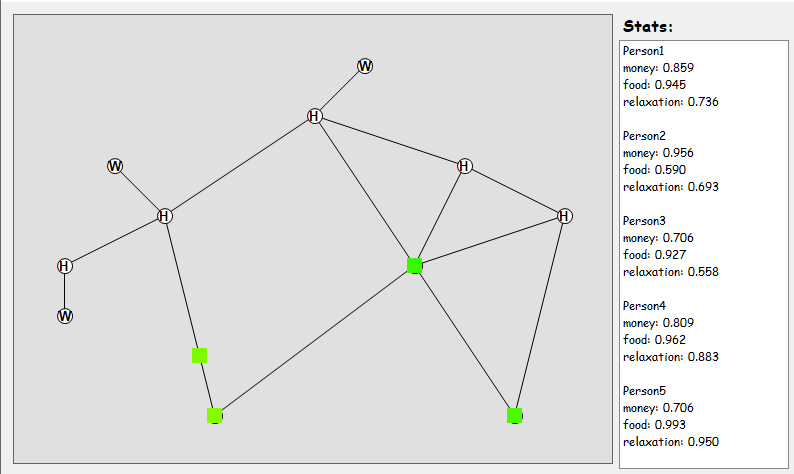


Fig. 3: Agenţii cu necesităţile satisfăcute.

În concluzie, acest program poate fi folosit şi cu alte necesităţi în funcţie de dorinţele utilizatorului. Acesta ar trebui să funcţioneze la fel de bine atâta timp cât satisfacţiile şi costurile nu favorează sau defavorează vreo necesitate.

**BIBLIOGRAFIE**

[1]Laslo, E, Ionescu, V.S. (2010): *Algoritmică C++*, MatrixRom Bucureşti.

[2]Knuth, D.E. (1999): *Arta programării calculatoarelor, vol. 1, Algoritmi fundamentali*, Editura Teora, Bucureşti.

[3]Dumitrescu, D. (2000): *Algoritmi genetici şi startegii evolutive – aplicaţii ȋn inteligenţă artificială şi domenii conexe*, Editura Albastră, Cluj-Napoca.

[4]Weller, D., Santos Lobão, A., Hatton, E. *.NET Game Programming in C#*, Apress ISBN(pbk) 159059-319-7, New-York, USA.