

Noțiuni fundamentale de programare funcțională

1. Obiective

Obiectivele acestui laborator sunt legate de prezentarea unor caracteristici fundamentale ale paradigmii de programare funcțională: imutabilitate, recursivitate, diferite moduri de declarare a unor obiecte-funcții, trimiterea unor funcții ca parametri pentru alte funcții, închideri (closures).

2. Programarea funcțională

Programarea funcțională este o paradigmă de programare, cum este și programarea orientată pe obiecte. Aceste idei sunt destul de vechi: primul limbaj funcțional, Lisp, a apărut în 1958 și este încă utilizat. Deși limbajele funcționale sunt succinte și expresive, până recent nu au fost folosite pe scară largă, ci în general pentru aplicații specifice de inteligență artificială.

Însă în zilele noastre există noi provocări. Trebuie prelucrate mulțimi mari de date în paralel și într-o manieră scalabilă. Programele trebuie testate cât mai ușor și ar fi de dorit ca logica acestora să fie exprimată clar, într-un mod ușor de înțeles, care să se axeze în special pe rezultatele obținute (*ce se dorește*) și nu pe detaliile de execuție (*cum* se obțin rezultatele).

De aceea, multe limbaje de programare importante au început să includă caracteristici funcționale, de exemplu tipuri generice, metode anonime și expresii lambda. *LINQ* din platforma .NET este bazat pe concepte funcționale. De asemenea, limbajele funcționale propriu-zise au început să se bucură de o mai mare atenție și sunt folosite din ce în ce mai mult pentru aplicații comerciale.

În general, programarea funcțională se studiază folosind un limbaj conceput special pe principiile funcționale, de exemplu Scheme (un dialect Lisp), Haskell (un limbaj funcțional „pur”), Scala (compatibil cu Java VM) sau F# (compatibil cu platforma .NET). Dezavantajul acestei abordări este că trebuie învățate atât conceptele programării funcționale, destul de diferite de cele ale programării imperative, cât și sintaxa unui limbaj nou, de obicei la fel de diferită de sintaxa limbajelor apropiate de C.

De aceea, în cadrul laboratoarelor de *Inteligentă artificială*, vom folosi limbajul C# și ne vom concentra doar pe aspectele funcționale. După cum se va vedea, chiar folosind o sintaxă cunoscută, programele vor arăta destul de diferit față de cele tipice din programarea orientată pe obiecte.

Un alt avantaj este folosirea numeroaselor funcții din bibliotecile disponibile pentru platforma .NET și interoperabilitatea deplină cu alte programe .NET.

Datorită sintaxei concise, programele funcționale sunt în general mai compacte decât programele imperative echivalente. Să considerăm un program care calculează suma pătratelor numerelor de la 1 la n .

Imperativ	Funcțional
<pre>private int Square(int x) { return x * x; } public int SumOfSquares(int n) { int sum = 0; for (int i = 1; i <= n; i++) sum += Square(i); return sum; }</pre>	<pre>public static int SumOfSquares(int n) { int Square(int x) => x * x; int SumSq(int m) => Range(1, m).Select(x => Square(x)).Sum(); return SumSq(n); }</pre>

Nu este necesară înțelegerea codului funcțional acum. Putem spune doar că ideea de bază este transformarea succesivă a unei liste de numere de la 1 la n într-o listă cu pătratele lor și apoi sumarea acestor elemente. Se poate observa înlănțuirea apelurilor de funcții. De asemenea, funcția de ridicare la patrat este definită foarte simplu, aproape de echivalentul său matematic.

3. Imutabilitate

Una din caracteristicile programării funcționale este *imutabilitatea* (immutability). Odată inițializat, un obiect nu mai poate fi modificat. Acest lucru poate părea ciudat pentru un programator obisnuit cu ideea de variabilă din programarea imperativă, al cărei sens este tocmai cel de a se modifica. Totuși, imutabilitatea este prezentă în platforma .NET. De exemplu, obiectele string sunt imutabile (immutable). Toate operațiile asupra unui string returnează alte string-uri, nu modifică obiectul inițial.

Valorile imutabile sunt asemănătoare celor din matematică. Datorită lor, programul poate deveni mai clar. Un avantaj important apare în cazul aplicațiilor concurente, unde sincronizarea se simplifică mult prin renunțarea la conceptul de stare partajată (shared state). De asemenea, imutabilitatea simplifică testarea aplicațiilor, deoarece starea internă a obiectelor nu se schimbă în timpul execuției programului și deci un test se va comporta la fel indiferent de operațiile anterioare efectuate.

În momentul de față, limbajul C# nu are suport implicit pentru valori imutabile. Totuși, programatorul poate folosi proprietăți read-only de acces la starea internă a obiectului și poate crea doar metode care nu schimbă starea obiectului, ci instanțiază un obiect nou cu starea modificată, ca în exemplul următor.

```
public class Ellipse
{
    // proprietățile Rx, Ry și Area nu se pot modifica din exterior
    public readonly double Rx, Ry;
    public double Area { get { return Rx * Ry * Math.PI; } }

    public Ellipse(double rx, double ry)
    {
        Rx = rx;
```

```

        Ry = ry;
    }

    public Ellipse Stretch(double factor)
    {
        // se returnează o nouă elipsă, nu se modifică Rx din obiectul curent
        return new Ellipse(Rx * factor, Ry);
    }

    public Ellipse ModifyWith(double newRx = 0, double newRy = 0)
    {
        if (newRx == 0) newRx = Rx;
        if (newRy == 0) newRy = Ry;
        return new Ellipse(newRx, newRy);
    }

    public override string ToString()
    {
        return $"Ellipse: rx = {Rx:F2}, ry = {Ry:F2}, a = {Area:F2}";
    }
}

public class MainEllipse
{
    public static void Test()
    {
        var e = new Ellipse(1, 2);           // Ellipse: rx = 1.00, ry = 2.00, a = 6.28
        e = e.Stretch(3);                  // Ellipse: rx = 3.00, ry = 2.00, a = 18.85
        e = e.ModifyWith(newRx: 5);       // Ellipse: rx = 5.00, ry = 2.00, a = 31.42
        e = e.ModifyWith(newRy: 6);       // Ellipse: rx = 5.00, ry = 6.00, a = 94.25
    }
}

```

4. Recursivitate

În programarea funcțională pură nu există bucle, deoarece acestea presupun actualizarea unui contor. Valorile fiind imutabile, acest lucru nu este posibil. De aceea, echivalentul funcțional al buclelor este reprezentat de funcțiile recursive.

Orice program iterativ care implică bucle poate fi convertit într-o variantă recursivă și viceversa. De obicei, transformarea se face ca mai jos:

Varianta iterativă	Varianta recursivă
program-iterativ (parametri) cod-înainte-de-bucă while (<i>condiție</i>) cod-bucă return <i>rezultat</i>	program-recursiv (parametri) cod-înainte-de-bucă return <i>funcție-recursivă</i> (parametri, variabile-în-cod-înainte-de-bucă) funcție-recursivă (parametri, variabile) if (<i>not condiție</i>) return <i>rezultat</i> cod-bucă return <i>funcție-recursivă</i> (parametri, variabile-modificate)

Transformarea este posibilă și în cazul în care bucla conține instrucțiuni de tip break, continue sau return. De exemplu, funcția recursivă poate returna, pe lângă rezultatul propriu-zis calculat, și o etichetă de tip sir de caractere, enumerare sau întreg, care să corespundă acestor instrucțiuni. Prin testarea etichetei, se pot trata și aceste situații speciale.

Mai jos este prezentat algoritmul lui Euclid pentru determinarea celui mai mare divizor comun pentru două numere întregi, în cele două variante.

Varianta iterativă (pseudocod)	Varianta recursivă (C#)
<pre>procedure gcd(a, b) while b ≠ 0 t ← b b ← a modulo b a ← t return a</pre>	<pre>public static int Gcd(int a, int b) { if (b == 0) return a; else return Gcd(b, a % b); }</pre>

Următorul exemplu arată o funcție de calcul al factorialului:

```
public static BigInteger Factorial(int n) // structura BigInteger se găsește în System.Numerics
{
    if (n <= 1)
        return 1;
    else
        return n * Factorial(n - 1);
}
```

sau, mai compact:

```
public static BigInteger Factorial(int n)
{
    return (n <= 1) ? 1 : (n * Factorial(n - 1));
}
```

Atunci când se folosesc funcții recursive, parametrii sunt memorati pe stivă, iar dacă nivelul de recursivitate este foarte mare, acesta poate produce o excepție de memorie insuficientă. Pentru a evita acest lucru, în unele limbi de programare se folosește metoda tail recursion. Mai jos este rescrisă funcția de calcul al factorialului în acest mod. Rezultatul este calculat în acumulatorul acc și transmis ca parametru în funcția recursivă.

```
public static BigInteger Factorial(int n, BigInteger acc)
{
    if (n < 2)
        return acc;
    return Factorial(n - 1, n * acc);
}
```

Tail recursion este o metodă de optimizare foarte utilă în aplicațiile din „viața reală”, care permite compilatorului să transforme, intern, funcția recursivă într-o buclă. Acest procedeu este recunoscut de compilator, de exemplu, în limbajul F#, dar nu și în C#, unde se poate folosi în mod

natural o buclă. Totuși, considerăm că această metodă nu ține de filosofia recursivității, ci este o modalitate de optimizare practică și de aceea nu vom insista asupra ei în restul laboratorului.

De multe ori, funcțiile de ordin superior, pe care le vom studia în laboratorul 2, pot fi folosite în loc de recursivitate sau bucle, iar această modalitate este mai simplă și mai clară decât recursivitatea, dar și mai elegantă decât folosirea bucelor.

5. Funcții ca obiecte

5.1. Delegatul *Func*

Limbajul C# permite lucrul cu funcții la fel cum se lucrează și cu alte tipuri de obiecte. Acesta se realizează cu ajutorul conceptului de delegat (delegate), care poate fi văzut ca un pointer la o funcție și care poate referenția o metodă cu aceeași semnătură. Delegații se folosesc în mod curent pentru a trata evenimente, de exemplu, evenimentele controalelor *Windows Forms*, sau alte metode de tip callback.

Pentru a simplifica lucrul cu funcții, există o serie de delegați generic predefiniți, dintre care noi vom studia delegatul *Func*. Fie codul de mai jos:

```
private static double Div(int x, int y)
{
    return (double)x / (double)y;
}

public static void Test1()
{
    Func<int, int, double> Operation = Div;
    double result = Operation(10, 10);
    Console.WriteLine(result);
}
```

Argumentele din *Func* reprezintă, în ordine, tipurile parametrilor funcției, iar ultimul este tipul de return. În cazul nostru, sunt două argumente de tip *int*, iar tipul de return este *double*.

Următorul exemplu definește o funcție fără niciun parametru, care întoarce un *int*:

```
private static Random _rand = new Random();

private static int Rand()
{
    return _rand.Next(100);
}

public static void Test2()
{
    Func<int> FuncRand = Rand;
    double result = FuncRand();
    Console.WriteLine(result);
}
```

Obiectele de tip Func pot fi folosite ca orice alt obiect, adică pot fi atribuite cu diferite valori, trimise ca parametru și apelate când este nevoie:

```
private static int Add(int x, int y)
{
    return x + y;
}

private static int Mul(int x, int y)
{
    return x * y;
}

private static int PerformOperation(Func<int, int, int> operation, int x, int y)
{
    return operation(x, y);
}

public static void Test3()
{
    int result = PerformOperation(Add, 5, 5);
    Console.WriteLine(result);
    result = PerformOperation(Mul, 5, 5);
    Console.WriteLine(result);
}
```

5.2. Expresii lambda

Expresiile lambda sunt funcții anonte, pentru care nu se definește un nume. În C#, funcțiile pot primi ca argumente alte funcții. Atunci când aceste funcții-argument sunt foarte simple sau sunt utilizate într-un singur loc, este utilă exprimarea lor mai succintă. De exemplu, expresiile lambda sunt foarte utile când se dorește executarea unor operații asupra listelor sau a altor tipuri de colecții, după cum vom vedea în laboratorul 2.

Să considerăm funcția de ordin superior PerformOperation(`Func<int, int, int> operation, int x, int y`) din exemplul de mai sus, care primește ca argument o funcție și o aplică asupra a două alte argumente.

Dacă funcțiile Add și Mul sunt definite doar pentru a fi folosite de către PerformOperation, atunci este mai simplu să scriem:

```
public static void Test4()
{
    int result = PerformOperation((x, y) => x + y, 5, 5);
    Console.WriteLine(result);
    result = PerformOperation((x, y) => x * y, 5, 5);
    Console.WriteLine(result);
}
```

O expresie lambda folosește operatorul `=>` pentru a separa lista de parametri de codul executabil. Următoarele exemple arată diferite moduri de definire a unor expresii lambda:

```

Func<int, int> Square = x => x * x;
Func<int, int, int> Add = (x, y) => x + y;
Func<string, int, bool> IsTooLong = (string s, int maxLen) => s.Length > maxLen;
Func<int, int, int> Max = (x, y) => { if (x > y) return x; else return y; }; // sau:
Func<int, int, int> Max = (x, y) => (x > y) ? x : y;

```

Definițiile de mai sus sunt echivalente cu următoarele:

```

int Square(int x) => x * x;
int Add(int x, int y) => x + y;
bool IsTooLong(string s, int maxLen) => s.Length > maxLen;
int Max(int x, int y) => (x > y) ? x : y;

```

5.3. Funcții locale

Începând cu versiunea 7, C# permite definirea unor funcții în cadrul altor funcții. Acestea sunt locale și nu pot fi apelate din exterior. De exemplu:

```

public static void Test5()
{
    // Add și Mul sunt funcții locale definite în interiorul funcției Test5

    int Add(int a, int b) // definire clasică
    {
        return a + b;
    }

    int Mul(int x, int y) => x * y; // definire ca expresie lambda

    Console.WriteLine(Add(3, 4));
    Console.WriteLine(Mul(3, 4));
}

```

6. Alte aspecte

6.1. Efecte secundare

În programarea funcțională pură, funcțiile trebuie să primească niște parametri, să-i prelucreze și să returneze rezultatul. *Efectele secundare* (side effects) apar atunci când o funcție realizează și altceva decât returnarea unei valori. Însă programele reale au deseori astfel de scopuri: afișarea unor date, salvarea acestora pe hard disk, comunicații în rețea etc.

Prin urmare, se recomandă izolarea funcțiilor cu efecte secundare de funcțiile care realizează prelucrări. În cadrul acestor laboratoare, în general, vom izola funcțiile care afișează rezultatele de funcțiile care le calculează.

6.2. Metode și clase statice

În programarea funcțională în limbajul C#, în general clasele vor avea doar metode statice, deoarece toate datele se transmit ca parametri sau sunt returnate de funcții/metode. Prin urmare, clasele nu vor avea stare internă (câmpuri), ca în programarea orientată pe obiecte. Aceste metode statice pot fi, după caz, publice (accesibile din exterior) sau private (folosite de alte metode doar în cadrul unei clase).

Începând cu versiunea 6, limbajul C# a introdus directiva `using static`, care permite accesarea membrilor statici ai unui tip fără specificarea numelui tipului. De exemplu:

```
using static System.Console; // inclusă înainte de declararea namespace-ului, în zona celorlalte directive using
WriteLine("mesaj"); // instrucțiune normală din cadrul programului, într-o metodă dintr-o clasă

using static System.Convert;
int n =ToInt32(ReadLine()));

using static System.Math;
double a = Sin(30.0 / 180.0 * PI); // sin(30°)
```

Folosirea acestei directive în măsura potrivită simplifică programul și îi crește claritatea. Nu trebuie însă abuzat de această facilitate, deoarece dacă se folosesc în acest mod multe clase, se poate crea confuzie.

7. Închideri (*closures*)

Când se declară în mod normal o variabilă locală, aceasta are un domeniu în care poate fi utilizată: este accesibilă în cadrul blocului în care este declarată. Când se încearcă accesarea unei variabile locale, compilatorul o caută în blocul curent, apoi mai sus în blocurile-părinte până la blocul de pe nivelul cel mai de sus. Când fluxul de control părăsește un bloc, variabilele sale locale nu mai sunt accesibile și de obicei memoria ocupată de ele este eliberată de garbage collector.

O *închidere* (closure) este un domeniu persistent care menține variabilele locale chiar și după ce fluxul de control a ieșit din blocul de cod respectiv. Domeniul și variabilele sale locale sunt menținute prin legătura cu o funcție.

Acest mecanism este prezentat în exemplul următor:

```
public static void TestClosure()
{
    var uc = UpdateCount();
    WriteLine(uc()); // 1
    WriteLine(uc()); // 2
    WriteLine(uc()); // 3
}
```

```

private static Func<int> UpdateCount()
{
    int counter = 0;

    int IncrementCounter()
    {
        return ++counter;
    }

    return IncrementCounter; // se returnează funcția locală care incrementează variabila locală
}

```

Mai succint, metoda UpdateCount poate fi scrisă și astfel:

```

private static Func<int> UpdateCount()
{
    int counter = 0;
    // se returnează o expresie lambda care nu are parametri de intrare și care incrementează contorul
    return () => ++counter;
}

```

O închidere este într-un fel echivalentul unei variabile globale:

```

private static int counter = 0;

private static int UpdateCount()
{
    return ++counter;
}

```

Închiderile permit lucrul cu stări globale, fără a utiliza însă variabile globale. În timp ce expresiile lambda nu au stare, închiderile au. Acestea reprezintă o modalitate mai elegantă de a încapsula și menține stări globale în programarea funcțională.

8. Aplicații

8.1. Introduceți de la tastatură trei numere întregi reprezentând lungimile laturilor unui triunghi și verificați dacă acesta este un triunghi valid: laturile sunt pozitive și respectă inegalitatea în triunghi, adică orice latură este mai mică decât suma celorlalte două. Definiți două funcții separate prin expresii lambda: una care să testeze inegalitatea în triunghi și una care să testeze dacă un număr este strict pozitiv. Folosiți directive `using static` pentru clasele `System.Console`, `System.Convert` și `System.Math`.

Exemplu:

```

Introduceti latura 1: 2
Introduceti latura 2: 3
Introduceti latura 3: 4

```

Reprezinta un triunghi valid

Introduceti latura 1: 1

Introduceti latura 2: -1

Introduceti latura 3: 1

Nu reprezinta un triunghi valid

Introduceti latura 1: 1

Introduceti latura 2: 2

Introduceti latura 3: 3

Nu reprezinta un triunghi valid

8.2. Se dă mai jos varianta iterativă a unui algoritm pentru calcularea radicalului unui număr real pozitiv. Implementați varianta sa recursivă.

```
procedure SqrlIterative(x)
  if x < 0.0 then
    return nan // not a number
  else
    b ← x
    while abs(b * b - x) > 1e-12 do
      b ← ((x / b) + b) / 2.0
    return b
```

8.3. Realizați un joc simplu de ghicire a unui număr de către utilizator. Mai întâi, programul va alege în mod aleatoriu un număr natural de la 1 la 10, după care utilizatorul va introduce succesiv diferite valori, până la ghicirea numărului.

Numărul aleatoriu se generează cu ajutorul clasei .NET Random. Implementați funcția Guess în manieră recursivă.

```
public static void Problem3()
{
  var r = new Random();
  var n = r.Next(10) + 1;
  Guess(n);
}
```

Exemplu:

Introduceti numarul: 5

Numarul corect este mai mare

Introduceti numarul: 7

Numarul corect este mai mare

Introduceti numarul: 9

Numarul corect este mai mic

Introduceti numarul: 8

Corect!

8.4. Realizați o funcție care primește o altă funcție $f: \mathbb{R} \rightarrow \mathbb{R}$ și un număr real x și calculează derivata $f'(x)$ folosind metoda diferenței centrale:

$$f'(x) \approx \frac{f(x + \varepsilon) - f(x - \varepsilon)}{2\varepsilon}$$

unde ε este un număr real pozitiv mic, de exemplu 10^{-6} .

Exemplu de apel:

```
public static void Problem4()
{
    double y = Derivative(Sin, 0);
    WriteLine($"sin(0) = 0\tsin'(0) = {y:F0}"); // sin(0) = 0 sin'(0) = 1

    y = Derivative(x => x * x, 3);
    WriteLine($"x^2(3) = 9\t(x^2)'(3) = {y:F0}"); // x^2(3) = 9 (x^2)'(3) = 6
}
```

8.5. Afipați zece termeni ai șirului lui Fibonacci, în care termenul $t_n = t_{n-1} + t_{n-2}$, folosind o închidere (closure). Pentru simplitate, considerăm că funcția generează termenii începând cu t_2 , întrucât $t_0 = 0$ și $t_1 = 1$ pot fi folosiți ca valori inițiale în închidere.

Exemplu de apel:

```
public static void Problem5()
{
    var fibo = Fibonacci(); // funcția Fibonacci trebuie implementată
    for (int i = 0; i < 10; i++)
        Write($"{fibo() } "); // rezultat: 1 2 3 5 8 13 21 34 55 89
    WriteLine();
}
```

Operații cu liste și tuple

1. Obiective

Obiectivele acestui laborator sunt următoarele:

- descrierea tipului `IEnumerable`, care poate fi considerat echivalent al listelor din limbajele funcționale și prezentarea numeroaselor metode *LINQ* pentru diverse prelucrări;
- introducerea unei biblioteci, *FunCs*, care include anumite facilități de programare funcțională în C#;
- descrierea lucrului cu tuple.

2. Crearea listelor

Lista este structura de date fundamentală în programarea funcțională. Majoritatea operațiilor de prelucrare a datelor presupun transformări ale unor liste în altele, precum și determinarea unor valori singulare din elementele unor liste. De multe ori, și funcțiile recursive lucrează cu liste, pe care le construiesc în mod dinamic.

În C#, echivalentele acestor tipuri de funcții care acționează asupra listelor sunt implementate în *LINQ* (pronunțat [linc]) – *Language-Integrated Query*. Deși există mai multe variante ale acestei tehnologii (LINQ to Objects, LINQ to SQL, LINQ to XML), noi ne vom concentra doar asupra primei variante, LINQ to Objects. Tipul de bază cu care se lucrează aici este `IEnumerable`, care în general expune un enumerатор al unei colecții de date.

Există mai multe metode de definire a unor astfel de obiecte:

```
int[] array = new int[] { 1, 2 };
IEnumerable<int> intList1 = array.AsEnumerable();

List<int> list = new List<int> { 1, 2, 3 };
IEnumerable<int> intList2 = list.AsEnumerable();
```

Introducem acum biblioteca *FunCs* (<https://github.com/florinleon/FunCs>), cu facilități de programare funcțională în C#. Vom folosi această bibliotecă și în laboratoarele următoare. Biblioteca este reprezentată de un *dll*, care trebuie adăugat la proiect. Este recomandată apoi includerea directivei `using FunCs` în lista de `namespace`-uri utilizate în program.

Cu ajutorul *FunCs*, se pot declara mai ușor liste de tip `int`, `double` și `string`. De asemenea, sunt implementate și metode pentru afișarea acestora. Numele metodelor specifice bibliotecii *FunCs* sunt colorate cu mov. De exemplu:

```

var intList3 = "[ 1 2 3 4 ]".ToIntEnumF();

WriteLine(intList1.ToStringF()); // [ 1 2 ]
WriteLine(intList2.ToStringF()); // [ 1 2 3 ]
WriteLine(intList3.ToStringF()); // [ 1 2 3 4 ]

var doubleList = "[ 1.1 2 3 4.2 ]"..ToDoubleEnumF();
WriteLine(doubleList.ToStringF(1)); // argumentul ToStringF este numărul de zecimale ⇒ [ 1.1 2.0 3.0 4.2 ]

var strList = "[ a b c ]".ToStringEnumF();
WriteLine(strList.ToStringF()); // [ "a" "b" "c" ]

```

O listă poate fi creată și din elementele returnate cu instrucțiunea `yield`. Funcția următoare creează o listă cu numerele prime mai mici decât un număr dat:

```

private static IEnumerable<int> CreatePrimeList(int n)
{
    for (int i = 2; i < n; i++)
    {
        bool prime = true;
        for (int j = 2; j <= Sqrt(i); j++)
            if (i % j == 0)
            {
                prime = false;
                break;
            }
        if (prime)
            yield return i;
    }
}

public static void PrimesYield()
{
    var primeList = CreatePrimeList(100);
    WriteLine(primeList.ToStringF());
    // [ 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 ]
}

```

3. Funcții de lucru cu liste

LINQ are implementate o multitudine de funcții pentru lucrul cu liste, pe care le vom prezenta în cele ce urmează. Deși unele funcții realizează operații specifice programării funcționale, numele lor în *LINQ* este inspirat din interogarea bazelor de date, fiind diferit de numele utilizate în alte limbaje funcționale. De aceea, biblioteca *FunCs* introduce niște alias-uri:

<i>LINQ</i>	<i>FunCs</i>
Select	Map
Aggregate	Reduce
SelectMany	Bind
Where	Filter

Funcțiile originare *LINQ* nu sunt ascunse, ele pot fi folosite în continuare în mod direct în programe. Scopul alias-urilor este însă familiarizarea cu terminologia funcțională specifică.

3.1. Funcții de transformare

Funcțiile prezentate în această secțiune sunt esențiale, fiind folosite în majoritatea programelor funcționale, de aceea trebuie bine înțelese.

Funcția Map/Select aplică o funcție tuturor elementelor unei liste, returnând o nouă listă. De exemplu, să presupunem că avem o listă de numere întregi și dorim să le ridicăm pe toate la patrat:

```
var list1 = "[ 2 1 3 ]".ToIntEnumF();
var list2 = list1.Map(x => x * x); // [ 4 1 9 ]
```

Transformările sunt generale, nu doar numerice. De exemplu, putem aplica funcția Map pentru a determina lungimea cuvintelor dintr-o listă:

```
var list1 = "[ Laborator de IA ]".ToStringEnumF();
var list2 = list1.Map(s => s.Length); // [ 9 2 2 ]
```

sau pentru a transforma numere în siruri de caractere etc.:

```
string Nume(int n)
{
    if (n == 1) return "unu";
    else if (n == 2) return "doi";
    else if (n == 3) return "trei";
    else return "";
}

var list1 = "[ 2 1 3 ]".ToIntEnumF();
var list2 = list1.Map(n => Nume(n)); // [ "doi" "unu" "trei" ]
```

Funcția Map/Select are și o variantă în care se pot utiliza indecșii elementelor din listă. De exemplu:

```
class Country
{
    public readonly string Name;
    public readonly int Population;

    public Country(string name, int population)
    {
        Name = name;
        Population = population;
    }
}
```

```

public static void MapExample()
{
    var countries = new List<Country> { new Country("Russia", 17098242), new Country("Ukraine", 603628),
        new Country("France", 551695), new Country("Spain", 498511), new Country("Sweden", 450295),
        new Country("Norway", 385178), new Country("Germany", 357386), new Country("Finland", 338145),
        new Country("Poland", 312685), new Country("Italy", 301338) };

    var formatedList = countries.Map((country, index) => $"{index+1}. {country.Name}: {country.Population}");
    foreach (string s in formatedList)
        WriteLine(s);
}

```

cu rezultatul:

```

1. Russia: 17098242
2. Ukraine: 603628
3. France: 551695
4. Spain: 498511
5. Sweden: 450295
6. Norway: 385178
7. Germany: 357386
8. Finland: 338145
9. Poland: 312685
10. Italy: 301338

```

Funcția Reduce/Aggregate aplică succesiv o funcție asupra elementelor unei liste, reducând în final lista la un singur element. Unul din argumentele sale este o funcție care reduce două elemente la un singur element.

Cu ajutorul său putem calcula, de exemplu, produsul elementelor unei liste:

```

var list1 = "[ 1 2 3 4 ]".ToIntEnumF();
int p = list1.Reduce((x, y) => x * y); // 24

```

Mai exact, în acest caz funcția lucrează astfel:

- calculează $1 * 2$ și pune valoarea într-un acumulator: $a = 1 * 2 = 2$;
- calculează $a * 3$ și actualizează valoarea acumulatorului: $a = a * 3 = 6$;
- calculează $a * 4$ și actualizează valoarea acumulatorului: $a = a * 4 = 24$.

Dacă lista are un singur element, atunci acesta este rezultatul. Dacă lista primită este vidă, funcția aruncă o excepție: *System.InvalidOperationException: Sequence contains no elements*.

Să considerăm un alt exemplu, în care dorim să concatenăm elementele unei liste într-un sir de caractere, în care elementele să fie despărțite prin spațiu:

```

var list2 = "[ a b c d ]".ToStringEnumF();
string s = list2.Reduce((x, y) => $"{x} {y}"); // "a b c d"

```

Aici concatenarea de siruri funcționează deoarece și rezultatul și elementele sunt de același tip, *string*. Dacă elementele listei ar fi numere, ar apărea o eroare la compilare: *Cannot implicitly convert type 'string' to 'int'*.

Într-o variantă mai generală a aceleiași funcționalități, se primește o valoare inițială explicită pentru acumulator, nu primul element din listă:

```
double StDev(IEnumerable<double> list)
{
    int size = list.Count();
    double mean = list.Reduce(0.0, (x, y) => x + y) / (double)size;
    double difSquares = list.Reduce(0.0, (sum, item) => sum + (item - mean) * (item - mean));
    double variance = difSquares / (double)size;
    return Sqrt(variance);
}

var list1 = "[ 1 2 3 4 5 ]".ToDoubleEnumF();
var stdev = StDev(list1); // 1.4142135623731
WriteLine(stdev);
```

Reduce este o funcție foarte puternică, deoarece poate fi folosită și în loc de recursivitate. O funcție de calcul al factorialului se poate realiza după cum urmează, folosind și indecesii elementelor din listă:

```
int Factorial(int x) => Range(1, x).Reduce(1, (a, i) => a * i);
int f = Factorial(5); // 120
```

Metoda Range returnează o listă cu elementele de la 1 la x . Pentru a fi utilizată ca atare, trebuie inclusă directiva `using static System.Linq.Enumerable` în lista de namespace-uri ale programului.

3.2. Funcții de ordonare a elementelor

Din această categorie, menționăm funcțiile Reverse, care returnează o nouă listă care conține elementele în ordine inversă și OrderBy, care sortează elementele unei liste după aplicarea unei funcții asupra elementelor:

```
var r = new Random();
var list1 = Range(1, 10).Map(x => r.Next(100));
var list2 = list1.OrderBy(x => x); // [ 4 4 20 31 41 45 53 55 69 87 ]

var list3 = Range(1, 10);
var list4 = list3.Reverse(); // sau: OrderBy(x => -x); => [ 10 9 8 7 6 5 4 3 2 1 ]

var list5 = "[ 1 4 8 -2 5 ]".ToIntEnumF();
var list6 = list5.OrderBy(x => Abs(x)); // [ 1 -2 4 5 8 ]

var list7 = "[ Laboratorul 2 de IA ]".ToStringEnumF();
var list8 = list7.OrderBy(s => s.Length); // [ "2" "de" "IA" "Laboratorul" ]
```

3.3. Funcții de filtrare și combinare

Funcția Filter/Where returnează o nouă listă care conține doar elementele care satisfac un predicat:

```
var list1 = "[ cat dog chicken wolf ]".ToStringEnumF();
var list2 = list1.Filter(s => s.Length == 3); // [ "cat" "dog" ]
```

Funcția Zip combină două liste de aceeași dimensiune într-o singură listă:

```
var list3 = "[ one two three ]".ToStringEnumF();
var list4 = "[ eins zwei drei ]".ToStringEnumF();
var translation = list3.Zip(list4, (en, de) => $"{en} = {de}"); // [ "one = eins" "two = zwei" "three = drei" ]
```

Funcția Bind>SelectMany este utilizată pentru „a aplatiza” (flatten) o listă în care fiecare element este la rândul său o listă:

```
int[][] lists = {
    new[] {1, 2, 3},
    new[] {4},
    new[] {5, 6, 7, 8},
    new[] {9, 10};
var result = lists.Bind(x => x); // [ 1 2 3 4 5 6 7 8 9 10 ]
```

La fel ca și Map/Select, această funcție are o importanță specială în programarea funcțională. Vom reveni asupra ei în laboratorul 4.

3.4. Funcții statistice

Din acest tip de funcții, menționăm: Sum, Average, Min și Max, cu semnificațiile obișnuite:

```
var list = "[ 1 2 3 4 ]".ToIntEnumF();
int sum = list.Sum(); // 10
double avg = list.Average(); // 2.5
int min = list.Min(); // 1
int max = list.Max(); // 4
```

Aceleași operații se pot realiza și după aplicarea unei funcții asupra elementelor listei:

```
int sum2 = list.Sum(x => x * x); // 30
double avg2 = list.Average(x => x * x); // 7.5
```

3.5. Funcții de căutare și adăugare

Pentru adăugarea unui element la începutul unei colecții `IEnumerable` se folosește funcția Prepend. Pentru adăugarea la sfârșit, se folosește funcția Append.

Biblioteca *FunCs* mai adaugă unele funcții de căutare. *Find* cauță un element după o condiție și întoarce primul element care îndeplinește condiția respectivă. *FindIndex* întoarce indexul unui element căutat sau indexul primului element care îndeplinește o anumită condiție.

```
var list = "[ 2 3 4 ]".ToIntEnumF();
list = list.Append(5).Prepend(1); // [ 1 2 3 4 5 ]
int i1 = list.FindIndex(3); // 2
int i2 = list.FindIndex(9); // -1

var list2 = "[ -1 2.3 4.1 -10 1 0.01 4 ]".ToDoubleEnumF();
double found = list2.Find(x => Math.Abs(x) < 0.1); // 0.01
int i3 = list2.FindIndex(x => Math.Abs(x) < 0.1); // 5
```

Lista tuturor funcțiilor din *LINQ* poate fi consultată în documentația MSDN:
<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable>.

4. Tuple

O tuplă este o colecție ordonată de date utilizată pentru gruparea acestora. Tuplele pot avea elemente de tipuri diferite, inclusiv alte tuple, separate prin virgulă și așezate, optional, între paranteze, spre deosebire de liste, care au elemente de același tip. De exemplu:

```
var pair = ("unu", "doi");
var numbers = (1, 3, 5);
var mixedTuple = (1, 1.2, "abc");
var mixedTuple2 = (1.1, (3, 4, "cuvant"), new int[] { 10, 11 });
```

Pentru a accesa elementele unei tuple se pot folosi proprietățile `Item_i`:

```
var first = pair.Item1; // "unu"
var second = pair.Item2; // "doi"
var third = mixedTuple.Item3; // "abc"
```

Elementele unei tuple se mai pot extrage printr-o atribuire multiplă de valori:

```
(int x, int y, int z) = numbers; // x = 1, y = 3, z = 5
```

De asemenea, se pot atribui nume elementelor unei tuple, iar extragerea valorilor se face în mod explicit:

```
var pair = (First: "unu", Second: "doi");
var first = pair.First;
var second = pair.Second;
```

5. Aplicații

5.1. Să se determine dacă o listă reprezintă un palindrom. Un palindrom este o listă care, citită de la stânga la dreapta sau de la dreapta la stânga, rămâne aceeași.

Pentru testarea egalității unor secvențe `IEnumerable` după conținut, se poate folosi funcția `SequenceEqual`.

Exemplu:

```
Lista [ 1 2 3 4 3 2 1 ] este un palindrom  
Lista [ 1 2 3 ] nu este un palindrom
```

5.2. Realizați o funcție care duplică fiecare element al unei liste.

Exemplu:

```
var list = "[ 2 1 3 ]".ToIntEnumF();  
var result = Duplicate(list);  
WriteLine($"Lista cu elemente duplicate este: {result.ToStringF()}");
```

```
Lista cu elemente duplicate este: [ 2 2 1 1 3 3 ]
```

5.3. Generați cu ajutorul funcțiilor `Range` și `Map` o listă de n numere naturale aleatorii între 0 și m .

5.4. Realizați o funcție care șterge fiecare al treilea element dintr-o listă. A nu se confunda cerința de față cu situația în care trebuie eliminate elementele divizibile cu 3. Aici trebuie testat indexul elementelor. Se pot folosi funcțiile `Map` și `Filter`; se poate transforma mai întâi lista inițială într-o lista de tuple, de forma `(element, index)`.

Exemplu:

```
var list = Range(2, 15);  
var result = Drop3(list);  
WriteLine($"Lista cu elementele eliminate este: {result.ToStringF()}");
```

```
Lista cu elementele eliminate este: [ 2 3 5 6 8 9 11 12 14 15 ]
```

5.5. Rezolvați problema optimizării unei funcții $f : \mathbb{R} \rightarrow \mathbb{R}$ în sensul determinării argumentului x dintr-un interval dat $D \subset \mathbb{R}$ pentru care $f(x)$ este minimă sau maximă.

Programul poate avea următoarea structură:

```

private static IEnumerable<double> DoubleRange(double min, double max, double step)
{
    // de exemplu, pentru parametrii: min = -1, max = 1, step = 0.1, rezultatul trebuie să fie lista:
    // [ -1.0 -0.9 -0.8 -0.7 -0.6 -0.5 -0.4 -0.3 -0.2 -0.1 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0 ]
    // se pot folosi Range+Map sau yield
}

private static double Argmin(Func<double, double> f, double min, double max, double step)
{
    var range = DoubleRange(min, max, step);
    // se returnează x-ul pentru care f(x) este minimă
    // nu se poate folosi o buclă de tip imperativ: for, while etc.
    // se poate folosi funcția Reduce sau altă abordare funcțională (o funcție recursivă etc.)
}

private static double Argmax(Func<double, double> f, double min, double max, double step)
{
    var range = DoubleRange(min, max, step);
    // se returnează x-ul pentru care f(x) este maximă
}

public static void Problem5()
{
    double f1(double x) => x * x - x;
    var xargmin = Argmin(f1, -5, 5, 0.1);
    var fmin = f1(xargmin);
    WriteLine($"x = {xargmin:F3} => f = {fmin:F3}"); // x = 0.500 => f = -0.250

    double f2(double x) => 1 - Math.Abs(x);
    var xargmax = Argmax(f2, -5, 5, 0.1);
    var fmax = f2(xargmax);
    WriteLine($"x = {xargmax:F3} => f = {fmax:F3}"); // x = 0.000 => f = 1.000
}

```

Compunerea funcțiilor și evaluarea pasivă

1. Obiective

Obiectivele acestui laborator sunt legate de prezentarea modului în care funcțiile pot fi înlántuite și compuse în *LINQ* și a lucrului cu sevențe care sunt inițial doar definite și apoi evaluate numai în momentul utilizării lor efective.

2. Înlăntuirea funcțiilor

Înlăntuirea apelurilor de funcții conduce la o structură logică mai naturală și deci mai ușor de înțeles. Să considerăm un prim exemplu, o funcție care calculează dimensiunea unui director:

```
public static long SizeOfFolder(string folder)
{
    // ia toate numele de fișiere din directorul folder și subdirectoarele sale
    var filesInFolder = Directory.GetFiles(folder, "*.*", SearchOption.AllDirectories);
    // transformă numele fișierelor în obiectele FileInfo corespunzătoare
    var fileInfo = filesInFolder.Map(file => new FileInfo(file));
    // transformă vectorul de obiecte FileInfo într-un vector de dimensiuni
    var fileSizes = fileInfo.Map(info => info.Length);
    // sumează dimensiunile și returnează suma
    return fileSizes.Sum();
}

public static void Ex1()
{
    var totalSize = SizeOfFolder(@"d:\TestSize");
    WriteLine($"Total size = {totalSize} bytes ({totalSize / (1024 * 1024):F1} MB)");
}
```

În această funcție, fiecare rezultat intermediu este pasat funcției următoare, astfel încât avem nevoie de mai multe instrucțiuni distincte. În *LINQ*, metodele sunt fluente (fluent), adică pot fi înlántuite. Acestea se aplică unor obiecte `IEnumerable` și returnează tot obiecte `IEnumerable`. Folosind această facilitate, funcția de mai sus poate fi rescrisă astfel:

```
public static long SizeOfFolderFluent(string folder)
{
    return Directory.GetFiles(folder, "*.*", SearchOption.AllDirectories)
        .Map(file => new FileInfo(file))
        .Map(info => info.Length)
        .Sum();
}
```

Pe lângă beneficiile aduse de claritate și concizie, înlănțuirea funcțiilor are totuși dezavantajul că procesul de depanare (debugging) devine mai dificil, deoarece programatorul nu mai poate inspecta rezultatele intermediare ale prelucrărilor.

Programatorul își poate adăuga propriile funcții de prelucrare a colecțiilor prin așa-numitele extension methods din C#. Acestea trebuie introduse într-o clasă statică și au drept caracteristică faptul că primul parametru este precedat de cuvântul cheie `this`, iar tipul său este tipul căruia î se va adăuga metoda.

Exemplul următor adaugă metoda `Cube` tipului `int`:

```
public static class Examples
{
    private static int Cube(this int x) => x * x * x;
}
```

În continuare, fie o funcție care calculează numărul de caractere cu care este scris cubul unui număr întreg:

```
private static int Cube(this int x) => x * x * x;

public static void Ex2()
{
    var x = 5;
    var l3 = x.Cube().ToString().Length;
    WriteLine($"x = {x}, l3 = {l3}");
}
```

Un alt exemplu este o extension method pentru tipul `IEnumerable`:

```
private static IEnumerable<int> Cube(this IEnumerable<int> list)
{
    return list.Map(x => x * x * x);
}

public static void Ex3()
{
    var list = "[ 1 2 3 4 ]".ToIntEnum();
    var list3 = list.Cube();
    WriteLine($"{list.ToStringF()} => {list3.ToStringF()}"); // [ 1 2 3 4 ] => [ 1 8 27 64 ]
}
```

3. Compunerea funcțiilor

O altă caracteristică fundamentală a programării funcționale este compunerea mai multor funcții pentru a realiza funcții mai complexe și cu o putere de prelucrare mai mare. În biblioteca `FunCs` este implementată o extension method pentru obiectele de tip funcție numită `To`, care creează o nouă funcție prin compunerea a două funcții.

```

public static void Ex4()
{
    Func<double, double> Cube = n => n * n * n;
    Func<double, string> String = n => n.ToString();
    Func<string, int> Length = s => s.Length;

    var CubeLength = Cube.ToString().To(Length);
    // echivalent cu: var CubeLength = Cube.ToString().To(Length);

    double x = 5.1;
    var l3 = CubeLength(x);

    WriteLine($"x = {x}, l3 = {l3}");
}

```

În acest caz, `CubeLength` este o funcție compusă, care primește argumentul primit de `Cube` și întoarce rezultatul calculat de `Length`.

Apelarea succesivă a funcțiilor poate fi realizată împreună cu funcțiile de transformare ale listelor. În exemplul următor, aplicăm în ordine trei funcții simple asupra elementelor unei liste de numere reale:

```

var functions = new List<Func<double, double>>
{
    Abs,
    Sqrt,
    x => x + 1
};

var list = "[ 1 -2.5 4 ]".ToDoubleEnumF();
var res = list.Map(x => functions.Reduce(x, (a, f) => f(a)));
WriteLine(res.ToStringF(3)); // [ 2.000 2.581 3.000 ]

```

4. Evaluarea pasivă

Obiectele de tip vector (array) sau listă sunt evaluate în mod *activ* (*eager*). Când se definește, de exemplu, o listă, ea se creează efectiv în memorie, chiar dacă nu este folosită ulterior. Există însă situații în care dorim să evaluăm o expresie doar atunci când este nevoie. De exemplu, putem să definim o secvență foarte mare de date, dar dorim să se calculeze aceste valori doar în momentul în care le folosim efectiv. Acest mod de evaluare se numește *pasiv* sau *întârziat* (*lazy*) și poate conduce la o scădere foarte mare a necesarului de memorie a unui program.

Colecțiile `IEnumerable` sunt evaluate pasiv. Prin urmare, pe lângă modurile de definire pe care le-am considerat până acum, se pot defini colecții infinite. Biblioteca `Funcs` include funcția `InfiniteF.Define` care definește o secvență infinită. În exemplul următor, se definește sirul numerelor naturale.

```

var seqInfinite = InfiniteF.Define(n => n + 1);

```

Parametrul n din expresia lambda merge de la 0 la infinit. Expresia definește termenul corespunzător lui n . Sirul începe de la 1, deci termenul corespunzător lui $n = 0$ este $t_0 = 0 + 1 = 1$, termenul corespunzător lui $n = 1$ este $t_1 = 1 + 1 = 2$ și.m.d.

O secvență (infinită sau nu) poate fi parcursă în maniera dorită cu ajutorul funcțiilor Skip, care permite saltul peste un număr de elemente, și Take, care evaluează un număr de elemente:

```
var list = seqInfinite.Skip(3).Take(4);
WriteLine(${list.ToStringF()}); // [ 4 5 6 7 ]
```

O colecție poate fi construită în mod recursiv. Următoarea funcție întoarce toate fișierele dintr-un director și subdirectoarele acestuia:

```
private static IEnumerable<string> AllFilesUnder(string basePath)
{
    foreach (var file in Directory.GetFiles(basePath)) // toate fișierele din directorul de bază
        yield return file;

    foreach (var subdir in Directory.GetDirectories(basePath))
        foreach (var file in AllFilesUnder(subdir)) // toate fișierele din subdirectoare
            yield return file;
}

public static void Ex5()
{
    var allFiles = AllFilesUnder(@"d:\TestDir");
    foreach (var file in allFiles)
        WriteLine(file);
}
```

5. Aplicații

5.1. La un magazin, dacă un client cumpără cel puțin 3 produse, i se acordă o reducere de 10% pentru al doilea cel mai scump produs și o reducere de 20% pentru al treilea cel mai scump produs. Să se calculeze suma totală pe care trebuie să o plătească clientul.

Pentru rezolvare, nu extrageți primele două elemente pentru a le prelucra separat, ci operați o transformare asupra întregii liste de prețuri.

Definiți funcția DiscountedSum ca o extension method pentru I Enumerable, apelabilă la fel ca funcția standard Sum.

Exemplu:

```
var list = "[ 20 10 30 ]".ToDoubleEnumF(); WriteLine($"Preturile articolelor fără reduceri: {list.ToStringF(2)}");
double sum = list.Sum(); WriteLine($"Suma fără reduceri: {sum:F2}");
double discSum = list.DiscountedSum(); WriteLine($"Suma finală: {discSum:F2}");

list = "[ 10.5 20.5 ]".ToDoubleEnumF(); WriteLine($"\\r\\nPreturile articolelor fără reduceri: {list.ToStringF(2)}");
sum = list.Sum(); WriteLine($"Suma fără reduceri: {sum:F2}");
discSum = list.DiscountedSum(); WriteLine($"Suma finală: {discSum:F2}");
```

Preturile articolelor fără reduceri: [20.00 10.00 30.00]
--

Suma fara reduceri: 60.00

Suma finală: 56.00

Preturile articolelor fara reduceri: [10.50 20.50]

Suma fara reduceri: 31.00

Suma finală: 31.00

5.2. Realizați un program care calculează valoarea reală a unei fracții continue. O fracție continuă este o expresie obținută în urma unui proces iterativ de reprezentare a unui număr ca suma unor numere întregi și inverse ale unor întregi, de forma:

$$x = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \dots}} = [a_0, a_1, a_2, \dots],$$

unde $a_0 \in \mathbb{Z}$ și $a_i \in \mathbb{N}, \forall i \geq 1$.

- a)** Folosiți mai întâi abordarea imperativă, cu o buclă for.
- b)** Rezolvați aceeași problemă în mod funcțional, folosind funcția Reduce.

Exemplu:

```
WriteLine($"Valoarea lui pi este:\t{Math.PI}");

double pi1 = ContFractionI(new int[] { 3, 7, 15 });
WriteLine($"Valoarea fractiei este:\t{pi1} (i)");

double pi2 = ContFractionF(new int[] { 3, 7, 15 });
WriteLine($"Valoarea fractiei este:\t{pi2} (f1)");

double pi3 = ContFractionF(new int[] { 3, 7, 15, 1, 292, 1 });
WriteLine($"Valoarea fractiei este:\t{pi3} (f2)");
```

```
Valoarea lui pi este:      3.14159265358979
Valoarea fractiei este:   3.14150943396226 (i)
Valoarea fractiei este:   3.14150943396226 (f1)
Valoarea fractiei este:   3.14159265392142 (f2)
```

5.3. Funcția exponențială poate fi calculată cu ajutorul unei serii Taylor:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = \frac{x^0}{0!} + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

Definiți o secvență infinită ai cărei termeni corespund termenilor seriei de mai sus. Realizați o funcție care aproximează funcția exponențială cu o precizie dorită, precizia însemnând în acest caz faptul că termenii seriei mai mici decât această valoare sunt ignorați.

Indicație: Pentru definirea secvenței infinite, observați că termenul generic corespunzător lui n poate fi scris recursiv:

$$\frac{x^n}{n!} = \frac{x}{n} \cdot \frac{x^{n-1}}{(n-1)!}$$

De exemplu, factorialul poate fi scris recursiv ca: $n! = n \cdot (n - 1)!$ iar funcția care îl calculează este:

```
public static BigInteger Factorial(int n) => (n <= 1) ? 1 : (n * Factorial(n - 1));
```

Realizați o funcție similară pentru termenul generic al seriei de mai sus, fără a folosi separat funcția de calcul al factorialului. La rezolvare, se poate utiliza funcția TakeWhile.

Exemplu:

```
var x = 2.0;      // exponentul din e^x
var p = 1e-6;    // precizia

var e1 = ExpF(x, p);
WriteLine($"Serie:\t{e1:F6}"); // 7.389056

var e2 = Math.Exp(x);
WriteLine($"Exact:\t{e2:F6}"); // 7.389056
```

5.4. Date fiind toate tipurile de bancnote ale unei țări și o valoare, determinați numărul minim de bancnote necesar pentru plata acelei valori.

Nu folosiți construcții imperative (de exemplu, bucle for sau while). Folosiți secvențe I Enumerable, recursivitatea și/sau funcția Reduce. Alte funcții utile din I Enumerable pot fi First și Append.

Algoritmul de descompunere este greedy: se sortează descrescător tipurile de bancnote, se calculează numărul maxim de bancnote de tipul cel mai mare care pot fi utilizate la un moment dat, după care se scade suma identificată din suma inițială și se trece la următorul tip de bancnotă.

Exemplu: descompunerea valorii 2778 cu tipurile de bancnote: 500, 200, 100, 50, 10, 5, 1.

2778 / 500	=	5 rest 278
278 / 200	=	1 rest 78
78 / 100	=	0 rest 78
78 / 50	=	1 rest 28
28 / 10	=	2 rest 8
8 / 5	=	1 rest 3
3 / 1	=	3 rest 0

```
var amount = 2778;
var money = "[ 500 200 100 50 10 5 1 ]".ToIntEnumF();
var result = Decompose(amount, money); // [ 5 1 0 1 2 1 3 ]
```

Functori și monade. Obiecte *Option*

1. Obiective

În acest laborator, vom prezenta concepțele de *functor* și *monadă*, tipice programării funcționale, împreună cu un tip special de obiecte (*Option*), care poate încapsula sau nu o valoare.

2. Tipul *Option*

În multe situații apare necesitatea de a trata valori nedefinite. Majoritatea limbajelor imperative folosesc în acest sens cuvântul cheie *null* sau unul echivalent. Să considerăm ca exemplu o problemă des întâlnită: citirea unui sir de caractere de la tastatură și convertirea lui într-un număr întreg pozitiv. Conversia este posibilă doar dacă sirul de caractere reprezintă un număr valid. În C#, conversia se poate face astfel:

```
private static int? ToNatural(string s) // int? = Nullable<int>
{
    bool ok = int.TryParse(s, out int result);
    if (ok && result >= 0)
        return result;
    else
        return null;
}
```

sau folosind excepții:

```
private static int ToNatural(string s)
{
    bool ok = int.TryParse(s, out int result);
    if (ok && result >= 0)
        return result;
    else
        throw new Exception($"Argument {s} is not a valid natural number");
}
```

Începând cu versiunea 8, în C# au fost adăugate o serie de metode pentru a identifica explicit situațiile în care un obiect, atât de tip valoare cât și de tip referință, poate fi nul, astfel încât compilatorul să le recunoască prin analiză statică și să scadă incidența erorilor legate de obiecte nule.

În multe limbaje funcționale, se folosește tipul *Option* sau echivalent (*Maybe*, *Optional*), cu două posibile valori: *Some* (*Just*) și *None* (*Nothing*). *Some* poate conține la rândul său orice tip de valoare.

În biblioteca *FunCs* este definit tipul *OptionF*, care poate fi utilizat ca în exemplul de mai jos, echivalentul funcțional al metodei de conversie a unui string într-un număr natural:

```
private static OptionF<int> ToNatural(string s)
{
    bool ok = int.TryParse(s, out int result);
    if (ok && result >= 0)
        return OptionF<int>.Some(result); // crearea unui obiect de tip Some, cu valoarea result
    else
        return OptionF<int>.None(); // crearea unui obiect de tip None
}

private static OptionF<int> ReadNatural()
{
    Write("Input a positive integer: ");
    var s = ReadLine();
    return ToNatural(s);
}

public static void TestOption()
{
    var rez = ReadNatural();
    if (rez.IsSome)                                // test dacă opțiunea are o valoare
        WriteLine($"Some: {rez.Value}");           // regăsirea valorii încapsulate
    if (rez.IsNone)                                // test dacă opțiunea nu conține nicio valoare
        WriteLine("None");
}
```

Se poate observa că tipul *OptionF* este un container (sau wrapper) care poate conține orice alt tip de obiect. De asemenea, se poate testa în mod explicit dacă un obiect de tip opțiune conține sau nu un obiect „valoare”.

În metoda următoare, se prezintă mai multe operații cu obiecte *IEnumerable* și *OptionF*:

```
public static void TestListOfOptions()
{
    I Enumerable<int> l1 = "[ 1 2 2.5 abc -5 14 ]".ToStringEnumF(' ')
        .Map(x => ToNatural(x))      // se convertesc elementele string în elemente OptionF<int>
        .Filter(x => x.IsSome)       // se rețin doar elementele de tip Some
        .Map(x => x.Value);         // se transformă OptionF<int> în valorile corespunzătoare int
    WriteLine(l1.ToStringF());      // [ 1 2 14 ]

    OptionF<int> o1 = ToNatural("1");
    WriteLine(o1);                // Some(1)

    OptionF<int> o2 = ToNatural("abc");
    WriteLine(o2);                // None

    // obiectele OptionF pot conține în Some alte obiecte de tip OptionF
    OptionF<OptionF<int>> o3 = OptionF<OptionF<int>>.Some(o1);
    WriteLine(o3);                // Some(Some(1))
}
```

```

OptionF<int> o4 = o3.Bind(x => x); // Bind aplatizează obiectele cu valori imbricate/nested
WriteLine(o4.ToStringF());           // [1]

// pot exista colecții I Enumerable de obiecte OptionF și invers
IEnumerable<OptionF<int>> l2 = new OptionF<int>[] { o1, o2, o1, o2 }.AsEnumerable();
IEnumerable<int> l3 = l2.FilterSome();    // se rețin doar elementele de tip Some, iar din acestea doar valorile
WriteLine(l3.ToStringF());             // [1 1]
}

```

În secvența de cod următoare, se observă cum se poate lucra cu funcția Map din OptionF:

```

private static OptionF<string> StringNotNumber(string s)
{
    if (!double.TryParse(s, out double d)) // s nu reprezintă un număr
        return OptionF<string>.Some(s);
    else
        return OptionF<string>.None();
}

public static void TestOptionMap()
{
    OptionF<string> optStr = StringNotNumber("abc");
    OptionF<int> optLen = optStr.Map(x => x.Length); // se transformă OptionF<string> în OptionF<int>
    WriteLine($"'{optStr}' -> {optLen}");               // Some(abc) -> Some(3)

    optStr = StringNotNumber("123");
    optLen = optStr.Map(x => x.Length);
    WriteLine($"'{optStr}' -> {optLen}");               // None -> None
}

```

3. Functori

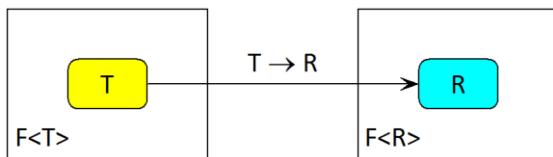
Atât listele (I Enumerable) cât și opțiunile (OptionF) sunt obiecte-container, care conțin alte obiecte de orice tip. O listă conține o serie de obiecte, iar o opțiune conține un singur obiect, în *Some*. Aceste tipuri de obiecte-container respectă o serie de proprietăți care le fac să aparțină unor categorii speciale definite în teoria programării funcționale.

Am văzut că Map aplică o funcție asupra elementelor interne ale unei liste sau opțiuni, rezultând o altă listă sau opțiune. Să notăm cu $F < T >$ tipul acestui obiect-container care conține elemente de tip T . Putem nota funcția care transformă elementele drept $T \rightarrow R$. Tipul obiectului-container rezultat după aplicarea funcției Map va fi $F < R >$. Map trebuie să aplique funcția elementelor interne din F și să nu aibă efecte secundare.

Un astfel de tip F pentru care este definită o funcție Map se numește *functor*.

Figura următoare prezintă în mod succint structura unui astfel de obiect.

Functor
 $\text{Map}(F<T>, T \rightarrow R) = F<R>$



Legile functorilor:

1. *Păstrarea identității*: dacă valorile interne ale unui functor sunt transformate în ele însese, rezultatul trebuie să fie același functor.

```
// functor.Map(x => x) == functor

OptionF<int> functor1 = OptionF<int>.Some(3);      // Some(3)
OptionF<int> functor2 = functor1.Map(x => x);       // Some(3)
```

2. *Păstrarea compunerii*: rezultatul a două operații successive de transformare trebuie să fie același cu rezultatul aplicării primei funcții asupra rezultatului celei de a doua.

```
// functor.Map(x => (f o g)(x)) == functor.Map(f).Map(g)

OptionF<int> functor = OptionF<int>.Some(3);
Func<int, int> f = x => x * x;
Func<int, int> g = x => x + 1;
OptionF<int> r1 = functor.Map(x => (f.To(g))(x));      // Some(10)
OptionF<int> r2 = functor.Map(x => f(x)).Map(x => g(x)); // Some(10)
```

4. Monade

În exemplele prezentate în laboratoarele 2 și 4, am văzut cum poate fi folosită funcția Bind pentru a evita lucrul cu obiecte imbricate (nested), de exemplu, liste de liste sau opțiuni cu valori de tip opțiune.

Să notăm cu $M<T>$ tipul unui astfel de obiect-container care conține elemente de tip T . Bind folosește o funcție de transformare de tip $T \rightarrow M(R)$. Tipul obiectului-container rezultat după aplicarea funcției Bind va fi $M<R>$.

Să mai considerăm o funcție simplă numită Return, care încapsulează o valoare T într-un obiect-container $M<T>$. Ca exemple de funcții Return putem menționa orice funcție care creează o enumerare populată de obiecte de un anumit tip, precum:

```
IEnumerable<int> list = "[ 1 2 3 ]".ToIntEnum();
```

sau o funcție care creează o opțiune setându-i valoarea internă:

```
OptionF<int> opt = OptionF<int>.Some(1);
```

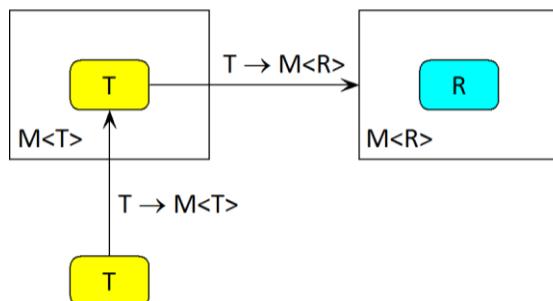
Un tip M pentru care sunt definite funcțiile `Return` și `Bind` se numește *monadă*.

Figura următoare prezintă în mod succint structura unui astfel de obiect.

Monadă

$\text{Return}(T) = M<T>$

$\text{Bind}(M<T>, T \rightarrow M<R>) = M<R>$



Legile monadelor:

1. Identitate la stânga

```
// Return(t).Bind(f) == f(t)

int t = 3;
Func<int, OptionF<int>> Return = x => OptionF<int>.Some(x);
Func<int, OptionF<int>> f = x => OptionF<int>.Some(x * x);
var r1 = Return(t).Bind(f); // Some(9)
var r2 = f(t);           // Some(9)
```

2. Identitate la dreapta

```
// m == m.Bind(Return)

OptionF<int> m1 = OptionF<int>.Some(3); // Some(3)
Func<int, OptionF<int>> Return = x => OptionF<int>.Some(x);
var m2 = m1.Bind(Return);                 // Some(3)
```

3. Asociativitate

```
// m.Bind(f).Bind(g) == m.Bind(x => f(x).Bind(g))

OptionF<int> m = OptionF<int>.Some(3);
Func<int, OptionF<int>> f = x => OptionF<int>.Some(x * x);
Func<int, OptionF<int>> g = x => OptionF<int>.Some(x + 1);
OptionF<int> r1 = m.Bind(f).Bind(g);      // Some(10)
OptionF<int> r2 = m.Bind(x => f(x).Bind(g)); // Some(10)
```

Folosirea functorilor și/sau monadelor permite lucrul la un nivel superior de abstractizare, care nu mai depinde de tipurile concrete ale obiectelor încapsulate. Pe lângă liste sau opțiuni, există și alte monade folosite în mod curent în programare. De exemplu, `Func<T>` nu este un T , ci o

expresie care trebuie evaluată pentru a obține un T. De asemenea, Task<T> nu este un T, ci o promisiune că la un moment dat în viitor vom avea disponibil un T.

5. Aplicații

5.1. Realizați o funcție care testează dacă un an este bisect și returnează Some dacă este bisect și None dacă nu este.

În calendarul gregorian, majoritatea anilor multipli de 4 sunt bisecți. În fiecare an bisect, luna februarie are 29 de zile în loc de 28. Adăugarea unei zile suplimentare la fiecare an compensează faptul că perioada de 365 de zile este mai scurtă decât un an tropic cu aproape 6 ore. Sunt necesare unele excepții de la această regulă simplă, întrucât durata anului tropical este puțin mai mică decât 365.25 zile. Eroarea se acumulează și pe o perioadă de 4 secole ea ajunge la 3 zile. Pentru aceasta, calendarul gregorian renunță la trei ani bisecți în 400 de ani. Aceasta se face prin renunțarea la ziua de 29 februarie în anii multipli de 100 și care nu sunt și multipli de 400. Anii 2000 și 2400 sunt ani bisecți, dar 1800, 1900, 2100, 2200, 2300 și 2500 nu sunt. (https://ro.wikipedia.org/wiki/An_bisect)

Exemplu:

```
public static void Problem1()
{
    var list = "[ 1900 2000 2017 2020 ]".ToIntEnumF()
        .Map(y => LeapYear(y))
        .FilterSome();
    WriteLine(list.ToStringF()); // [ 2000 2020 ]
}
```

5.2. Se dă o listă de studenți cu rezultatele obținute la un examen:

```
private class Student
{
    public readonly string Name;
    public readonly string Result;

    public Student(string name, string result)
    {
        Name = name;
        Result = result;
    }
}

public static void Problem2()
{
    var students = new List<Student>
    {
        new Student("Alex Neagoe", "10"),
        new Student("Carla Stoenescu", "3.50"),
        new Student("Flavius Predoiu", "absent"),
        new Student("Doina Vasiliu", "9.75"),
        new Student("Dan Draghicescu", "5"),
        new Student("Maria Raducanu", "7.25")
    };
}
```

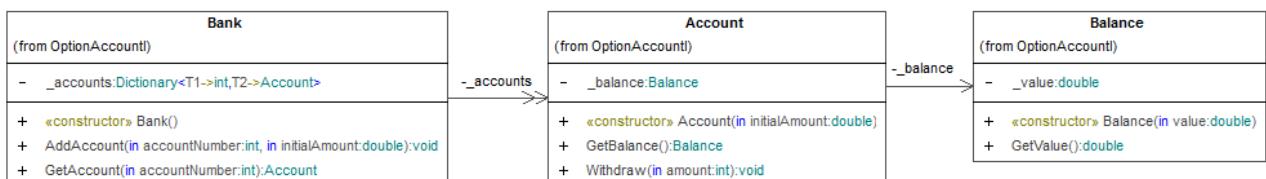
```
}
```

Completați funcția de mai sus cu o singură instrucțiune cu funcții înlăncuite, care să transforme și să filtreze lista students astfel încât să poată fi afișați în final doar studenții cu medii mai mari sau egale cu 5, iar pentru aceștia, mediile să fie rotunjite la cel mai apropiat întreg:

```
Alex Neagoe - 10
Doina Vasiliu - 10
Dan Draghicescu - 5
Maria Raducanu - 7
```

Lucrați cu opțiuni. În funcțiile înlăncuite, puteți folosi tuple în loc de obiecte de tip Student. Puteți adăuga și alte funcții în program.

5.3. Fie următorul program, proiectat într-o manieră imperativă, care realizează diferite operațiuni financiare:



```

namespace OptionAccount
{
    public class Bank
    {
        private Dictionary<int, Account> _accounts;

        public Bank()
        {
            _accounts = new Dictionary<int, Account>();
        }

        public void AddAccount(int accountNumber, double initialAmount)
        {
            _accounts[accountNumber] = new Account(initialAmount);
        }

        public Account GetAccount(int accountNumber)
        {
            if (!_accounts.ContainsKey(accountNumber))
                return null;

            return _accounts[accountNumber];
        }
    }

    public class Account
    {
        private Balance _balance;
    }
}
  
```

```

public Account(double initialAmount)
{
    _balance = new Balance(initialAmount);
}

public Balance GetBalance()
{
    if (_balance.GetValue() < 0)
        return null;

    return _balance;
}

public void Withdraw(int amount)
{
    _balance = new Balance(_balance.GetValue() - amount);
}

public class Balance
{
    private double _value;

    public Balance(double value)
    {
        _value = value;
    }

    public double GetValue()
    {
        return _value;
    }
}

public class Operations
{
    public static void Test()
    {
        var bank = new Bank();
        bank.AddAccount(1, 100);
        bank.AddAccount(2, 10);
        PrintAmount(bank, 1);
        bank.GetAccount(1).Withdraw(1000); PrintAmount(bank, 1);
        PrintAmount(bank, 10);
    }
}

private static void PrintAmount(Bank bank, int accountNumber)
{
    var account = bank.GetAccount(accountNumber);

    bool ok = false;
    if (account != null)
    {
        var balance = account.GetBalance();
        if (balance != null)
        {
            double a = balance.GetValue();
        }
    }
}

```

```

        WriteLine($"Soldul este pozitiv: {a}");
        ok = true;
    }

    if (!ok)
        WriteLine("Soldul este negativ sau nu poate fi interogat");
}
}

```

Refactorizați programul într-o manieră funcțională:

- Clasele Bank, Account și Balance vor conține doar proprietățile (câmpurile), cu vizibilitate publică (fără alte metode). În mod normal, proprietățile ar trebui să fie read-only, însă în acest caz, vom face o excepție și le vom defini ca read-write (get-set). În cadrul operațiilor pe care va trebui să le efectuăm, vom modifica proprietățile obiectelor imbricate (de tipul a.B.C). În programarea funcțională, aceste operațiuni se fac cu așa-numitele *lentile* (lenses). Pentru aplicația de față, această metodă ar fi prea complexă. În plus, în C# modificarea proprietăților se face ușor și natural, spre deosebire de limbajele în care imutabilitatea este implicită. Deci trebuie să reținem că aici vom încălca recomandările programării funcționale în beneficiul simplității;
- Toate metodele de procesare vor fi statice și vor fi mutate într-o clasă distinctă, de exemplu, chiar în clasa Operations. Dacă în abordarea POO o metodă era în clasa X, în abordarea PF metoda va primi un prim argument de tipul X (obiectul pentru care trebuie să realizeze prelucrările);
- Constructorii vor fi înlocuiți de metode statice de tipul CreateX, care returnează obiectele create de tipul corespunzător (X);
- Operațiile care pot întoarce sau nu rezultate valide (GetAccount, GetBalance) vor întoarce obiecte de tip OptionF;
- Metoda de test PrintAmount(Bank bank, int accountNumber) trebuie să conțină o singură operațiune cu funcții înlănțuite în care să se acceseze contul, obiectul de sold (balance) și valoarea sa. În final, rezultatul va fi de tip OptionF, cu IsSome dacă toate informațiile și prelucrările au fost corecte și IsNone în caz contrar. Aici se pot folosi funcțiile tipice functorilor și monadelor, Map și Bind.

Potrivirea modelelor

1. Obiective

Potrivirea modelelor este echivalentă cu utilizarea unor instrucțiuni condiționale mai avansate și mai flexibile. Reprezintă verificarea unei secvențe de simboluri pentru a determina dacă conține anumite elemente într-o anumită ordine, adică un model. În acest laborator, vom prezenta modalitatea de lucru cu un algoritm de potrivire a modelelor care poate aplica eficient reguli sau modele multiple pentru o mulțime de obiecte sau fapte dintr-o bază de cunoștințe.

2. Potrivirea modelelor cu *switch – when*

Potrivirea modelelor (pattern matching) reprezintă verificarea unei secvențe de simboluri pentru a determina dacă conține anumite elemente într-o anumită ordine, adică un model. O potrivire poate reprezenta și deconstrucția secvenței în părțile sale constitutive, de exemplu identificarea primului element al unei liste și restul listei, elementele distincte ale unei tuple etc.

La un nivel simplificat, potrivirea modelelor este asemănătoare unei instrucțiuni switch, însă este mai puternică. Începând cu versiunea 7, C# permite, pe lângă instrucțiunea switch clasică și testarea unor condiții suplimentare cu instrucțiunea when. Pentru a prezenta sintaxa unei astfel de expresii, să considerăm un exemplu simplu, în care se identifică un județ pe baza indicativului de pe plăcuța de înmatriculare:

```
public static string Judet1(string cod)
{
    switch (cod)
    {
        case "IS": return "Iasi";
        case "SV": return "Suceava";
        case "B": return "Bucuresti";
        default: return "Alt judet";
    }
}

public static string Judet2(string cod)
{
    var d = new Dictionary<string, string>
    {
        { "IS", "Iasi" },
        { "SV", "Suceava" },
        { "B", "Bucuresti" }
    };
}
```

```

switch (cod)
{
    case string c when d.ContainsKey(c): // c ia valoarea lui cod
        return d[c];

    case string c when c.Length != 2:
        return "Cod invalid";

    default:
        return "Alt judet";
}

public static void Ex1()
{
    WriteLine(Judet1("IS"));           // Iasi
    WriteLine(Judet1("VS"));           // Alt judet
    WriteLine(Judet1("Arad"));         // Alt judet

    WriteLine(Judet2("IS"));           // Iasi
    WriteLine(Judet2("VS"));           // Alt judet
    WriteLine(Judet2("Arad"));         // Cod invalid
}

```

3. Potrivirea modelelor cu algoritmul *Rete*

3.1. Algoritmul *Rete* și limbajul *CLIPS*

În cele ce urmează, vom prezenta niște scenarii de potrivire a modelelor mai puternice. Deoarece acestea nu sunt incluse în platforma .NET, vom folosi facilitățile din biblioteca *FunCs*. Aceste metode de potrivire a modelelor se bazează pe algoritmul *Rete*, implementat de *CLIPS*, un instrument pentru crearea sistemelor expert, dezvoltat de Software Technology Branch, NASA Lyndon B. Johnson Space Center, cu scopul facilitării realizării de programe care modelează cunoașterea și experiența umană. *CLIPS* este un acronim pentru *C Language Integrated Production System*.

Rete este un algoritm de potrivire a modelelor pentru implementarea sistemelor bazate pe reguli. Algoritmul a fost dezvoltat pentru a aplica eficient multe reguli sau modele la multe obiecte sau fapte dintr-o bază de cunoștințe. Permite, de asemenea, găsirea unor soluții multiple pentru astfel de probleme.

3.2. Fapte și variabile

Faptele cu care vom lucra sunt constituite din zero sau mai multe câmpuri separate de spații, de exemplu: "1 2 3", "abc efg", "a 1.4 15 -6 alte cuvinte".

Elementele a căror valoare dorim să o determinăm prin potrivirea modelelor sunt echivalente variabilelor și se notează cu un semn de întrebare pus înaintea numelui, de exemplu: ?varsta, ?loc_nastere, ?specie. Aceste variabile lor lua valoarea unui singur câmp dintr-un fapt.

Dacă dorim ca o variabilă să poată conține orice număr de câmpuri (zero, unu sau mai multe), numele acesteia trebuie precedat și de simbolul dolar, de exemplu: \$?text.

Uneori, dorim să identificăm unul sau mai multe câmpuri dintr-un fapt, dar nu ne interesează valorile lor specifice. În această situație se folosesc variabile libere, notate cu ? sau \$? și fără un nume de variabilă.

Să presupunem că avem un fapt care reprezintă numele unei persoane, compus din numele de familie, inițiala tatălui și unul sau mai multe prenume și că dorim să aflăm numele de familie. Faptele pot fi de tipul: "Ionescu T. George" sau "Popescu S. Andrei Sebastian". Potrivirea se face cu modelul "?nume \$?", unde variabila nume va lua valorile Ionescu, respectiv Popescu.

Dacă dorim să identificăm primul prenume al unei persoane, vom folosi modelul "? ? ?prenume \$?", unde variabila prenume va lua valorile George, respectiv Andrei.

Dacă se folosesc în același timp două variabile libere multi-câmp, rezultatele pot fi multiple. De exemplu, cu faptul "1 2 3" și modelul "\$? ?x \$?", variabila x va lua trei valori: 1, 2, respectiv 3, deoarece prima variabilă liberă poate înlocui zero sau mai multe câmpuri, iar a doua poate și ea înlocui zero sau mai multe câmpuri. În acest caz, numărul de soluții va fi egal cu numărul de câmpuri din fapt.

4. Potrivirea modelelor pe liste

Operațiile pe care le-am prezentat mai sus pot fi aplicate pe colecții de tip `IEnumerable` în C#, cu ajutorul bibliotecii *FunCs*. Mai jos se prezintă câteva exemple de utilizare:

```
private static void MatchList()
{
    var list = "[ 1 2 3 4 5 6 7 8 9 10 ]".ToStringEnumF();
    list.MatchF("?head ? $rest ? ?last", out var head, out var rest, out var last);
    WriteLine($"{head} - {rest} - {last}"); // 1 - 3 4 5 6 7 8 - 10

    list.MatchF(" ?x $? ?y", out Dictionary<string, string> matches);
    WriteLine($"{matches["?x"]} {matches["?y"]}); // 1 10

    list.MatchF(" ?x ?y $?w ?z", out var x, out var y, out var w, out var z);
    WriteLine($"{x} - {y} - {w} - {z}"); // 1 - 2 - 3 4 5 6 7 8 9 - 10
}
```

Un alt exemplu este determinarea combinărilor de n elemente luate câte k . Pentru soluții multiple, se instanțiază un obiect de tip `ExpertMatchF`, iar rezultatele multiple sunt disponibile într-o listă de dicționare, câte un dicționar pentru fiecare soluție. Valoarea fiecărei variabile dintr-un dicționar se poate regăsi folosind numele acesteia:

```
public static void Combinations()
{
    var em = new ExpertMatchF("1 2 3 4 5"); // "[" și "]" nu sunt necesare aici
    var patterns = new List<string> { "$? ?x $? ?y $?" }; // combinații de 5 luate câte 2
    em.Match(patterns, out var matchList); // out List<Dictionary<string, string>> matchList)
    foreach (var m in matchList)
        WriteLine($"{m["?x"]} - {m["?y"]});
```

```
// 1 - 2
// 1 - 3
// 1 - 4
// 1 - 5
// 2 - 3
// 2 - 4
// 2 - 5
// 3 - 4
// 3 - 5
// 4 - 5
}
```

Întrucât listele sunt unele dintre cele mai folosite structuri de date în programarea funcțională, potrivirea modelor pentru liste este de asemenea foarte des întâlnită, mai ales în contextul funcțiilor recursive. De obicei, se identifică primul element din listă (`head`) și lista de elemente următoare (`tail`).

În *FunCs*, această facilitate este implementată pentru tipul `IEnumerable` generic. Celelalte tipuri de potrivire de modele se pot utiliza doar pentru colecții de tip string.

Exemplul următor prezintă o funcție care calculează recursiv suma elementelor dintr-o listă:

```
private static int MySum(IEnumerable<int> list)
{
    if (list.Count() == 0)
        return 0;
    list.MatchHeadTailF(out int head, out var tail);
    return head + MySum(tail);
}

public static void MySumTest()
{
    var list = "[ 1 2 3 4 5 ]".ToIntEnumF();
    int sum = MySum(list); // 15
}
```

Exemplul următor prezintă o funcție care aplică altă funcție `f`, primită ca parametru, asupra tuturor elementelor dintr-o listă. Funcția aceasta este echivalentă de fapt cu funcția `Map`:

```
private static IEnumerable<int> MyMap(this IEnumerable<int> list, Func<int, int> f)
{
    if (list.Count() == 0)
        return "["].ToIntEnumF(); // o listă goală, fără elemente
    list.MatchHeadTailF(out int head, out var tail);
    return tail.MyMap(f).Prepend(f(head));
}

public static void MyMapTest()
{
    var list = "[ 3 6 2 ]".ToIntEnumF();
    var list2 = list.MyMap(n => n * n); // [ 9 36 4 ]
    var list3 = list.MyMap(n => -n); // [ -3 -6 -2 ]
}
```

4. Potrivirea modelelor generală

4.1. Utilizarea multiplă a unei variabile

O proprietate foarte utilă și importantă pe care o au variabilele este aceea că pot fi folosite în mai multe modele, însă o variabilă are o singură valoare, indiferent în câte modele apare.

Fie următorul exemplu, în care avem o bază de fapte cu relațiile de rudenie între mai multe persoane:

```
(father Christopher Arthur) // Christopher este tatăl lui Arthur  
(father Christopher Victoria)  
...  
(mother Penelope Arthur)  
(mother Penelope Victoria)  
...
```

Să determinăm ce persoane sunt părinții aceluiași copil. Se observă că variabilele corespund exact locului din listă unde se află un anumit nume. De exemplu, pentru perechea de fapte:

```
(father Marco Sophia)  
(mother Lucia Sophia)
```

folosind variabilele `?f` pentru tată (`father`), `?m` pentru mamă (`mother`) și `?c` pentru copil (`child`), `?f` va fi Marco, `?m` va fi Lucia și `?c` va fi Sophia. Legarea celor două fapte se face prin utilizarea *aceleiași* variabile `?c`, care trebuie să ia același nume, în acest caz Sophia.

În același mod, regula descoperă și celealte perechi de fapte `father` și `mother`, care au același copil.

```
public static void Parents()  
{  
    // Încărcarea faptelor din fișier  
    var sr = new StreamReader("kinship.csv");  
    var lines = sr.ReadToEnd().Split("\r\n".ToCharArray(), StringSplitOptions.RemoveEmptyEntries);  
    sr.Close();  
  
    var em = new ExpertMatchF(lines.ToList());  
  
    // găsirea mamei unui anumit copil în fapte de tip:  
    // mother Penelope Arthur  
    string child = "Arthur";  
    em.MatchVar($"mother ?m {child}", out var mother);  
    WriteLine(mother); // Penelope  
  
    // găsirea părinților aceluiași copil  
  
    var patterns = new List<string>  
    {  
        "father ?f ?c",  
        "mother ?m ?c"  
    };
```

```

em.Match(patterns, out var matches);
foreach (var m in matches)
    WriteLine($"Father {m["?f"]}, mother {m["?m"]}, child {m["?c"]}");

// Father Marco, mother Lucia, child Sophia
// Father Marco, mother Lucia, child Alfonso
// Father Pierro, mother Francesca, child Angela
// Father Pierro, mother Francesca, child Marco
// Father Roberto, mother Maria, child Lucia
// Father Roberto, mother Maria, child Emilio
// Father James, mother Victoria, child Charlotte
// Father James, mother Victoria, child Colin
// Father Andrew, mother Christine, child Jennifer
// Father Andrew, mother Christine, child James
// Father Christopher, mother Penelope, child Victoria
// Father Christopher, mother Penelope, child Arthur
}

```

4.2. Constrângeri

Limbajul *CLIPS*, al cărui motor de inferență este încapsulat în biblioteca *FunCS*, permite și specificarea unor constrângeri asupra valorilor variabilelor din modele. Constrângerile sunt implementate ca funcții predicative în format prefix, operatorul fiind plasat în fața operanzilor.

Dintre aceste funcții, menționăm: and, or, not, eq (egal), neq (diferit) pentru valori generale și = (egal), <> (diferit), <, <=, >, >= pentru valori numerice.

Pentru exemplul anterior, să determinăm acum ce persoane sunt bunic și nepot:

```

public static void GrandParents()
{
    // Încărcarea faptelor din fișier
    ...
    var em = new ExpertMatchF(lines.ToList());
    var patterns = new List<string>
    {
        "father ?g ?p",
        "?r ?p ?c" // ?r este o relație asupra căreia se va aplica o constrângere: să fie de tip father sau mother
    };

    var constraint = "(or (eq ?r father) (eq ?r mother))";

    em.Match(patterns, constraint, out var matches);
    foreach (var m in matches)
        WriteLine($"Grandfather {m["?g"]}, {m["?r"]} {m["?p"]}, child {m["?c"]}");

    // Grandfather Roberto, mother Lucia, child Sophia
    // Grandfather Roberto, mother Lucia, child Alfonso
    // Grandfather Christopher, mother Victoria, child Charlotte
    // Grandfather Christopher, mother Victoria, child Colin
    // Grandfather Pierro, father Marco, child Sophia
    // Grandfather Pierro, father Marco, child Alfonso
    // Grandfather Andrew, father James, child Charlotte
    // Grandfather Andrew, father James, child Colin
}

```

Tot un mecanism de constrângeri poate fi folosit împreună cu soluțiile multiple pentru a determina permutările elementelor unei liste:

```
public static void Permutations()
{
    var facts = new List<string> { "1", "2", "3" };
    var em = new ExpertMatchF(facts);
    var patterns = new List<string> { "?o1", "?o2", "?o3" };
    var constraint = "(and (neq ?o1 ?o2) (neq ?o1 ?o3) (neq ?o2 ?o3))";
    em.Match(patterns, constraint, out var matches);
    foreach (var m in matches)
        WriteLine($"{m["?o1"]} - {m["?o2"]} - {m["?o3"]}");

    // 3 - 2 - 1
    // 3 - 1 - 2
    // 1 - 3 - 2
    // 2 - 3 - 1
    // 1 - 2 - 3
    // 2 - 1 - 3
}
```

Un alt exemplu în care avem soluții multiple și constrângeri este problema damelor, care constă în așezarea pe tabla de șah a unui număr de dame, fără ca acestea să intre în conflict. Nu vom putea avea deci două dame pe aceeași linie, coloană sau diagonală. Constrângerile generale sunt:

$$\begin{cases} \text{linie}_1 \neq \text{linie}_2 \\ \text{coloana}_1 \neq \text{coloana}_2 \\ |\text{linie}_1 - \text{linie}_2| \neq |\text{coloana}_1 - \text{coloana}_2| \end{cases}$$

Constrângerile pot include și funcții matematice elementare. Aici, pentru determinarea valorii absolute vom utiliza funcția `abs`. De exemplu, în *CLIPS*, `(abs (- 3 7))` returnează 4.

```
public static void MatchQueens4()
{
    var em = new ExpertMatchF("n 1 2 3 4");

    var patterns = new List<string>
    {
        "n $? ?row $?",
        "n $? ?col $?"
    };

    var queens = new List<string>();
    em.Match(patterns, out var matchList);
    foreach (var m in matchList)
        queens.Add($"queen {m["?row"]} {m["?col"]}");

    em = new ExpertMatchF(queens);

    patterns = new List<string>
    {
        "queen 1 ?c1",
        "queen 2 ?c2",
        "queen 3 ?c3",
        "queen 4 ?c4"
    };
}
```

```

"queen 2 ?c2",
"queen 3 ?c3",
"queen 4 ?c4"
};

var constraints = "(and " +
  "(neq (abs (- ?c1 ?c2)) 1)" +
  "(neq (abs (- ?c1 ?c3)) 2)" +
  "(neq (abs (- ?c1 ?c4)) 3)" +
  "(neq (abs (- ?c2 ?c3)) 1)" +
  "(neq (abs (- ?c2 ?c4)) 2)" +
  "(neq (abs (- ?c3 ?c4)) 1)" +
  "(neq ?c1 ?c2)" +
  "(neq ?c1 ?c3)" +
  "(neq ?c1 ?c4)" +
  "(neq ?c2 ?c3)" +
  "(neq ?c2 ?c4)" +
  "(neq ?c3 ?c4)" +
")";

em.Match(patterns, constraints, out matchList);
foreach (var m in matchList)
  WriteLine($"{{m["?c1"]}} {{m["?c2"]}} {{m["?c3"]}} {{m["?c4"]}}");
  // 2 4 1 3
  // 3 1 4 2
}

```

După cum se poate vedea urmărind rezultatele programului de mai sus, presupunând că prima damă se află pe linia 1, a doua pe linia 2 și.a.m.d., există două soluții:

```

Dama 1: linia 1 coloana 2
Dama 2: linia 2 coloana 4
Dama 3: linia 3 coloana 1
Dama 4: linia 4 coloana 3

Dama 1: linia 1 coloana 3
Dama 2: linia 2 coloana 1
Dama 3: linia 3 coloana 4
Dama 4: linia 4 coloana 2

```

5. Sisteme expert

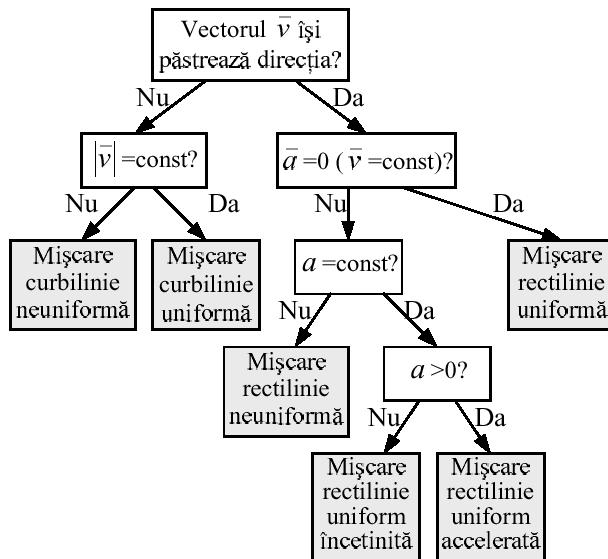
Sistemele expert sunt programe concepute pentru a raționa în scopul rezolvării problemelor pentru care în mod obișnuit se cere o expertiză umană considerabilă. Un sistem expert încorporează o bază de cunoștințe sau fapte și un motor de inferență utilizat pentru soluționarea problemelor.

Cunoașterea într-un sistem expert este organizată într-o manieră care separă cunoștințele despre domeniul problemei de alte tipuri de cunoștințe, precum cele despre algoritmii de căutare pentru rezolvarea problemelor specifice și cele despre interacțiunea cu utilizatorul.

Sistemele expert înmagazinează cunoștințe specializate, introduse de experți. De cele mai multe ori, baza de cunoștințe este foarte mare, de aceea este foarte importantă modalitatea de

reprezentare a cunoașterii. Baza de cunoștințe a sistemului trebuie separată de program, care la rândul său trebuie să fie cât mai stabil.

În cele ce urmează, vom prezenta un exemplu de sistem expert bazat pe arbori de decizie, care realizează clasificarea mișcărilor unui punct material. Sunt luate în considerare doar cele mai importante caracteristici ale mișcării: vectorul viteză și vectorul acceleratie. În funcție de valorile acestora se realizează o clasificare primară a mișcării. Deși arborele de decizie reprezentat mai jos nu este foarte mare, odată cu introducerea altor parametri, dimensiunile acestuia pot crește foarte mult.



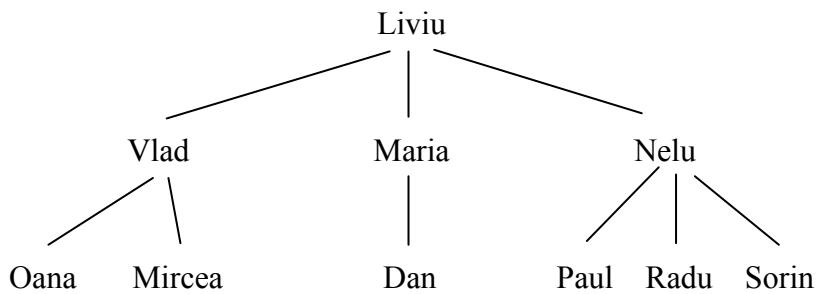
6. Aplicații

6.1. Realizați un program care afișează numerele de la 1 la 20. Pentru multiplii de 3, afișați *Fizz* în loc de număr, pentru multiplii de 5, afișați *Buzz*, iar pentru multiplii atât de 3 cât și de 5, afișați *FizzBuzz*. Folosiți metoda bazată pe instrucțiuni *switch* cu *when*.

Exemplu:

1	Fizz	11	16
2	7	Fizz	17
Fizz	8	13	Fizz
4	Fizz	14	19
Buzz	Buzz	FizzBuzz	Buzz

6.2. Fie următorul arbore genealogic. Să se afișeze verii unei anumite persoane date.



Indicații: Urmărind arborele, putem vedea că verii lui x au același bunic ca și x , dar părinți diferiți. Se dă următoarea funcție, care trebuie completată:

```

public static void Problem2()
{
    var facts = new List<string>
    {
        "parent Liviu children Vlad Maria Nelu",
        "parent Vlad children [de completat]",
        "[de completat]"
    };

    var em = new ExpertMatchF(facts);
    string searchFor = "Sorin"; // se caută verii lui Sorin

    var patterns = new List<string>
    {
        [de completat]
    };

    string constraint = [de completat];

    Write($"Verii lui {searchFor} sunt: ");
    em.Match(patterns, constraint, out var matches);
    foreach (var m in matches)
        Write($"{m["?c"]}");
    WriteLine();
}
  
```

Exemplu:

Verii lui Sorin sunt: Oana Mircea Dan

6.3. Realizați un sistem expert bazat pe un arbore de decizie generic, reprezentând raționamentul pentru rezolvarea unei anumite probleme. Problema va fi rezolvată interactiv, prin întrebări adresate utilizatorului. Se pleacă din rădăcina arborelui și apoi, în funcție de răspunsurile date, se coboară în arbore până se atinge o frunză, corespunzătoare unei concluzii (soluții).

Pentru fiecare nod neterminal, se definesc următoarele caracteristici:

- O întrebare de al cărei răspuns depinde alegerea următorului nod;
- O listă de răspunsuri acceptabile;
- O mulțime de reguli care determină nodul următor, în funcție de un anumit răspuns.

În acest scop, se utilizează fapte de tipul:

```
regulă <nod_curent> <răspuns> <nod_urmaș>
întrebare <nod_curent> <text_întrebare>
răspunsuri <nod_curent> <listă_răspunsuri_acceptabile>
concluzie <nod_terminal> <text_concluzie>
```

Se dă următoarea bază de cunoștințe, inclusă în fișierul *viteza.kbf*, care definește un arbore de decizie pentru a determina tipul de mișcare al unui punct material:

Fișierul viteza.kbf

```
root n11
question n11 Viteza isi pastreaza directia?
answers n11 da nu nu_stiu
rule when n11 if nu then n21
rule when n11 if da then n22
rule when n11 if nu_stiu then n23
conclusion n23 Informatii insuficiente.
question n21 Modulul vectorului viteza este constant?
answers n21 da nu
rule when n21 if nu then n31
rule when n21 if da then n32
conclusion n31 Miscare curbilinie neuniforma.
conclusion n32 Miscare curbilinie uniforma.
question n22 Vectorul acceleratie este nul?
answers n22 da nu
rule when n22 if nu then n33
rule when n22 if da then n34
conclusion n34 Miscare rectilinie uniforma.
question n33 Acceleratie constanta?
answers n33 da nu
rule when n33 if nu then n41
rule when n33 if da then n42
conclusion n41 Miscare rectilinie neuniforma.
question n42 Acceleratie pozitiva?
answers n42 da nu
rule when n42 if nu then n51
rule when n42 if da then n52
conclusion n51 Miscare rectilinie uniform incetinita.
conclusion n52 Miscare rectilinie uniform accelerata.
```

De exemplu, pentru nodul rădăcină (notat n11) vom avea următoarele fapte:

```
question n11 Viteza isi pastreaza directia?
answers n11 da nu nu_stiu
rule when n11 if nu then n21 //răspuns = if, nod următor = then
rule when n11 if da then n22
rule when n11 if nu_stiu then n23
```

Trebuie subliniat faptul că programul este independent de arborele de decizie pe care lucrează. Același program poate fi aplicat pentru alte probleme fără modificări suplimentare.

Exemplu:

```
Viteza isi pastreaza directia? (da/nu/nu_stiu): da
Vectorul acceleratie este nul? (da/nu): nu
Acceleratie constanta? (da/nu): da
Acceleratie pozitiva? (da/nu): nu
Miscare rectilinie uniform incetinita.
```

Indicații: Se dau următoarele funcții, care trebuie completeate:

```
public static void Problem3 ()
{
    var facts = ReadFromFile("viteza.kbf");
    var em = new ExpertMatchF(facts);

    // root n11
    em.MatchVar("root ?r", out string currentNode);

    while (true) // până când se ajunge la o concluzie
    {
        // conclusion n51 Miscare rectilinie uniform incetinita.
        em.MatchVar($"conclusion {currentNode} ${c}", out string conclusion);

        if (conclusion != "")
        {
            WriteLine(conclusion);
            break;
        }

        // se găsește întrebarea din nodul curent
        // se găsesc răspunsurile permise din nodul curent
        // se afișează întrebarea, se citește și se validează răspunsul utilizatorului
        // se găsește regula corespunzătoare răspunsului dat și se trece la următorul nod în arbore,
        // adică se actualizează variabila currentNode
    }
}

private static List<string> ReadFromFile(string fileName)
{
    var sr = new StreamReader(fileName);
    var lines = sr.ReadToEnd().Split("\r\n".ToCharArray(), StringSplitOptions.RemoveEmptyEntries);
    sr.Close();
    return lines.ToList();
}
```

Algoritmul A*

1. Obiective

Obiectivul acestui laborator este înțelegerea și implementarea algoritmului de căutare a căilor A*.

2. Algoritmul A*

O *euristică* este o metodă care furnizează rapid o soluție, nu neapărat optimă. Este o metodă aproximativă, spre deosebire de un algoritm exact optim. Deși nu garantează găsirea soluției optime, metodele euristice găsesc de obicei o soluție acceptabilă și deseori chiar soluția optimă.

Metodele euristice de căutare utilizează cunoștințe specifice fiecărei probleme pentru a ordona nodurile, astfel încât cele mai „promițătoare” noduri sunt plasate la începutul frontierei.

Funcții utilizate în general:

- $f(n)$ este un cost estimat. Cu cât este mai mic $f(n)$, cu atât este mai bun nodul n ;
- $g(n)$ este costul căii de la nodul inițial la n . Este cunoscută;
- $h(n)$ este estimarea costului căii de la n la un nod scop. Este o estimare euristică.

Tipuri de căutare:

- Căutarea de cost uniform (neinformată): $f(n) = g(n)$;
- Căutarea greedy: $f(n) = h(n)$;
- Căutarea A*: $f(n) = g(n) + h(n)$. Reunește ideile căutărilor de cost uniform și greedy.

Pentru a garanta găsirea soluției optime, funcția h trebuie să fie *admisibilă*: niciodată mai mare decât costul real. Pentru metodele euristice de căutare, problema principală este găsirea celei mai bune funcții h , care să dea estimări cât mai apropiate de realitate. Cu cât h este mai bună, numărul de noduri expandate este mai mic.

Algoritmul A* folosește două liste: *Open* (frontiera) și *Closed* (nodurile deja vizitate). Într-o iterație, se preia primul nod din lista *Open* (cu cel mai mic f). Dacă este un scop, căutarea se termină cu succes. Altfel, nodul se expandează. Pentru fiecare succesor, se calculează $f = g + h$. Succesorul se ignoră dacă în listele *Open* și *Closed* există un nod cu aceeași stare, dar cu g mai mic. Altfel, se introduce succesorul în lista *Open*. Nodul curent (care a fost expandat) se introduce în lista *Closed*, iar dacă acolo există deja un nod cu aceeași stare, acesta este înlocuit. Dacă lista *Open* devine vidă, nu există soluție.

O funcție euristică h este *monotonă* sau *consistentă* dacă respectă inegalitatea în triunghi. O euristică monotonă devine din ce în ce mai precisă cu cât înaintează în adâncimea arborelui de căutare. În multe cazuri (dar nu întotdeauna), monotonia euristicii accelerează găsirea soluției.

Ecuația pathmax face ca valorile lui f să fie monoton nedescrescătoare pe căile traversate din arborele de căutare: la generarea unui nod fiu c al lui p : $f(c) = \max(f(p), g(c) + h(c))$.

Algoritmul A* este complet și optim dacă nodurile care revizitează stările nu sunt eliminate. A* este optim eficient: niciun alt algoritm de același tip nu garantează expandarea unui număr mai mic de noduri, dar doar dacă euristică este monotonă.

Pseudocodul algoritmului A* este prezentat mai jos:

```

initialize the OPEN list
initialize the CLOSED list
push the starting node (the root) on the OPEN list (its f can be 0)
while the OPEN list is not empty
    find the node with the least f on the OPEN list, call it current
    pop current off the OPEN list
    if current is the goal, stop the search and return solution
    generate current's successors and set their parents to current
    for each successor
        successor.g = current.g + distance between successor and current
        successor.h = estimated distance from successor to goal
        successor.f = successor.g + successor.h
        if a node with the same state as successor is in the OPEN list which has a lower g than successor, skip successor
        if a node with the same state as successor is in the CLOSED list which has a lower g than successor, skip successor
        otherwise add successor to the OPEN list
    end
    push current on the CLOSED list (if a node with the same state exists in CLOSED list, replace it)
end

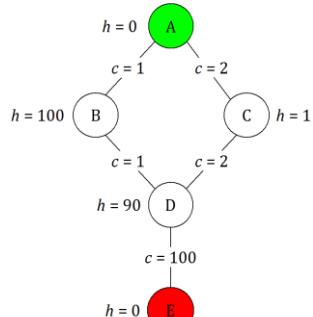
```

Pentru explicații suplimentare, consultați cursul 2.

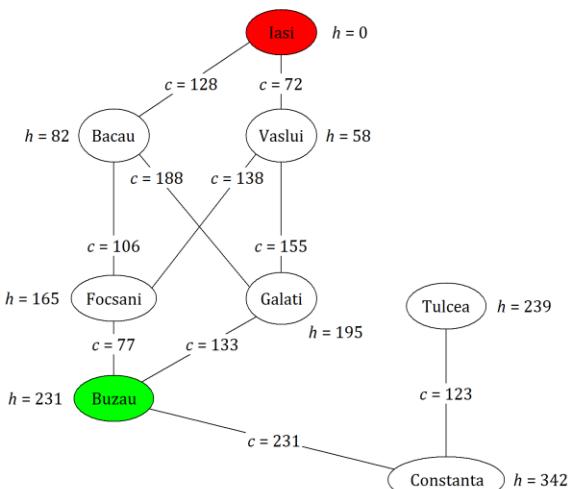
3. Aplicații

Fie următoarele probleme de căutare:

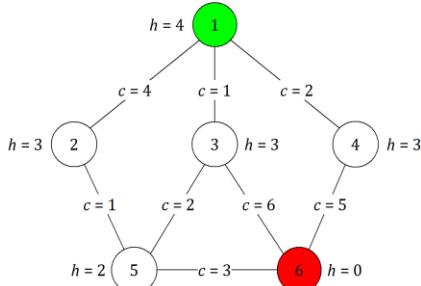
Problema 1 (Problem1.cs)



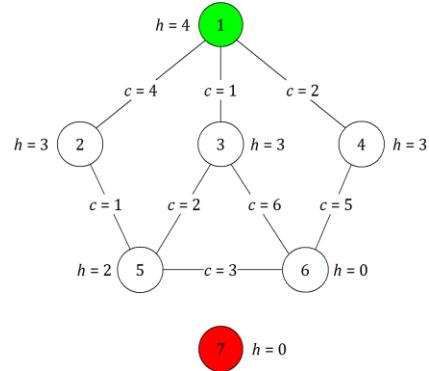
Problema 2 (Problem2.cs)



Problema 3 (Problem3.cs)

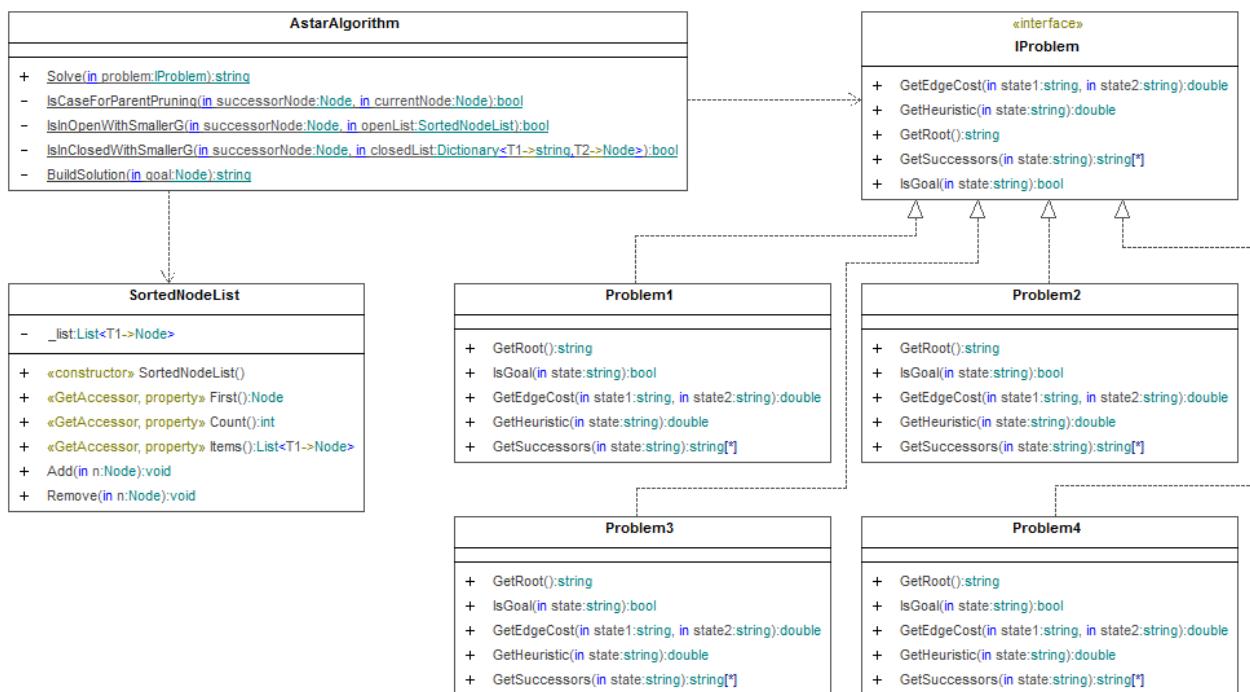


Problema 4 (Problem4.cs)



4.1. Rezolvați pe hârtie problema 2 sau problema 3, la alegere, după modelul de rezolvare a problemei 1 din cursul 2. Arătați cum evoluează lista deschisă, ce noduri sunt scoase din listă, ce noduri sunt expandate, ce valori f au fiile etc.

4.2. Implementați algoritmul A* pentru o problemă generică de căutare a căilor. Se dă un schelet de program, în care există o interfață IProblem, care trebuie respectată atunci când se definește o problemă de căutare. Soluția mai conține implementările celor 4 probleme prezentate mai sus. Diagrama UML de clase a soluției complete este următoarea:



Trebuie completată clasa AstarAlgorithm, cu metoda Solve și cele trei metode de test pentru optimizarea căutării. Se va folosi și ecuația $pathmax$.

Rezultatele programului pentru cele 4 probleme ar trebui să fie următoarele:

Problema 1

A - B - D - E

Problema 2

Buzau - Focsani - Vaslui - Iasi

Problema 3

1 - 3 - 5 - 6

Problema 4

Nu exista solutie

Jocuri: algoritmul minimax

1. Obiectiv

Obiectivul acestui laborator este implementarea unui joc simplu între om și calculator, folosind algoritmul minimax.

2. Jocuri

Un *joc* reprezintă o succesiune de decizii (acțiuni) luate de părți ale căror interese sunt opuse. Jocurile pot fi clasificate după:

- *numărul de jucători*: 1, 2, n (n nu înseamnă că participă efectiv n jucători, ci că regulile permit împărțirea jucătorilor în n mulțimi disjuncte, astfel încât jucătorii din fiecare mulțime să aibă interese comune);
- *natura mutărilor*: există jocuri cu mutări libere (mutarea este aleasă conștient, dintr-o mulțime de acțiuni posibile, de exemplu șahul) și jocuri cu mutări aleatorii (mutarea este dictată de un factor aleatoriu – zaruri, cărți de joc, monede etc.). Pot exista jocuri cu ambele tipuri de mutări;
- *cantitatea de informație* de care dispune un jucător (un jucător de șah vede configurația completă a pieselor adversarului, pe când un jucător de cărți nu are acces la configurațiile celorlalți jucători).

Există două motive pentru care jocurile sunt un bun domeniu de explorat pentru inteligența artificială:

- Au o organizare clară, în care e foarte ușor de măsurat succesul sau eșecul;
- Nu necesită cantități mari de cunoștințe inițiale. Multe jocuri sunt concepute astfel încât să poată fi rezolvate prin metode de căutare din starea inițială până în poziția câștigătoare. Bineînteles, acest lucru este adevărat numai pentru jocurile simple. Un contraexemplu este jocul de șah, care presupune un arbore de căutare cu factorul de ramificare 35, adică în fiecare moment există (în medie) 35 de mutări disponibile. Având în vedere că un jucător face (tot în medie) aproximativ 50 de mutări, putem trage concluzia că pentru un joc trebuie examineate $35^{250} = 2,55 \cdot 10^{154}$ variante.

Din cauza numărului mare de posibilități, trebuie să existe euristici de căutare. În acest scop, se folosesc proceduri de generare și test. Procedurile de generare pot fi optimizate astfel încât să se

genereze numai mutări (sau căi) bune. De asemenea, procedurile de test trebuie să recunoască mutările bune, pentru ca acestea să fie explorate primele.

Există jocuri pentru care avem și alte tehnici, în afară de căutare. De exemplu, în șah („*drosophila IA*”, după cum îl consideră Alexander Kronod), deschiderile și finalurile sunt deseori memorate sub formă de modele în baze de date. Aici pot fi combinate tehnici de căutare cu tehnici mai directe de genul celor menționate mai sus.

Când dimensiunea arborelui de căutare este prea mare, se utilizează o *funcție de evaluare statică*, care folosește informațiile disponibile la un moment dat pentru a evalua pozițiile care au probabilitate mare de a conduce la atingerea scopului – câștigarea jocului. În absența informațiilor complete, se alege euristică poziția cea mai promițătoare. De multe ori, este greu de găsit o funcție bună. De fapt, „inteligenta” cu care joacă calculatorul depinde în mare măsură de această funcție de evaluare.

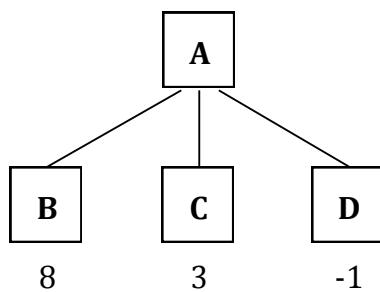
3. Algoritmul de căutare *minimax*

Minimax este un algoritm de căutare limitată în adâncime (depth-first, depth-limited). Se pleacă de la poziția curentă și se generează o mulțime de poziții următoare posibile. În acest scop, structura de date cel mai des folosită este arborele. Se aplică funcția de evaluare statică și se alege poziția cea mai bună.

Se presupune că jocul este de sumă nulă, deci se poate folosi o singură funcție de evaluare pentru ambii jucători. Dacă $f(n) > 0$, poziția n este bună pentru calculator și rea pentru om, iar dacă $f(n) < 0$, poziția n este rea pentru calculator și bună pentru om. Funcția de evaluare este stabilită de proiectantul aplicației.

Pentru aplicarea algoritmului, se selectează o limită de adâncime și o funcție de evaluare. Se construiesc arboarele până la limita de adâncime. Se calculează funcția de evaluare pentru frunze și se propagă evaluarea în sus, selectând minimele pe nivelul minimizant (decizia omului) și maximele pe nivelul maximizant (decizia calculatorului).

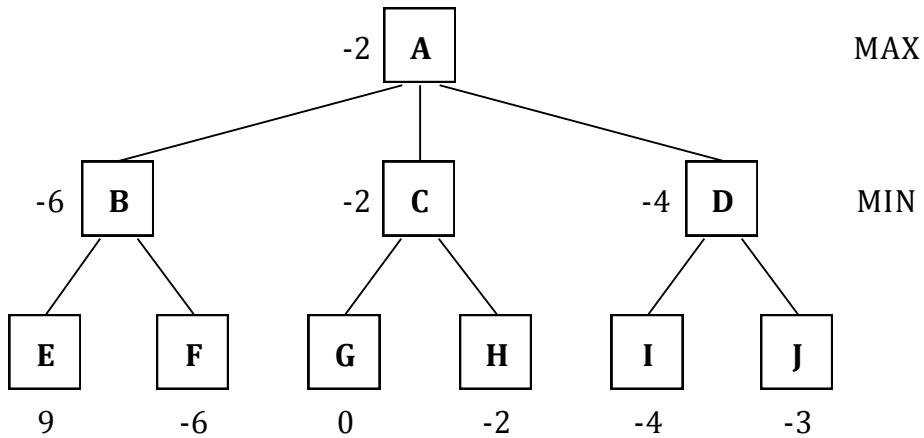
Să considerăm mai întâi căutarea pe un singur nivel (engl. ply – pliu, cută, strat; în teoria jocurilor, această denumire sugerează numărul de niveluri în arbore care se studiază):



Primul nivel este un nivel maximizant, de aceea valoarea rădăcinii arborelui este maximul dintre valorile fiilor. Aici, valoarea lui A este 8, deoarece valoarea lui B (8) este valoarea maximă.

Procedura minimax presupune căutarea alternativă pe niveluri maximizante și niveluri minimizante. Dacă vom cerceta două niveluri, primul nivel va fi maximizant, iar al doilea minimizant.

Căutarea pe două niveluri este de forma:



În nodul *B*, vom avea valoarea minimă a fiilor săi (-6). La fel în *C* și *D*. În *A*, vom prelua valoarea maximă dintre *B*, *C* și *D*, adică -2 .

Acstea valori numerice sunt date de funcția de evaluare statică. Deoarece aceasta este deseori imprecisă, e importantă căutarea pe cât mai multe niveluri, pentru a crește numărul de posibilități avute în vedere, și deci și şansele de a determina mutarea optimă.

Pseudocodul algoritmului minimax este următorul:

```

Minimax (board, depth)
  if depth = depthBound then
    return StaticEvaluation(board)
  else if IsMaximizingLevel(depth) then
    foreach child c of board
      compute Minimax(c, depth+1)
    return maximum over all children
  else if IsMinimizingLevel(depth) then
    foreach child c of board
      compute Minimax(c, depth+1)
    return minimum over all children

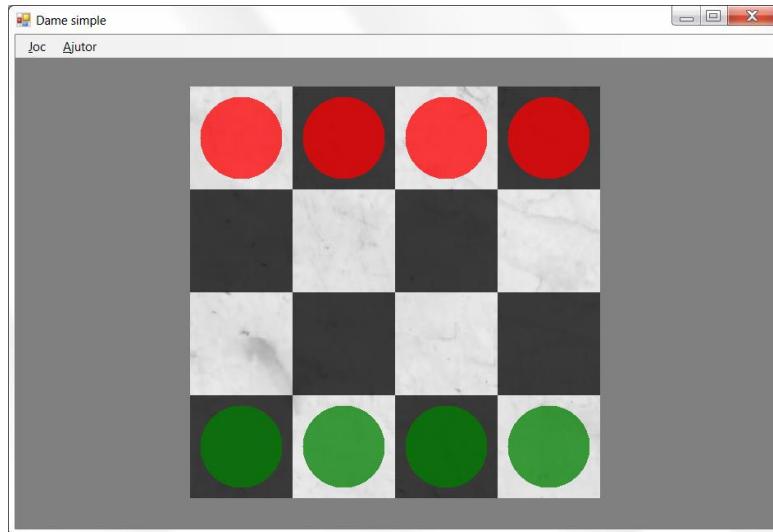
Initialize(depthBound)
Minimax(currentBoard, 0)
  
```

Pentru explicații suplimentare, consultați cursul 3.

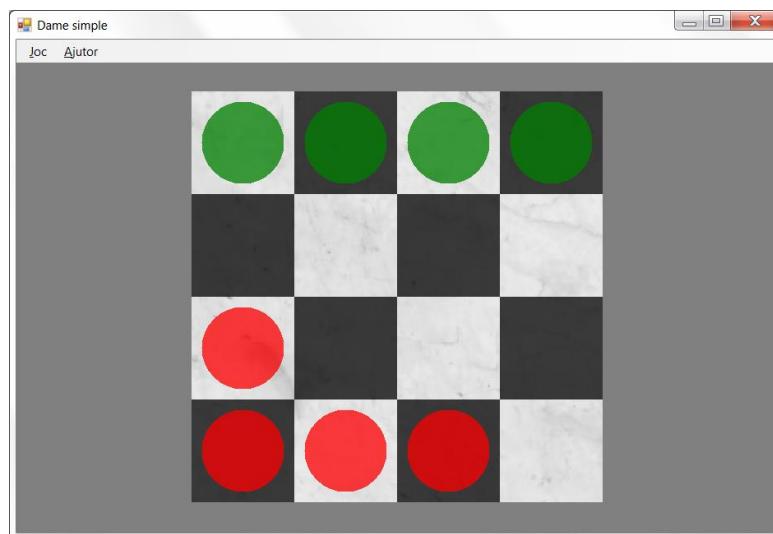
5. Aplicație

Realizați o variantă simplificată a unui joc de dame, cu următoarele reguli:

- există doi jucători;
- piesele sunt dispuse pe o tablă de 4x4 careuri;
- configurația (tablă) inițială este:

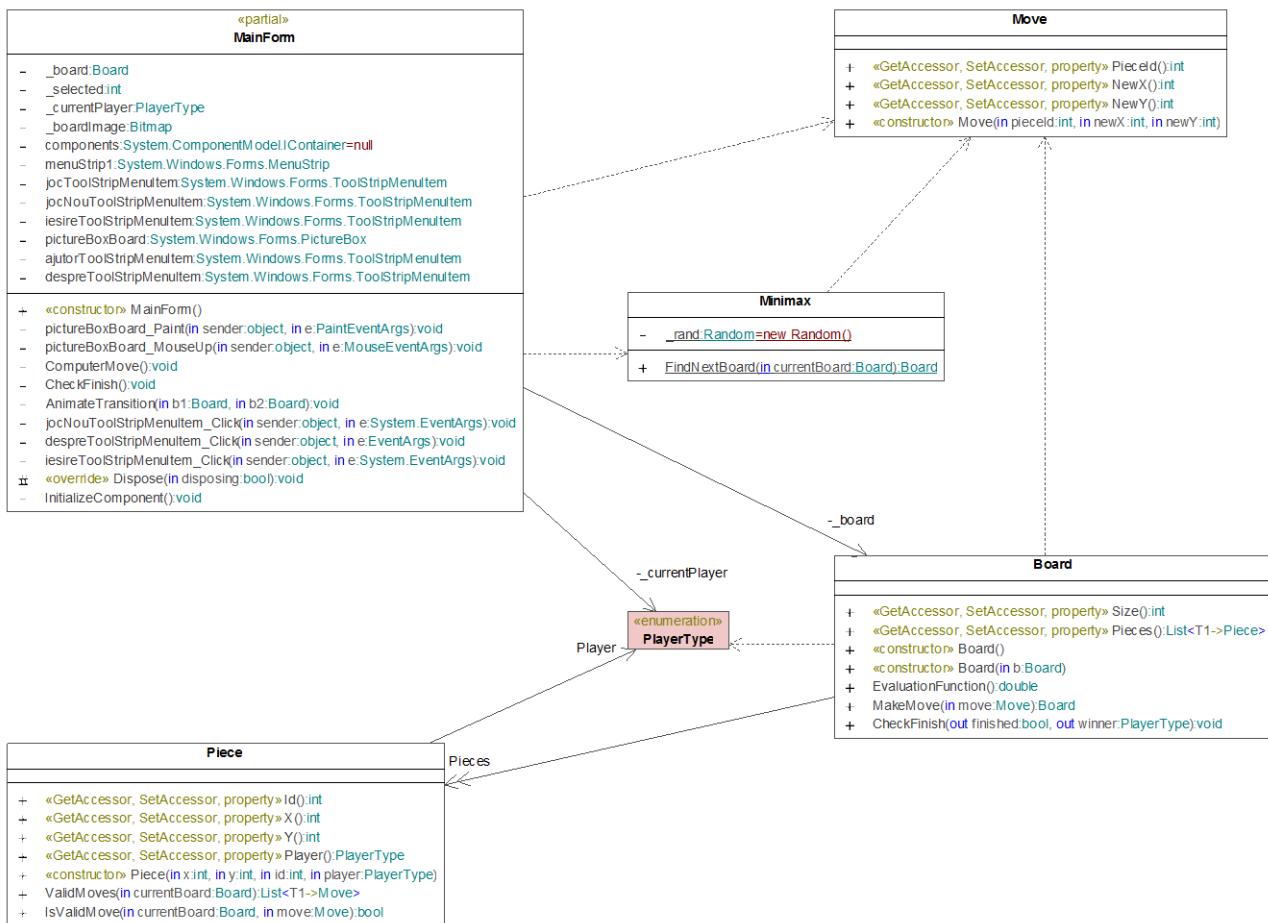


- fiecare jucător are ca scop să ajungă pe ultima linie cu toate piesele (în acest caz, a câștigat omul):



- mutări posibile: fiecare piesă poate fi mutată în orice direcție, într-un careu liber adiacent.

Indicații. Se dă un prototip de aplicație, în care sunt implementate toate clasele necesare pentru rezolvare și interfața grafică cu utilizatorul. Diagrama UML de clase a aplicației complete este următoarea:



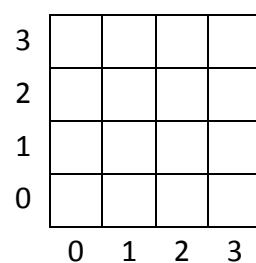
Trebuie implementate doar următoarele metode:

Metoda EvaluationFunction() din clasa Board

Calculează funcția de evaluare statică pentru configurația (tabla) curentă. Pentru un joc oarecare, programatorul trebuie să imagineze funcția de evaluare. Pot exista mai multe funcții posibile pentru un joc. Cu cât funcția aceasta este mai „inteligentă”, cu atât calculatorul va juca mai bine.

De exemplu, în cazul nostru, o funcție de evaluare poate lua în calcul diferența dintre cât au avansat piesele calculatorului și cât au avansat piesele omului. Cu cât funcția este mai mare, cu atât poziția este mai bună pentru calculator, adică aceasta este mai aproape de configurația câștigătoare.

În program, coordonatele x și y ale pieselor sunt considerate ca mai jos:



Să presupunem că alegem funcția:

$$F = \sum_c dy_c - \sum_o dy_o,$$

unde dy_c reprezintă avansul pieselor calculatorului, iar dy_o , ale omului.

Funcția va fi 0 când niciun jucător nu are vreun avantaj, de exemplu:

3				
2	C	C	O	O
1	O	O	C	C
0				
	0	1	2	3

Aici, primele două piese ale calculatorului au avansat o căsuță pe axa y , de la 3 la 2, iar ultimele două au avansat două căsuțe, de la 3 la 1. Analog pentru om, de la 0 la 1, respectiv de la 0 la 2:

$$F = (1+1+2+2) - (1+1+2+2) = 0.$$

Funcția va fi mare când calculatorul are un avantaj, de exemplu:

3				
2				
1	O	O	O	O
0	C	C	C	C
	0	1	2	3

$$F = (3+3+3+3) - (1+1+1+1) = 8.$$

Funcția va fi mică când omul are un avantaj, de exemplu:

3	O	O	O	O
2	C	C	C	C
1				
0				
	0	1	2	3

$$F = (1+1+1+1) - (3+3+3+3) = -8.$$

Se poate verifica faptul că această funcție poate fi scrisă mai compact, luând în calcul direct coordonatele y ale pieselor și nu cât au avansat acestea:

$$F = 12 - \sum_c y_c - \sum_o y_o ,$$

unde y_c reprezintă coordonatele y ale pieselor calculatorului, iar y_o , ale omului.

Am putea alege și o funcție mai simplă, care să ia în considerare numai piesele calculatorului, însă în acest caz calculatorul ar încerca doar să avanseze și nu ar putea încerca să blocheze avansarea omului. De asemenea, se pot imagina și funcții mai complexe, care să ia în calcul pozițiile libere de pe linia scop, pozițiile relative ale pieselor adversarului etc.

Se va acorda un bonus de 0.25 p pentru utilizarea unei funcții de evaluare mai „inteligente”. În acest caz, trebuie inclus în metodă un comentariu care să explice ideea funcției.

Metoda ValidMoves(Board currentBoard) din clasa Piece

Returnează lista tuturor mutărilor permise pentru piesa curentă în configurația currentBoard.

Folosește metoda IsValidMove.

Metoda IsValidMove(Board currentBoard, Move move) din clasa Piece

Testează dacă o mutare este permisă în configurația curentă. O mutare este invalidă dacă: nu mută nicio piesă, nu mută într-un careu adjacente ci sare peste mai multe careuri, mută în afara tablei sau mută într-un careu ocupat de altă piesă. Această metodă validează atât mutările calculatorului (în metoda ValidMoves din clasa Piece) cât și mutările omului (în evenimentul pictureBoxBoard_MouseUp din clasa MainForm).

Metoda FindNextBoard(Board currentBoard) din clasa Minimax

Aplică algoritmul minimax pe un singur nivel. Primul nivel corespunde mutării calculatorului, deci este maximizant. Transformă mutările valide ale tuturor pieselor din currentBoard într-o listă de configurații, folosind metoda MakeMove din clasa Board. Apoi selectează din această listă configurația cu funcția de evaluare statică maximă. Dacă există mai multe configurații cu funcția de evaluare maximă, se alege aleatoriu una dintre ele.

În cazul general, descris în laborator, algoritmul ar trebui să lucreze pe un arbore.

Se va acorda un bonus de 0.5 p pentru realizarea căutării minimax în mod recursiv, pe cel puțin 2 niveluri în arborele jocului.

Algoritmi evolutivi

1. Obiective

Obiectivele acestui laborator sunt următoarele:

- prezentarea structurii generale a unui algoritm evolutiv, împreună cu operatorii specifici;
- rezolvarea a două probleme de optimizare bazate pe valori reale, folosind același algoritm.

2. Structura unui algoritm evolutiv

Probleme precum alegerea traseelor optime ale unor vehicule, rutarea mesajelor într-o rețea, planificarea unor activități, problema orarului etc., pot fi formulate ca probleme de optimizare. Multe dintre acestea sunt probleme NP-dificele, necunoscându-se algoritmi de rezolvare care să aibă complexitate polinomială. Pentru astfel de probleme, algoritmii evolutivi oferă posibilitatea obținerii în timp rezonabil a unor soluții sub-optimale de calitate acceptabilă.

Algoritmii evolutivi sunt aplicații cu succes în proiectarea circuitelor digitale, a filtrilor dar și a unor structuri de calcul precum rețelele neuronale. Ca metode de estimare a parametrilor unor sisteme care optimizează anumite criterii, se aplică în diverse domenii din inginerie cum ar fi: proiectarea avioanelor, proiectarea reactoarelor chimice, proiectarea structurilor în construcții etc. Uneori nici nu se cunoaște o expresie analitică a funcțiilor care trebuie optimizate, ci acestea se estimează din simulări. În aceste cazuri, nu pot fi aplicate ușor metodele de optimizare diferențială.

Un algoritm evolutiv este o metodă de optimizare prin analogie cu evoluția biologică. Pentru găsirea soluției, se utilizează o populație de soluții potențiale, care evoluează prin aplicarea iterativă a unor operatori stohastici. Elementele populației reprezintă soluții potențiale ale problemei.

Pentru a ghida căutarea către soluția problemei, asupra populației se aplică transformări specifice evoluției naturale:

- *Selecția*. Determină părinții care se vor reproduce pentru a forma următoarea generație. Individii mai adaptați din populație (care se apropie mai mult de soluția problemei) sunt favorizați, în sensul că au mai multe șanse de reproducere;
- *Încrucișarea*. Pornind de la doi părinți, se generează copii;
- *Mutarea*. Pentru a asigura diversitatea populației, se aplică transformări cu caracter aleatoriu asupra copiilor nou generați, permitând apariția unor trăsături care nu ar fi apărut în cadrul populației doar prin selecție și încrucișare.

Structura generală a unui astfel de algoritm este cea din figura 1.

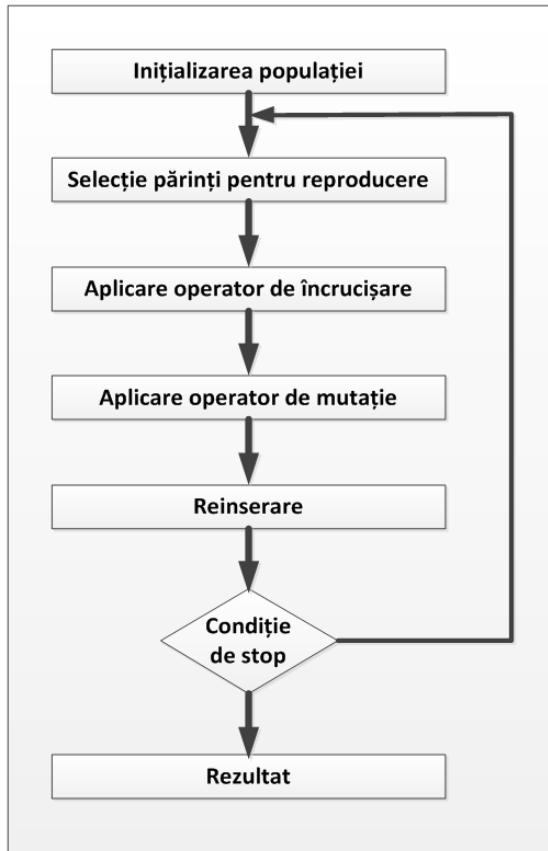


Figura 1. Structura generală a unui algoritm evolutiv

Codificarea diferă de la problemă la problemă, cele mai des folosite metode fiind codificările binară și reală. Condiția de stop se poate referi la evoluția pentru un anumit număr de generații sau la proprietățile populației curente, de exemplu convergența populației către o anumită valoare, pierderea diversității indivizilor din populație etc.

Dimensiunea populației poate fi în jur de 50. Pentru probleme simple poate fi mai mic (30), iar pentru probleme dificile poate fi mai mare (100). Dacă sunt prea puțini cromozomi, algoritmul nu are diversitatea necesară găsirii soluției. Dacă sunt prea mulți, algoritmul va fi mai lent, fără a se îmbunătăți însă calitatea soluției.

Numărul maxim de generații variază de obicei între 100-1000.

3. Tipuri de codificare

Modul în care o posibilă soluție este codificată într-un cromozom depinde de problema concretă care trebuie rezolvată.

Codificarea binară

Este varianta clasică a algoritmilor genetici. În acest caz, cromozomii sunt siruri de n biți, n fiind numărul de gene. Este adecvată pentru problemele de optimizare combinatorială în care configurațiile pot fi specificate ca vectori binari.

Exemplul 3.1. Problema rucsacului

Se consideră o mulțime de n obiecte caracterizate prin greutățile (w_1, \dots, w_n) și prin valorile (v_1, \dots, v_n). Se pune problema determinării unei submulțimi de obiecte pentru a fi introduse într-un rucsac de capacitate C astfel încât valoarea obiectelor selectate să fie maximă. O soluție a acestei probleme poate fi codificată ca un sir de n valori binare în felul următor: $s_i = 1$ dacă obiectul i este selectat, respectiv $s_i = 0$ dacă obiectul nu este selectat.

Pentru 5 obiecte posibile de introdus în sac, o *genă* reprezintă un bit (x_i), în timp ce cromozomul este o soluție potențială (de exemplu 01101 înseamnă că articolele 2, 3 și 5 vor fi incluse în sac).

Funcția de adaptare (engl. “fitness”) arată cât de bună este o soluție, cât este de adaptat un cromozom.

De exemplu, presupunem că la un moment dat avem combinația:

i	1	2	3	4	5
w_i	70	55	40	15	5
v_i	40	15	5	30	10

Atunci, pentru cromozomul 01101, constrângerea este satisfăcută: $w = 55 + 40 + 5 = 100 \leq C = 100$, iar funcția de adaptare este: $F = 15 + 5 + 10 = 30$.

Exemplul 3.2. Problema împachetării

Se consideră un set de n obiecte caracterizate prin dimensiunile (d_1, d_2, \dots, d_n) și o mulțime de m cutii având capacitatele (c_1, c_2, \dots, c_m). Se pune problema plasării obiectelor în cutii astfel încât capacitatea acestora să nu fie depășită, iar numărul de cutii utilizate să fie cât mai mic. O posibilă reprezentarea binară pentru această problemă este următoarea: se utilizează o matrice cu n linii și m coloane iar elementul S_{ij} are valoarea 1 dacă obiectul i este plasat în cutia j și 0 în caz contrar. Prin liniarizarea matricii se ajunge ca fiecare soluție potențială să fie reprezentată printr-un cromozom cu $m \cdot n$ gene.

Codificarea reală

Este adekvată pentru problemele de optimizare pe domenii continue. În acest caz, cromozomii sunt vectori cu elemente reale din domeniul de definiție al funcției. Avantajul acestei reprezentări este faptul că este naturală și nu necesită proceduri speciale de codificare/decodificare.

Exemplul 3.3. Funcția Rastrigin generalizată

Presupune minimizarea funcției următoare:

$$f(\mathbf{x}) = n \cdot A + \sum_{i=1}^n x_i^2 - A \cos(\omega \cdot x_i)$$

Pentru $A = 10$ și $\omega = 2\pi$, funcția are minimul $f(\mathbf{0}) = 0$.

Domeniul variabilelor este: $-5.12 < x_i < 5.12$.

În acest caz, cromozomii vor fi vectori cu n elemente din acest interval.

4. Construirea funcției de adaptare

Funcția de adaptare definește problema de optimizare. În formularea standard, scopul algoritmilor evolutivi este maximizarea acestei funcții.

Orice problemă de minimizare poate fi transformată într-o de maximizare prin schimbarea semnului funcției obiectiv. Pentru o problemă de minimizare în care dorim minimizarea unei funcții $g(x)$, funcția de adaptare va fi $F(x) = -g(x)$.

5. Selecția

Selecția are ca scop determinarea părinților care vor genera elementele populației din următoarea generație. Criteriul de selecție se bazează pe valoarea funcției de adaptare. Procesul de selecție nu depinde de modul de codificare a elementelor populației (binar, real etc.).

Selecția de tip ruletă

Ideea de bază a acestui tip de selecție este că probabilitatea unui individ de a fi selectat este proporțională cu funcția sa de adaptare. Ruleta se învârte de n ori pentru a se alege n indivizi.

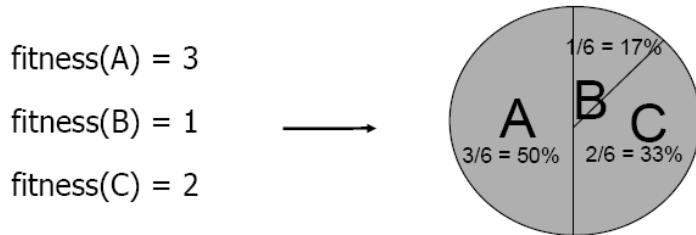


Figura 2. Selecția de tip ruletă

Selecția bazată pe ranguri

Se ordonează crescător valorile funcției de adaptare pentru toate elementele populației. Se rețin valorile distințe și li se asociază câte un rang: cea mai mică valoare are rangul 0 sau 1, iar cea mai mare are rangul maxim, n sau $n - 1$, unde n este dimensiunea populației. Probabilitatea de selecție este în acest caz proporțională cu rangul individului.

Selecția de tip competiție

Se aleg aleatoriu k membri din populație și se determină cel mai adaptat dintre aceștia. De obicei, $k = 2$. Procedura se repetă pentru a selecta mai mulți părinți. Această metodă este mai rapidă decât selecțiile prin ruletă sau ranguri.

Elitismul

Cel mai adaptat individ este copiat direct în noua populație. Asigură faptul că niciodată nu se pierde soluția cea mai bună.

6. Încrucișarea

Încrucișarea (engl. “crossover”) permite combinarea informațiilor provenite de la doi părinți pentru generarea a 1 sau 2 copii. Vom considera doar cazul a doi părinți (notați cu **m** și **f**, de la “mother” și “father”) care generează doi copii (notați cu **c**₁ și **c**₂). Încrucișarea (numită uneori și recombinare) depinde de modul de codificare al datelor, aplicându-se direct asupra cromozomilor.

Încrucișarea binară

Cea mai des folosită este *încrucișarea cu un punct*. Reamintim că aici cromozomul este un sir de n biți. Se alege în mod aleatoriu un punct de tăietură $k \in \{1, \dots, n-1\}$ și se construiesc copiii astfel:

$$\begin{aligned}\mathbf{c}_1 &= (m_1, \dots, m_k, f_{k+1}, \dots, f_n) \\ \mathbf{c}_2 &= (f_1, \dots, f_k, m_{k+1}, \dots, m_n)\end{aligned}$$

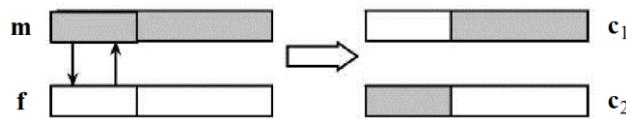


Figura 3. Încrucișarea binară cu un punct

Încrucișarea reală

Încrucișarea aritmetică combină materialul genetic al părinților **m** și **f** (vectori de numere reale) aplicând următoarele reguli pentru a obține, de exemplu, un copil **c**:

$$c_i = a \cdot m_i + (1 - a) \cdot f_i$$

În mod analog cu încrucișarea binară, se pot obține simultan doi copii în loc de unul singur.

Valoarea a este un număr aleatoriu din intervalul $(0, 1)$ sau $(-0.25, 1.25)$. În ultimul caz, copiii se vor găsi în interiorul unui hipercub puțin mai mare decât hipercubul delimitat de valorile genelor părinților. Această metodă este utilă în cazul în care domeniul de definiție al genelor nu poate fi estimat foarte bine și trebuie extins în mod dinamic pe parcursul execuției algoritmului.

Astfel, copilul va fi plasat pe un segment de dreaptă determinat de genele părinților.



Figura 4. Încrucișarea reală aritmetică

Încrucișarea are loc cu o anumită probabilitate, numită rată de încrucișare și notată cu p_c . Aceasta trebuie să fie în general mare, de exemplu 0.9.

Din punct de vedere al implementării, se generează un număr aleatoriu r între 0 și 1. Dacă $r < p_c$, se aplică încrucișarea. Dacă nu, se generează un copil egal cu un parinte.

7. Mutăția

Mutăția asigură modificarea valorilor unor gene pentru a evita situațiile în care o anumită alelă (valoare posibilă a unei gene) nu apare în populație deoarece nu a fost generată de la început. În felul acesta poate fi crescută diversitatea populației și se poate evita convergența într-un optim local.

Mutăția se efectuează asupra copiilor generați după aplicarea operatorului de încrucișare.

Mutăția binară

Cel mai simplu și des utilizat tip de mutație binară este cel în care se selectează o genă iar valoarea acesteia este negată (0 devine 1 iar 1 devine 0) cu o anumită probabilitate, numită rată de mutație și notată cu p_m . Aceasta trebuie să fie în general mică, de exemplu 0.02.

Mutăția reală

Pentru codificarea reală, se folosește de obicei resetarea valorii unei gene la un număr aleatoriu din domeniul ei de definiție. Rata de mutație reală poate fi mai mare decât la mutația binară, deoarece numărul valorilor reale este mai mic decât numărul de biți, de exemplu 0.1.

Pentru explicații suplimentare, consultați cursul 4.

8. Aplicații

8.1. Determinați soluția reală a ecuației: $x^5 - 5x + 5 = 0$. (Soluția reală este: $x = -1.680494$.)

Se va acorda un bonus de 0.25 p pentru determinarea în plus și a soluțiilor complexe.

În acest scop, trebuie modificată corespunzător funcția de adaptare. (Soluțiile complexe sunt: $1.05 \pm 0.3i$ și $-0.2 \pm 1.57i$.)

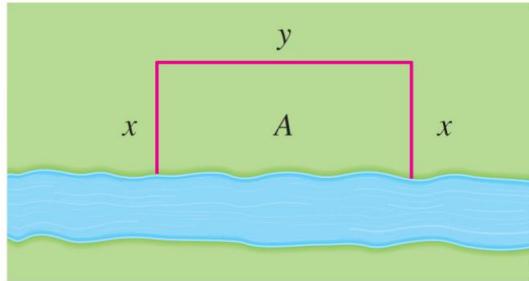
Indicații. Trebuie să exprimăm cerința sub formă unei probleme de optimizare, adică de găsire a maximului (sau minimului) unei funcții. Dacă trebuie să determinăm x astfel încât $f(x) = y$, înseamnă că trebuie să minimizăm diferența dintre $f(x)$ și y : $\min |f(x) - y|$. Minimul așteptat ar trebui să fie 0. Pentru ecuația noastră, va trebui să determinăm: $\min |x^5 - 5x + 5|$. Deoarece algoritmul evolutiv standard găsește maximul unei funcții, nu minimul ei, trebuie să transformăm problema într-o maximizare: $\max -|x^5 - 5x + 5|$. Prin urmare, funcția de adaptare corespunzătoare problemei este: $F(x) = -|x^5 - 5x + 5|$.

Pentru că toți operatorii genetici presupun lucrul cu numere aleatorii, reamintim două metode utile din clasa Random. Fie obiectul Random rand = new Random(). Atunci:

- rand.Next(max) returnează un număr natural din mulțimea { 0, ..., max - 1 };
- rand.NextDouble() returnează un număr real din intervalul [0, 1).

8.2. Un fermier are 100 m de gard cu care vrea să împrejmuiască un teren dreptunghiular care se învecinează cu un râu. Latura vecină cu râul nu are nevoie de gard. Care sunt dimensiunile terenului cu arie maximă care poate fi împrejmuit în acest fel?

Problema de optimizare formalizată este maximizarea ariei $A = x \cdot y$, respectând constrângerea $2x + y = 100$. (Soluția este: $x = 25$ și $y = 50$, deci $A = 1250$).



Indicații. Aici funcția de adaptare este direct cea exprimată de problemă: $F(x, y) = x \cdot y$. Dificultatea apare din existența constrângerii. Întrucât foarte puțini indivizi ar putea respecta constrângerea, o soluție intelligentă este „repararea” acestora, adică forțarea respectării constrângerii. De exemplu, un individ are genele x și y , care nu respectă constrângerea. Genele sale se pot scala proporțional cu un factor r , păstrând informațiile utile pentru căutare și diversitate genetică:

$$\begin{aligned} r &= 100 / (2x + y) \\ x' &= x \cdot r \\ y' &= y \cdot r \end{aligned}$$

Astfel, $2x' + y'$ va fi întotdeauna 100 (constrângerea va fi satisfăcută de toți indivizii), iar optimizarea se va face după funcția de adaptare $F(x', y') = x' \cdot y'$.

Repararea genelor presupune faptul ca în metoda corespunzătoare funcției de adaptare să se modifice efectiv genele unui cromozom din vechile valori x și y în noile valori scalate x' și y' .

Se dă un prototip de aplicație, în care sunt implementate toate clasele necesare pentru rezolvare. Folosind variantele *reale* ale operatorilor genetici, implementați următoarele metode:

Metoda Tournament(Chromosome[] population) din clasa Selection

Alege în mod aleatoriu doi indivizi/cromozomi diferenți din populație, le compară valorile funcției de adaptare și îl returnează pe cel mai adaptat.

Metoda GetBest(Chromosome[] population) din clasa Selection

Returnează individul din populație cu funcția de adaptare maximă. Nu returnă referința la cel mai bun individ din populație, ci creați un obiect nou, copie a celui mai bun individ.

Metoda Arithmetic(Chromosome m, Chromosome f, double rate) din clasa Crossover

Aplică încrucișarea aritmetică între cei doi părinți, cu probabilitatea rate.

Metoda Reset(Chromosome child, double rate) din clasa Mutation

Aplică asupra copilului child mutația prin resetare, cu probabilitatea rate.

Metoda Solve(...) din clasa EvolutionaryAlgorithm

Trebuie completată bucla principală, cu apelul operatorilor genetici.

Metoda ComputeFitness(Chromosome c) din clasa Equation

Funcția de adaptare pentru prima problemă.

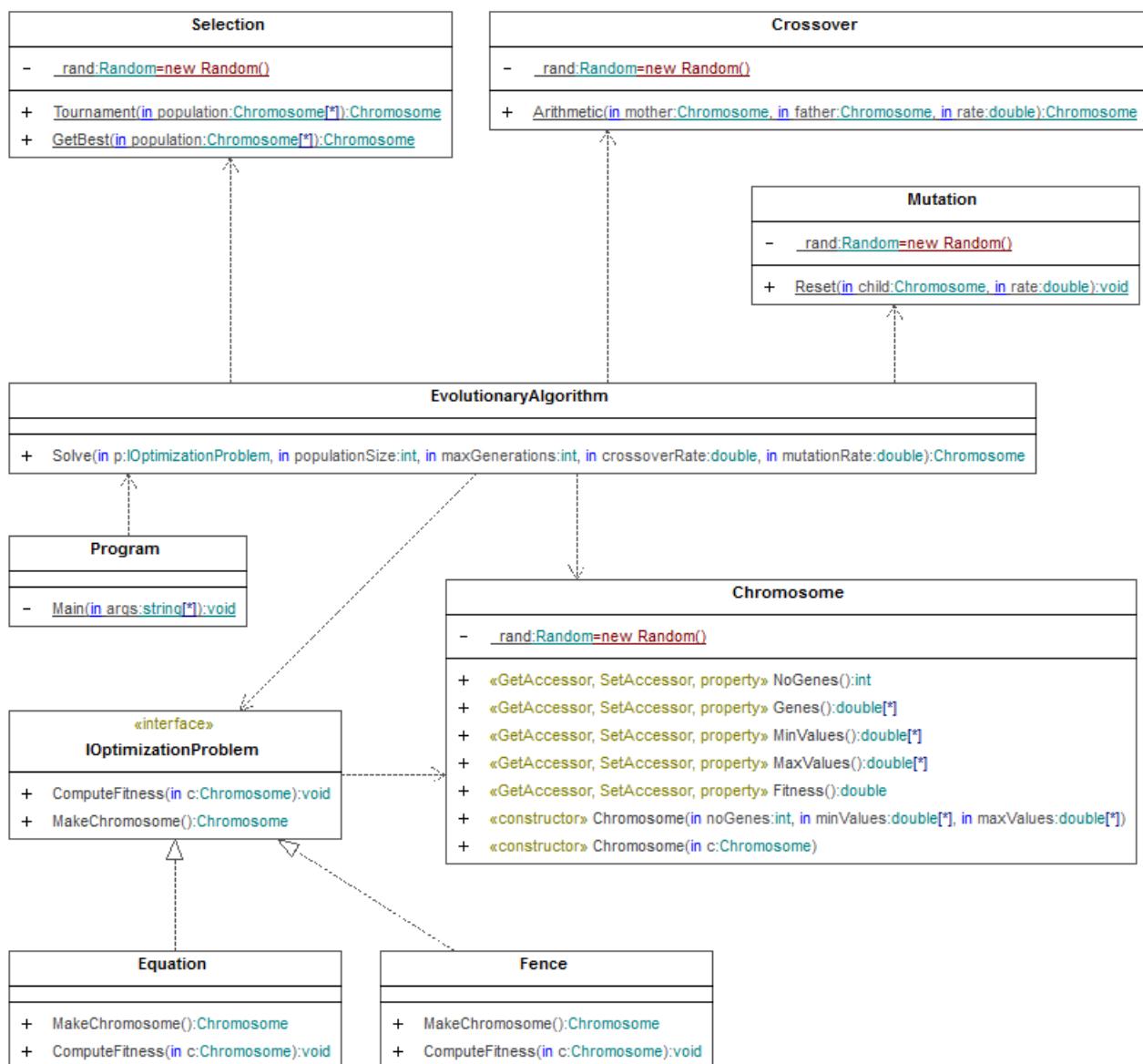
Metoda ComputeFitness(Chromosome c) din clasa Fence

Funcția de adaptare pentru a doua problemă. Aici trebuie satisfăcută și constrângerea. Fezabilitatea soluțiilor se poate asigura prin repararea cromozomilor.

Metoda Main() din clasa Program

Pentru ambele probleme, trebuie să găsiți, prin încercări repetitive, valorile parametrilor algoritmului (dimensiunea populației, numărul maxim de generații, ratele de încrucișare și mutație) care dă cele mai bune rezultate.

Diagrama UML de clase a aplicației complete este următoarea:



Rezoluția propozițională

1. Obiectiv

Obiectivul acestui laborator este prezentarea algoritmului de rezoluție propozițională, o metodă generală de demonstrare automată a teoremelor în logica propozițională.

2. Forma normal conjunctivă

Vom aminti mai întâi câteva noțiuni fundamentale ale logicii propoziționale. Un *literal* este o propoziție atomică sau negația unei propoziții atomice, de exemplu p și $\neg p$. O *clauză* este un literal sau o disjuncție de literali, de exemplu: p , $\neg p$, $\neg p \vee q$ etc.

O clauză este *realizabilă* (engl. “satisfiable”) dacă există o interpretare, adică o atribuire de valori de adevăr pentru literalii care o compun, care o fac adevărată.

Clauza vidă $\{\}$ este tot o clauză. Este echivalentă cu o disjuncție vidă și deci este *nerealizabilă* (engl. “unsatisfiable”, care nu poate fi adevărată în nicio interpretare). După cum vom vedea, aceasta joacă un rol important în procesul de rezoluție.

Rezoluția propozițională este o regulă puternică de inferență pentru logica propozițională, cu ajutorul căreia se poate construi un demonstrator de teoreme corect și complet. Ea poate fi aplicată însă doar după aducerea premiselor și concluziei într-o formă standardizată, numită *formă normal conjunctivă*.

Există o procedură simplă pentru a aduce o propoziție arbitrară în forma normal conjunctivă, adică sub forma unei conjuncții de disjuncții (un *SI* de *SAU*-uri). Regulile de conversie sunt prezentate mai jos și trebuie aplicate în ordine.

1. Implicațiile (I):

$$\begin{aligned}\varphi \Rightarrow \psi &\rightarrow \neg\varphi \vee \psi \\ \varphi \Leftarrow \psi &\rightarrow \varphi \vee \neg\psi \\ \varphi \Leftrightarrow \psi &\rightarrow (\neg\varphi \vee \psi) \wedge (\varphi \vee \neg\psi)\end{aligned}$$

2. Negațiile (N):

$$\begin{aligned}\neg\neg\varphi &\rightarrow \varphi \\ \neg(\varphi \wedge \psi) &\rightarrow \neg\varphi \vee \neg\psi \\ \neg(\varphi \vee \psi) &\rightarrow \neg\varphi \wedge \neg\psi\end{aligned}$$

3. Distribuția (D):

$$\begin{aligned}\varphi \vee (\psi \wedge \chi) &\rightarrow (\varphi \vee \psi) \wedge (\varphi \vee \chi) \\ (\varphi \wedge \psi) \vee \chi &\rightarrow (\varphi \vee \chi) \wedge (\psi \vee \chi) \\ \varphi \vee (\varphi_1 \vee \dots \vee \varphi_n) &\rightarrow \varphi \vee \varphi_1 \vee \dots \vee \varphi_n \\ (\varphi_1 \vee \dots \vee \varphi_n) \vee \varphi &\rightarrow \varphi_1 \vee \dots \vee \varphi_n \vee \varphi \\ \varphi \wedge (\varphi_1 \wedge \dots \wedge \varphi_n) &\rightarrow \varphi \wedge \varphi_1 \wedge \dots \wedge \varphi_n \\ (\varphi_1 \wedge \dots \wedge \varphi_n) \wedge \varphi &\rightarrow \varphi_1 \wedge \dots \wedge \varphi_n \wedge \varphi\end{aligned}$$

De exemplu, să considerăm următoarea conversie:

$$\begin{array}{ll} I & \neg(g \wedge (r \Rightarrow f)) \\ & \neg(g \wedge (\neg r \vee f)) \\ N & \neg g \vee \neg(\neg r \vee f) \\ & \neg g \vee (\neg \neg r \wedge \neg f) \\ & \neg g \vee (r \wedge \neg f) \\ D & (\neg g \vee r) \wedge (\neg g \vee \neg f)\end{array}$$

3. Algoritmul de rezoluție propozițională

Idea de bază a acestei forme de raționament este deducerea din două propoziții, în care unul din termeni apare cu valori de adevăr contrare, a unei concluzii din care este eliminat termenul respectiv. Fie următorul exemplu:

Afirmații:

Afară plouă sau este soare.
Dacă este soare, atunci este cald.

Reprezentări:

ploaie \vee *soare*
soare \Rightarrow *cald*

În forma normal conjunctivă, cea de a doua reprezentare va fi:

\neg *soare* \vee *cald*

Se observă că *soare* apare în prima expresie afirmat și în a doua negat. Ambele propoziții trebuie să fie adevărate. Dacă *soare* este adevărat, atunci *ploaie* poate lua orice valoare, pe când *cald* trebuie să fie obligatoriu adevărat. Dacă *soare* este fals, atunci *ploaie* trebuie să fie obligatoriu adevărat, pe când *cald* poate lua orice valoare. În orice caz, una din propozițiile *ploaie* sau *cald* trebuie să fie adevărată. Acest lucru corespunde formulei *ploaie* \vee *cald*.

Pe acest fapt se bazează procesul de rezoluție: din două clauze: *p* \vee *q* și \neg *p* \vee *r*, rezultă: *q* \vee *r*.

Cu alte cuvinte, din două clauze în care una conține un literal iar cealaltă negația acelui literal, putem deduce o clauză care conține reuniunea literalilor celor două clauze-premisă, fără perechea complementară.

Într-o formulare mai generală, rezoluția propozițională se bazează pe următoarea regulă de inferență:

$$\begin{array}{c} \varphi_1 \vee \dots \vee \varphi_{i-1} \vee \cancel{\chi} \vee \varphi_{i+1} \vee \dots \vee \varphi_n \\ \psi_1 \vee \dots \vee \psi_{j-1} \vee \cancel{\chi} \vee \psi_{j+1} \vee \dots \vee \psi_m \\ \hline \varphi_1 \vee \dots \vee \varphi_{i-1} \vee \varphi_{i+1} \vee \dots \vee \varphi_n \vee \psi_1 \vee \dots \vee \psi_{j-1} \vee \psi_{j+1} \vee \dots \vee \psi_m \end{array}$$

sau echivalent:

$$\begin{array}{c} C_1 = \{ \dots, \cancel{\chi}, \dots \} \\ C_2 = \{ \dots, \neg \cancel{\chi}, \dots \} \\ \hline C_1 \setminus \{ \cancel{\chi} \} \cup C_2 \setminus \{ \neg \cancel{\chi} \} \end{array}$$

Linia punctată reprezintă o *implicație*. Clauza produsă se mai numește *rezolvent*.

În exemplul de mai sus, dacă avem clauzele $p \vee q$, respectiv $\neg p \vee r$, putem deriva clauza $q \vee r$ într-un singur pas:

$$\begin{array}{c} p \vee q \\ \neg p \vee r \\ \hline q \vee r \end{array}$$

Trebuie menționat că, deoarece clauzele sunt mulțimi, un literal nu poate apărea de două ori într-o clauză, de exemplu:

$$\begin{array}{c} \neg p \vee q \\ p \vee q \\ \hline q \end{array}$$

Dacă o clauză-premisă este o mulțime singleton (cu un singur element), numărul de literali din rezolvent este mai mic decât numărul de literali din a două premissă. De exemplu, din clauza $p \vee q \vee r$ și clauza singleton $\neg p$, putem deriva clauza mai scurtă $q \vee r$:

$$\begin{array}{c} p \vee q \vee r \\ \neg p \\ \hline q \vee r \end{array}$$

Rezolvarea a două clauze singleton produce o clauză vidă, ca mai jos. Derivarea clauzei vide indică faptul că mulțimea de clauze, sau mulțimea de fapte, conține o contradicție:

$$\begin{array}{c} p \\ \neg p \\ \hline \{\} \end{array}$$

La rezolvarea a două clauze, pot exista mai mulți rezolvenți dacă există mai multe perechi de literali complementari:

$$\begin{array}{c} p \vee q \\ \neg p \vee \neg q \\ \hline \\ p \vee \neg p \\ q \vee \neg q \end{array}$$

Atunci când două clauze conțin mai multe perechi de literali complementari, o singură pereche poate genera un rezolvent. Exemplul următor nu este corect:

$$\begin{array}{c} p \vee q \\ \neg p \vee \neg q \\ \hline \textcolor{red}{Greșit!} \\ \{\} \end{array}$$

Dacă am permite această implicație, am trage concluzia că aceste două clauze sunt inconsistente. Însă, este posibil ca $(p \vee q)$ și $(\neg p \vee \neg q)$ să fie adevărate simultan. De exemplu, dacă p este adevărat și q este fals, ambele clauze sunt satisfăcute.

4. Demonstrarea automată a propozițiilor

Rezoluția nu este singura metodă de raționament din logica propozițională. De exemplu, există și metoda raționamentului înainte, care folosește o succesiune de implicații, de genul celor din demonstrațiile matematice clasice: $p \rightarrow q$, $q \rightarrow r$, deci $p \rightarrow r$. Prin urmare, dacă p , atunci r . Avantajul rezoluției este că propozițiile nu trebuie să fie doar implicații. De exemplu, $p \vee q$ nu poate fi pusă sub forma unei implicații.

De multe ori, procesul de rezoluție poate fi aplicat pentru a deriva direct o concluzie din premise, de exemplu, din premisele $\neg p \vee r$, $\neg q \vee r$ și $p \vee q$, se poate deriva concluzia r :

- | | | |
|----|-----------------|---------|
| 1. | $\neg p \vee r$ | Premisă |
| 2. | $\neg q \vee r$ | Premisă |
| 3. | $p \vee q$ | Premisă |
| 4. | $q \vee r$ | 1, 3 |
| 5. | r | 2, 4 |

Trebuie însă subliniat că rezoluția nu este completă din punct de vedere generativ, adică nu se pot deriva prin rezoluție toate clauzele care sunt implicate logic din mulțimea de premise. De exemplu, din premisele p și q , nu se poate deriva clauza $p \vee q$, chiar dacă aceasta este implicată logic de premise.

Totuși, dacă o mulțime de clauze Δ este nerealizabilă, rezoluția garantează derivarea clauzei vide din Δ . În exemplul următor, cele 4 premise nu pot fi adevărate simultan pentru nicio combinație de valori logice pentru p și q :

1.	$p \vee q$	Premise
2.	$p \vee \neg q$	Premise
3.	$\neg p \vee q$	Premise
4.	$\neg p \vee \neg q$	Premise
5.	p	1, 2
6.	$\neg p$	3, 4
7.	{}	5, 6

Această relație dintre nerealizabilitate și implicația logică poate fi folosită pentru determinarea implicațiilor logice, adică pentru demonstrarea generală a teoremelor. Pentru aceasta, se neagă concluzia, se adaugă la mulțimea de premise și se folosește rezoluția pentru a determina dacă mulțimea de clauze rezultată este nerealizabilă.

O propoziție φ este demonstrabilă dintr-o mulțime Δ de propoziții dacă și numai dacă procesul de rezoluție propozițională generează clauza vidă din mulțimea $\Delta \cup \{\neg\varphi\}$.

Ca exemplu, fie următoarea problemă. Se dau 3 premise: p , $p \Rightarrow q$, $(p \Rightarrow q) \Rightarrow (q \Rightarrow r)$ și trebuie demonstrat r . Procesul de rezoluție este descris mai jos. Clauzele 3 și 4 rezultă din aducerea premisei 3 în forma normal conjunctivă: $(p \vee \neg q \vee r) \wedge (\neg q \vee r)$.

1.	p	Premisa 1
2.	$\neg p \vee q$	Premisa 2
3.	$p \vee \neg q \vee r$	Premisa 3 FNC
4.	$\neg q \vee r$	Premisa 3 FNC
5.	$\neg r$	Concluzia negată
6.	q	1, 2
7.	r	4, 6
8.	{}	5, 7

5. Decidabilitatea

Un sistem logic este *decidabil* dacă există o metodă eficientă care determină dacă o formulă arbitrară este o teoremă a sistemului logic considerat, adică dacă se poate stabili dacă o formulă este adevărată sau nu. *Logica propozițională este decidabilă*. De exemplu, cu algoritmul de rezoluție propozițională, o demonstrație se termină întotdeauna: deoarece există un număr finit de clauze care pot fi generate dintr-o mulțime inițială finită de clauze, la un moment dat algoritmul nu mai poate genera noi clauze. Dacă până în acel moment a ajuns la o contradicție, demonstrația a reușit. Dacă nu, concluzia propusă nu poate fi demonstrată.

Spre deosebire de logica propozițională, în logica predicativă de ordin întâi, problema demonstrării teoremelor este *semidecidabilă*: se poate afla dacă o propoziție se poate demonstra, dar nu se poate afla dacă o propoziție nu se poate demonstra. Algoritmul de rezoluție predicativă poate rula la infinit.

6. Aplicații

6.1a. Fie următoarele premise:

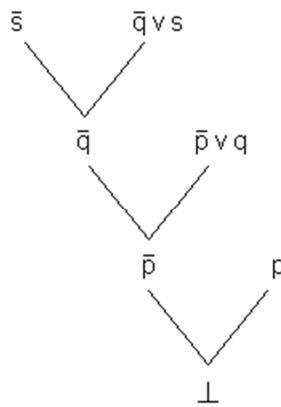
$$\begin{cases} p \\ p \rightarrow q \\ (q \vee r) \rightarrow s \end{cases}$$

Demonstrați prin rezoluție propozițională adevărul propoziției s .

Indicații. În primul rând, cele trei premise trebuie aduse la forma normal conjunctivă:

$$\begin{cases} p \\ \neg p \vee q \\ (\neg q \wedge \neg r) \vee s \end{cases} \equiv \begin{cases} p \\ \neg p \vee q \\ \neg q \vee s \\ \neg r \vee s \end{cases}$$

Presupunem că $\neg s$ este adevărat și încercăm să ajungem la o contradicție:



6.1b. Pentru aceleași premise ca la problema 1, încercați să demonstrați adevărul propoziției r .

6.2. Date fiind premisele $p \Rightarrow q$ și $r \Rightarrow s$, demonstrați că $(p \vee r) \Rightarrow (q \vee s)$.

Transformați mai întâi aceste propoziții, pe hârtie, în forma normal conjunctivă.

Indicații privind programul. Se dă un prototip de aplicație, în care sunt implementate toate clasele necesare pentru rezolvare. Diagrama UML de clase a aplicației complete este prezentată în pagina următoare.

Se observă următoarele clase principale:

Literal: Reprezintă un literal, de exemplu „p” sau „NOT p”.

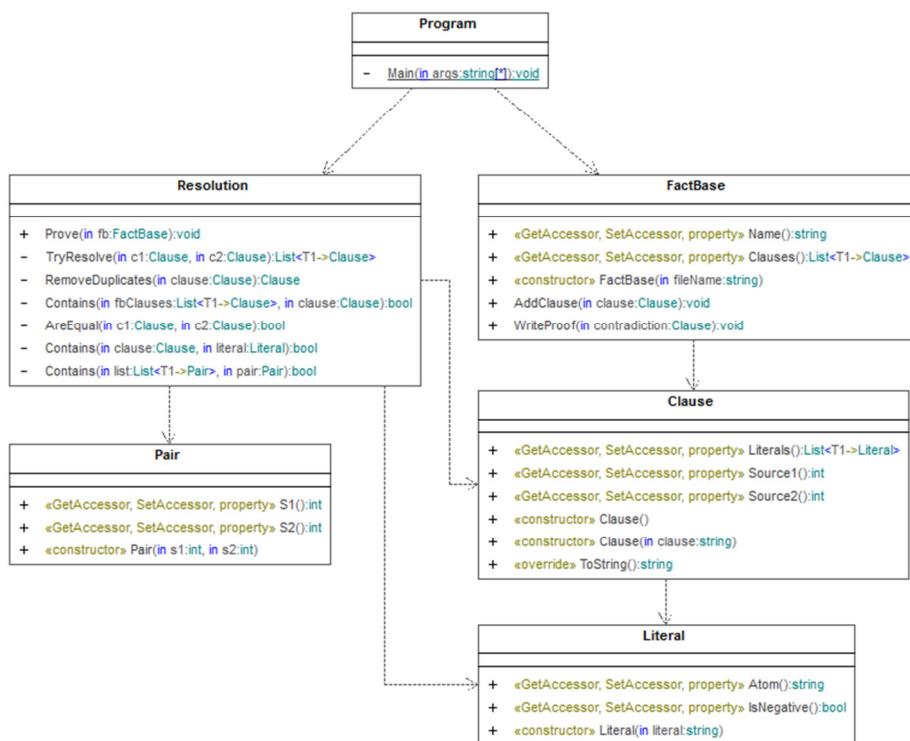
Clause: Reprezintă o clauză, adică o disjuncție de literali, de exemplu: „p OR NOT q OR r”. Conține o listă cu literalii din disjuncție.

FactBase: Corespunde întregii mulțimi de clauze asupra căreia se desfășoară procesul de inferență. Conține o listă de clauze. Primele clauze, premisele, sunt citite dintr-un fișier cu extensia .facts, care pentru prima problemă arată astfel (ex1a.facts):

```
p
NOT p OR q
NOT q OR s
NOT r OR s

NOT s
```

Tot în lista de clauze din FactBase vor fi adăugate clauzele noi, generate în cadrul procesului de rezoluție.



Trebuie implementate doar următoarele metode:

Metoda Prove(FactBase fb) din clasa Resolution

Metoda care implementează algoritmul. Primește o bază de fapte și încearcă să găsească o contradicție. Apelează metoda TryResolve.

Metoda TryResolve(Clause c1, Clause c2) din clasa Resolution

Metoda care încearcă să găsească literali complementari în cele două clauze primite ca parametri. Întoarce o listă de rezolvenți, posibil vidă.

Pentru primele probleme din laborator, rezultatele ar trebui să fie de forma:

ex1a.proof

0. p
1. NOT p OR q
2. NOT q OR s
3. NOT r OR s
4. NOT s
5. q <- 0+1
6. NOT p OR s <- 1+2
7. NOT q <- 2+4
8. s <- 2+5
9. NOT r <- 3+4
10. NOT p <- 4+6
11. {} <- 4+8

Demonstratie:

0. p
1. NOT p OR q
2. NOT q OR s
4. NOT s
5. q <- 0+1
8. s <- 2+5
11. {} <- 4+8

ex1b.proof

0. p
1. NOT p OR q
2. NOT q OR s
3. NOT r OR s
4. NOT r
5. q <- 0+1
6. NOT p OR s <- 1+2
7. s <- 2+5

Propozitia nu poate fi demonstrata.

Inferența vagă (fuzzy)

1. Obiective

Obiectivele acestui laborator sunt următoarele:

- prezentarea mulțimilor fuzzy (se pronunță [fázi]) și a procesului de inferență fuzzy;
- implementarea unei aplicații care simulează un controler fuzzy: controlul mișcării unui metrou care trebuie să plece dintr-o stație și să oprească la timp în stația următoare.

2. Mulțimi fuzzy

Logica tradițională consideră că un obiect poate aparține sau nu unei mulțimi. Logica fuzzy permite o interpretare mai flexibilă a noțiunii de apartenență. Astfel, mai multe obiecte pot apartine unei mulțimi în grade diferite. De exemplu, dacă avem în vedere mulțimea oamenilor tineri. Un copil de 10 ani e cu siguranță Tânăr, în timp ce o persoană de 60 de ani cu siguranță nu. Dar un om de 30 de ani? Sau de 40? În acest caz, putem afirma că persoana de 30 de ani aparține mulțimii respective într-o măsură mai mare decât cea de 40.

Acest instrument de reprezentare și manipulare a termenilor vagi sau nuanțați se numește *logică fuzzy*. Mulțimile fuzzy permit elementelor să aparțină parțial unei clase sau mulțimi. Fiecărui element i se atribuie un grad de apartenență la o mulțime. Acest grad de apartenență poate lua valori între 0 (nu aparține mulțimii) și 1 (aparține total mulțimii). În cazul în care gradul de apartenență ar fi doar 0 sau 1, mulțimea fuzzy ar fi echivalentă cu o mulțime binară.

Fie X universul discursului, cu elemente notate x . O mulțime fuzzy A a universului de discurs X este caracterizată de o funcție de apartenență $\mu_A(x)$ care asociază fiecărui element x un grad de apartenență la mulțimea A :

$$\mu_A(x): X \rightarrow [0,1]$$

Pentru a reprezenta o mulțime fuzzy, trebuie să-i definim mai întâi funcția de apartenență. În acest caz, o mulțime fuzzy A este complet definită de mulțimea tuplelor:

$$A = \{(x, \mu_A(x)) \mid x \in X\}$$

Să presupunem că pentru exemplul amintit mai sus, cu variabila lingvistică „tânăr”, avem universul discursului $X = \{0, 20, 30, 50\}$ și următoarea funcție de apartenență: o persoană de 20 de ani aparține mulțimii oamenilor tineri în proporție de 90%, una de 30 de ani în proporție de 70% iar una de 50 de ani nu aparține mulțimii (gradul său de apartenență este 0). Aceste lucruri se reprezintă grafic ca în figura 1.

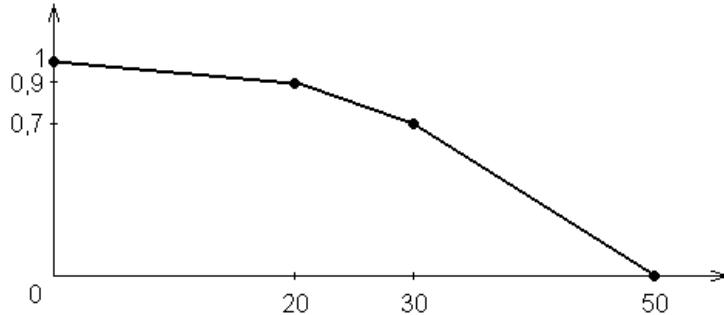


Figura 1. Funcție de apartenență pentru mulțimea oamenilor tineri

Pe un univers de discurs pot fi definite mai multe submulțimi fuzzy. De exemplu, pentru universul vîrstelor unor persoane, putem defini submulțimile oamenilor tineri, bătrâni sau de vîrstă mijlocie. Aceste submulțimi se pot intersecta (este chiar recomandat acest fapt). Aceeași persoană va aparține submulțimii oamenilor tineri cu un grad de 70%, submulțimii oamenilor de vîrstă mijlocie cu un grad de 90% și submulțimii oamenilor bătrâni cu un grad de 30%.

Un alt exemplu cu mai multe submulțimi fuzzy este prezentat în figura 2. Aici, submulțimile reprezintă numere negative mari, medii, mici, negative tinzând la zero, pozitive tinzând la zero, mici, medii, și mari. Valoarea μ reprezintă gradul de apartenență la mulțimea fuzzy.

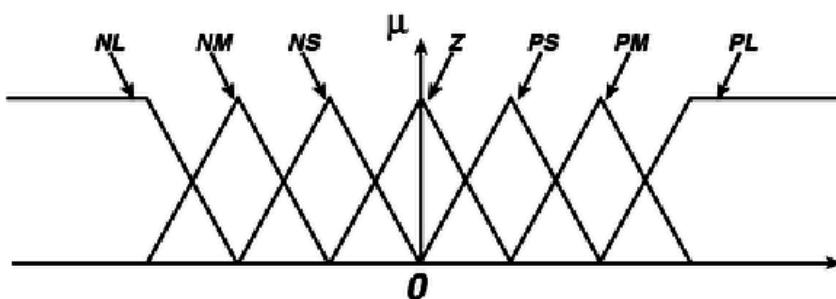


Figura 2. Submulțimi fuzzy pentru mulțimea numerelor reale
(N = Negativ, P = Pozitiv, L = Mare, M = Mediu, S = Mic)

2.1. Noțiuni fundamentale

Fie A o submulțime fuzzy a universului de discurs X . Se numește *suportul* lui A submulțimea strictă a lui X ale cărei elemente au grade de apartenență nenule în A :

$$supp(A) = \{x \in X \mid \mu_A(x) > 0\}.$$

Înălțimea lui A se definește drept cea mai mare valoare a funcției de apartenență:

$$h(A) = \max_{x \in X} \mu_A(x).$$

Se numește *nucleul* lui A submulțimea strictă a lui X ale cărei elemente au grade de apartenență unitare în A :

$$n(A) = \{x \in X \mid \mu_A(x) = 1\}.$$

2.2. Numere fuzzy

De multe ori, oamenii nu pot caracteriza precis informațiile numerice, folosind formulări precum „aproape 0”, „în jur de 100” etc. În teoria mulțimilor fuzzy, aceste numere pot fi reprezentate ca submulțimi fuzzy ale mulțimii numerelor reale.

Un *număr fuzzy* este o mulțime fuzzy a mulțimii numerelor reale, cu o funcție de apartenență convexă și continuă și suport mărginit.

O mulțime fuzzy A se numește *număr fuzzy triunghiular* cu centrul c , lățimea la stânga $\alpha > 0$ și lățimea la dreapta $\beta > 0$ dacă funcția sa de apartenență are forma:

$$\mu_A(x) = \begin{cases} 1 - \frac{c-x}{\alpha}, & c-\alpha \leq x \leq c \\ 1 - \frac{x-c}{\beta}, & c < x \leq c + \beta \\ 0, & \text{altfel} \end{cases}$$

Folosim notația: $A = (c, \alpha, \beta)$. Este evident că $\text{supp}(A) = (c - \alpha, c + \beta)$. Semnificația unui număr fuzzy triunghiular cu centrul c este „ x este aproximativ egal cu c ”.

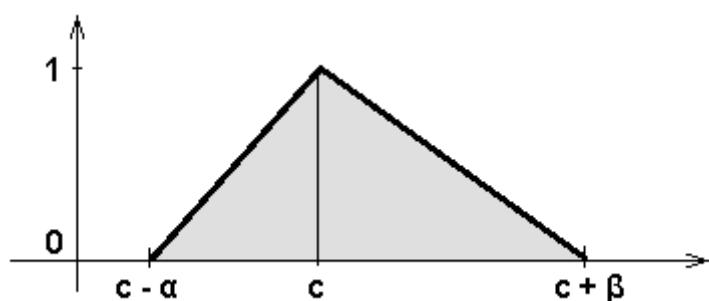


Figura 3. Număr fuzzy triunghiular

O mulțime fuzzy A se numește *număr fuzzy trapezoidal* cu intervalul de toleranță $[c, d]$, lățimea la stânga $\alpha > 0$ și lățimea la dreapta $\beta > 0$ dacă funcția sa de apartenență are forma:

$$\mu_A(x) = \begin{cases} 1 - \frac{c-x}{\alpha}, & c-\alpha \leq x \leq c \\ 1, & c < x \leq d \\ 1 - \frac{x-d}{\beta}, & d < x \leq d+\beta \\ 0, & \text{altfel} \end{cases}$$

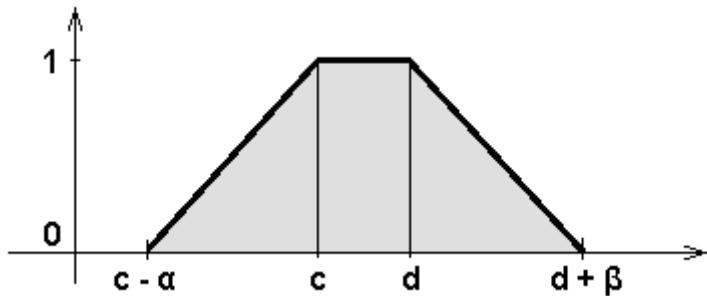


Figura 4. Număr fuzzy trapezoidal

Aici notăm: $A = (c, d, \alpha, \beta)$, iar $\text{supp}(A) = (c - \alpha, d + \beta)$. Semnificația unui număr fuzzy trapezoidal cu intervalul de toleranță $[c, d]$ este „ x este aproximativ între c și d ”.

3. Inferența Mamdani (max-min) cu o singură regulă

3.1. Modus Ponens generalizat

În logica fuzzy și raționamentul aproximativ, cea mai importantă regulă de inferență este *Modus Ponens generalizat*.

În logica clasică, această regulă de inferență este de forma $(p \wedge (p \rightarrow q)) \rightarrow q$, adică:

regulă: dacă p , atunci q

premisă: p

concluzie: q

În logica fuzzy, regula de inferență corespunzătoare este următoarea:

regulă: dacă x este A , atunci y este B

premisă (antecedent): x este A'

concluzie (consecvent): y este B' , unde $B' = A' \circ (A \rightarrow B)$.

Dacă $A' = A$ și $B' = B$, regula se reduce la Modus Ponens clasic.

Matricea $A \rightarrow B$ deseori se notează cu R . Procesul de inferență fuzzy este văzut ca o transformare a unei mulțimi fuzzy într-o altă mulțime fuzzy. Submulțimea indușă în B , B' se calculează astfel: $b_j' = \max_i (\min(a_i', r_{ij}))$. Există mai multe metode de definire a matricei R . În cele ce urmează, noi vom folosi tipul de inferență Mamdani, prin care mulțimea B' este o variantă

„retezată” a lui B , la înălțimea fixată de A' . A' poate fi o submulțime normală a lui A , nu doar cu un singur element.

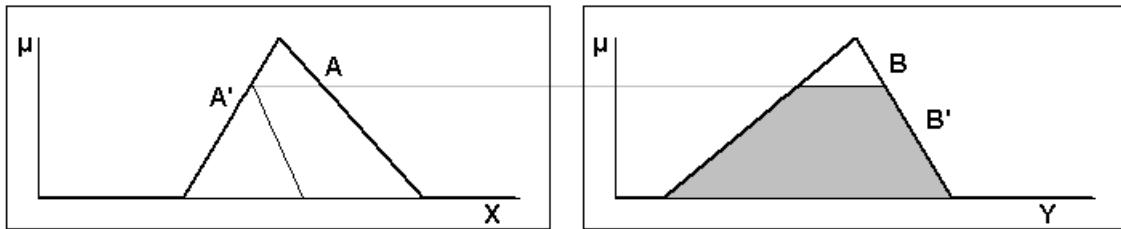


Figura 5. Inferență de tip Mamdani

3.3. Defuzzificarea

După ce am determinat mulțimea fuzzy indusă de o regulă de inferență, în unele aplicații trebuie obținută o valoare singulară, strictă, pe baza acestei mulțimi. Procesul se numește *defuzzificare*. Cea mai utilizată tehnică de defuzzificare este *metoda centrului de greutate* (sau a *centroidului*):

$$x_{CG} = \frac{\sum_i x_i \cdot \mu_A(x_i)}{\sum_i \mu_A(x_i)}$$

În figura 6, programul folosește drept universuri de discurs intervalele (0,100). În figura din dreapta, mulțimea delimitată cu albastru este B , cea delimitată cu roz este B' iar valoarea pe axa X indicată cu roșu este centrul de greutate.

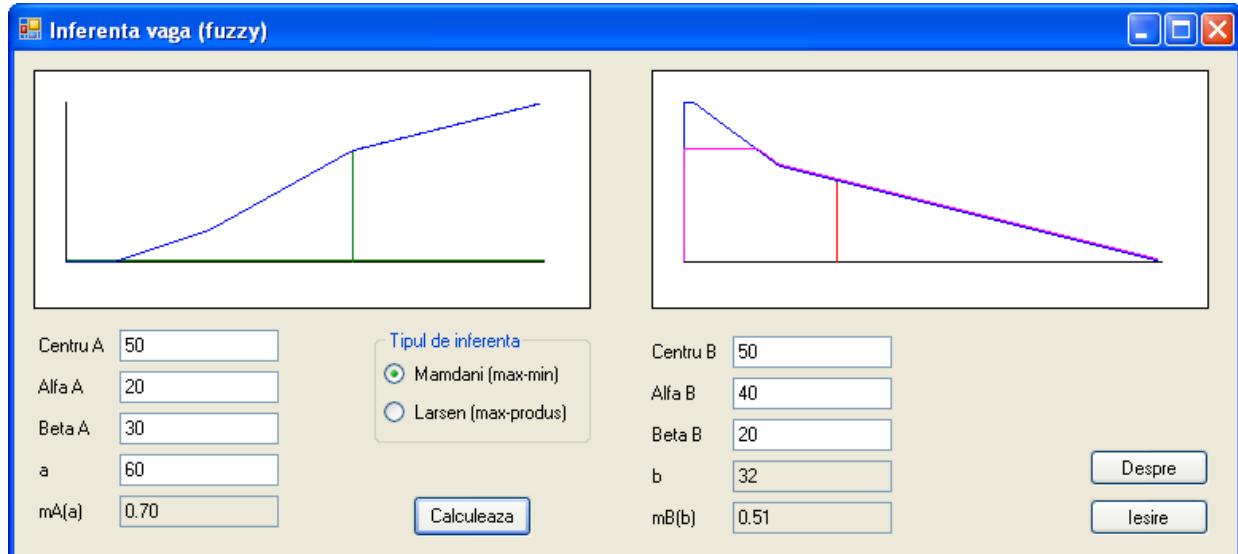


Figura 6. Inferență de tip Mamdani și defuzzificare

4. Inferență Mamdani cu reguli multiple

Un exemplu de sistem de inferență fuzzy de tip Mamdani este prezentat în figura 7. Pentru a calcula ieșirea acestui sistem când se dău intrările trebuie parcursi următorii 6 pași:

1. Se determină o mulțime de reguli fuzzy;
2. Se realizează fuzzificarea intrărilor utilizând funcțiile de apartenență;
3. Se combină intrările fuzzificate urmând regulile fuzzy pentru stabilirea puterilor de activare ale regulilor;
4. Se calculează consecvenții regulilor prin combinarea puterilor de activare ale regulilor cu funcțiile de apartenență ale ieșirilor;
5. Se combină consecvenții pentru a determina mulțimea de ieșire;
6. Se defuzzifică mulțimea de ieșire, doar dacă se dorește ca ieșirea să fie strictă.

În cele ce urmează se prezintă o descriere detaliată a acestui proces.

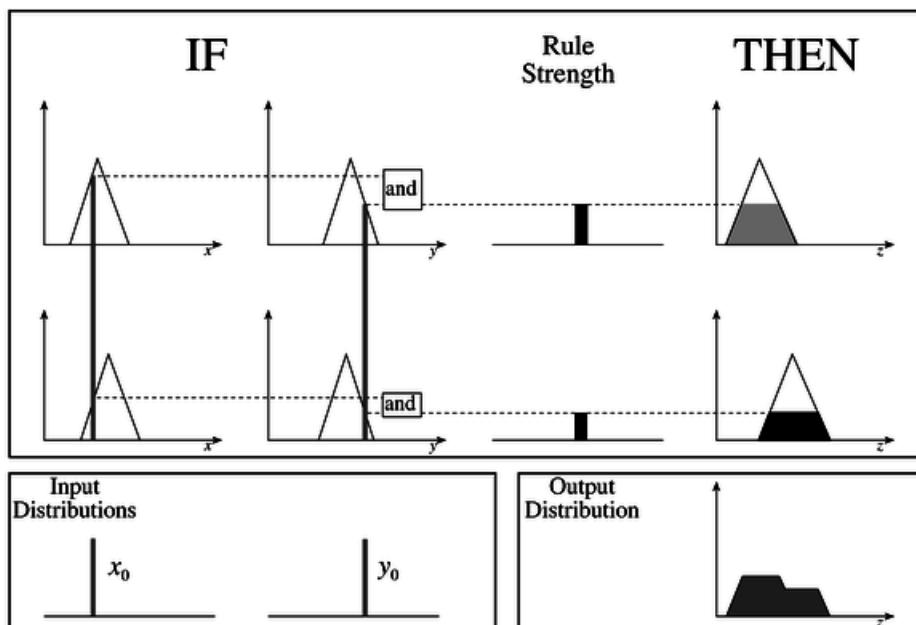


Figura 7. Sistem de inferență fuzzy de tip Mamdani cu două reguli și două intrări stricte

4.1. Crearea regulilor fuzzy

Regulile fuzzy reprezintă o mulțime de afirmații care descriu modul în care sistemul poate lua o decizie privind estimarea ieșirii. Regulile fuzzy au următoarea formă:

DACĂ (*intrarea-1 este mulțime-fuzzy-1*) **ȘI/SAU**
(intrarea-2 este mulțime-fuzzy-2) **ȘI/SAU ...**
ATUNCI (*ieșirea este mulțime-fuzzy-de-ieșire*).

Un exemplu de regulă scrisă în acest mod este următorul:

DACĂ finanțarea-proiectului este adekvată și numărul de angajați este redus
ATUNCI riscul-proiectului este mic.

4.2. Fuzzificarea

Scopul fuzzificării este de a transforma intrările în valori cuprinse între 0 și 1 utilizând funcțiile de apartenență. În exemplul din figura 8 există două intrări x_0 și y_0 prezentate în colțul din stânga jos. Pentru aceste intrări stricte se marchează gradele de apartenență în mulțimele corespunzătoare.

4.3. Combinarea antecedenților multipli

În crearea regulilor fuzzy utilizăm operatorii $\$I$, SAU și uneori **NEGATIE**. Operatorul fuzzy $\$I$ este scris ca: $\mu_{A \cap B}(x) = T(\mu_A(x), \mu_B(x))$ unde T este o funcție numită *T-normă*, $\mu_A(x)$ este gradul de apartenență a lui x la mulțimea A , iar și $\mu_B(x)$ este gradul de apartenență a lui x la mulțimea B . Cu toate că există mai multe moduri pentru calculul funcției $\$I$, cel mai des folosit este: $\min(\mu_A(x), \mu_B(x))$. Operatorul fuzzy $\$I$ reprezintă o generalizare a operatorului logic boolean $\$I$ în sensul că valoarea de adevăr a unei propoziții nu este doar 0 sau 1, ci poate fi cuprinsă între 0 și 1. O funcție *T-normă* este monotonă, comutativă, asociativă și respectă condițiile $T(0, 0) = 0$ și $T(x, 1) = x$.

Operatorul fuzzy SAU se scrie ca $\mu_{A \cup B} = S(\mu_A, \mu_B)$ unde S este o funcție numită *T-conormă*. În mod similar cu operatorul $\$I$, aceasta poate fi: $\max(\mu_A(x), \mu_B(x))$. Operatorul fuzzy SAU reprezintă de asemenea o generalizare a operatorului logic boolean SAU la valori cuprinse între 0 și 1. O funcție *T-conormă* este monotonă, comutativă, asociativă și respectă condițiile $S(x, 0) = x$ și $S(1, 1) = 1$.

4.4. Calcularea consecvenților

Mai întâi, se calculează puterile de activare ale regulilor, după cum s-a prezentat în secțiunea 4.3. În figura 8, se poate observa că este utilizat operatorul fuzzy $\$I$ asupra funcțiilor de apartenență pentru a calcula puterile de activare ale regulilor. Apoi, pentru un sistem de inferență fuzzy de tip Mamdani, mulțimea de ieșire este reținută la nivelul dat de puterea de activare a regulii, așa cum s-a arătat și în secțiunea 3.2.

4.5. Agregarea ieșirilor

Ieșirile obținute după aplicarea regulilor fuzzy sunt combinate pentru a se obține mulțimea de ieșire. De obicei, acest lucru se realizează utilizând operatorul fuzzy SAU . În figura 8, funcțiile de apartenență din partea dreaptă sunt combinate utilizând operatorul fuzzy SAU pentru a obține mulțimea de ieșire prezentată în colțul din dreapta jos.

4.6. Defuzzificarea

De multe ori se dorește obținerea unei ieșiri stricte. De exemplu, dacă se încearcă clasificarea literelor scrise de mâna pe o tabletă, sistemul de inferență fuzzy trebuie să genereze un număr strict care poate fi interpretat. Acest număr se obține în urma procesului de defuzzificare. Cea mai des utilizată metodă de defuzzificare este metoda centrului de greutate, prezentată în secțiunea 3.3 și exemplificată și în figura 8.

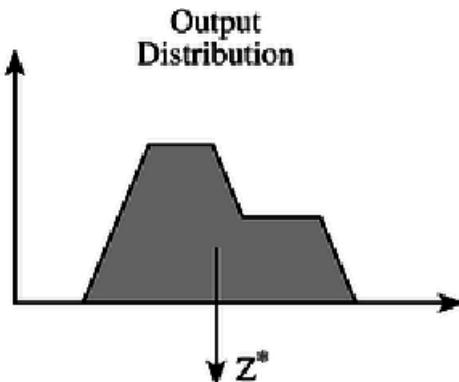


Figura 8. Defuzzificarea utilizând metoda centrului de greutate

5. Aplicație

În cadrul acestei aplicații, vom simula un sistem fuzzy de control al unui metrou care trebuie să parcurgă 1 km între două stații. Scopul este ca metroul să plece din prima stație și să opreasă la timp în a două stație. Sistemul de control poate fi văzut ca o funcție care calculează în fiecare moment accelerația metroului, dată fiind distanța rămasă până la destinație.

Se dă un prototip de aplicație, în care sunt implementate toate clasele necesare pentru rezolvare și interfață grafică cu utilizatorul. Mai jos este diagrama UML de clase a aplicației complete.

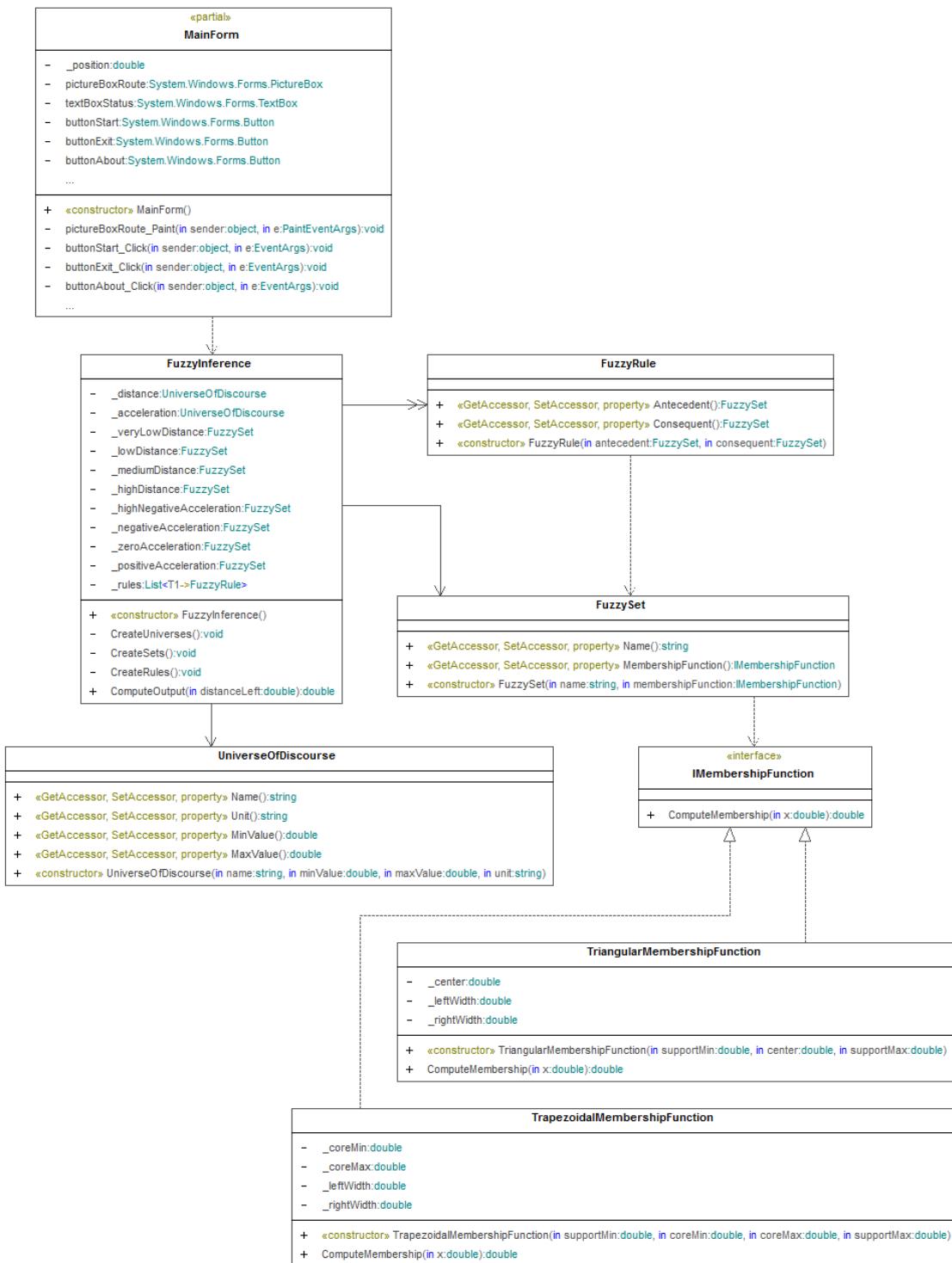
5.1. Implementați mai întâi algoritmul de inferență fuzzy cu reguli multiple, completând metodele de mai jos.

Metoda ComputeMembership(double x) din clasa TriangularMembershipFunction

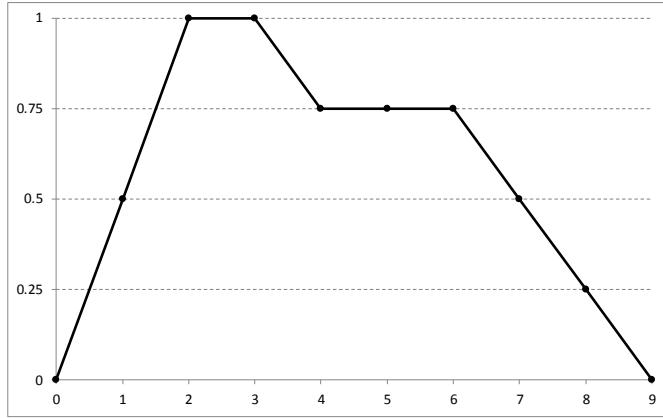
Metoda care calculează valoarea funcției de apartenență a unui număr fuzzy triunghiular în punctul x. Se poate completa pe baza funcției din laborator și/sau prin analogie cu expresia funcției de apartenență pentru numerele fuzzy trapezoidale.

Metoda ComputeOutput(double distanceLeft) din clasa FuzzyInference

Metoda care implementează procesul efectiv de inferență Mamdani cu reguli multiple. Metoda primește ca parametru distanța rămasă și folosește informațiile date de regulile și mulțimile fuzzy pentru a calcula accelerația metroului.



Indicație. În program, mulțimile fuzzy trebuie reprezentate ca niște vectori, prin eșantionare. De exemplu, în figura de mai jos este prezentată o mulțime fuzzy eșantionată într-un vector cu 10 elemente, cu indexul plecând de la 0:



Vectorul corespunzător mulțimii este: $(0, 0.5, 1, 1, 0.75, 0.75, 0.75, 0.5, 0.25, 0)$.

În cazul general, în care universul de discurs la mulțimii fuzzy este $[x_{min}, x_{max}]$ și dorim să eșantionăm mulțimea într-un vector v cu n elemente, avem următoarele corespondențe:

$$\begin{aligned} v[0] &\rightarrow \mu(x_{min}) \\ v[n-1] &\rightarrow \mu(x_{max}). \end{aligned}$$

Pentru un index i între 0 și n al vectorului v , corespondentul va fi o valoare a funcției de apartenență a mulțimii fuzzy într-un punct intermedian x_i dintre x_{min} și x_{max} , după regula de trei simplă. Prin urmare, valoarea unui element din vector va fi: $v[i] = \mu(x_i)$, unde:

$$x_i = x_{min} + \frac{i}{n-1} \cdot (x_{max} - x_{min}).$$

5.2. Adăugați mulțimi și reguli fuzzy potrivite pentru a face metroul să funcționeze corect, adică să opreasă exact în stație.

Rețele bayesiene

1. Obiective

Obiectivul acestui laborator este de a prezenta structura rețelelor bayesiene și de a descrie un algoritm exact de inferență, inferența prin enumerare. Folosind un program de lucru cu rețele bayesiene, se vor modela două rețele și se vor explora probabilitățile nodurilor în prezenta diferitelor observații.

2. Probabilități condiționate. Teorema lui Bayes

Vom aminti câteva noțiuni legate de probabilitățile condiționate. Când trebuie să definim $P(A|B)$, presupunem că se cunoaște B și se calculează probabilitatea lui A în această situație. Să considerăm evenimentul D (durere de cap) și să presupunem că are, în general, o probabilitate de 1/10. Probabilitatea de a avea gripă (evenimentul G) este de numai 1/40. După cum se vede în figura 1, dacă cineva are gripă, probabilitatea de a avea și dureri de cap este de 1/2. Deci probabilitatea durerii de cap, dată fiind gripe, este de 1/2. Această probabilitate corespunde intersecției celor două regiuni, cu aria egală cu jumătate din G .

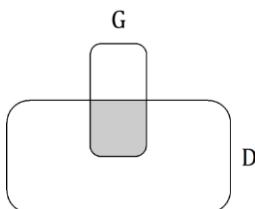


Figura 1. Reprezentare grafică a unei probabilități condiționate

Pe baza acestei relații rezultă *teorema lui Bayes*, care este importantă pentru toate raționamentele probabilistice pe care le vom studia.

Considerăm formula probabilităților condiționate:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}.$$

Putem exprima probabilitatea intersecției în două moduri și de aici deducem expresia lui $P(B|A)$ în funcție de $P(A|B)$:

$$P(A \cap B) = P(A|B) \cdot P(B),$$

$$P(A \cap B) = P(B|A) \cdot P(A),$$

$$\Rightarrow P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}.$$

Această ecuație reprezintă un rezultat fundamental. Mai clar, putem considera următoarea expresie alternativă:

$$P(I|E) = \frac{P(E|I) \cdot P(I)}{P(E)},$$

unde I este ipoteza, E este evidența (provenind din datele observate), $P(I)$ este *probabilitatea a-priori* a ipotezei, adică gradul inițial de încredere în ipoteză, $P(E|I)$ este *verosimilitatea* datelor observate (engl. “likelihood”), adică măsura în care s-a observat evidența în condițiile îndeplinirii ipotezei, iar $P(I|E)$ este *probabilitatea a-posteriori* a ipotezei, dată fiind evidența.

Relația este importantă deoarece putem calcula astfel probabilitățile cauzelor, date fiind efectele. Este mai simplu de cunoscut când o cauză determină un efect, dar invers, când cunoaștem un efect, probabilitățile cauzelor nu pot fi cunoscute imediat. Teorema ne ajută să diagnosticăm o anumită situație sau să testăm o ipoteză.

Să considerăm următorul exemplu de diagnosticare. Știm că probabilitatea de apariție a meningitei în populația generală este $P(M) = 0,002\%$. De asemenea, probabilitatea ca o persoană să aibă gâtul înțepenit este $P(G) = 5\%$. Mai știm că meningita cauzează gât înțepenit în jumătate din cazuri: $P(G|M) = 50\%$.

Dorim să aflăm următorul lucru: dacă un pacient are gâtul înțepenit, care este probabilitatea să aibă meningită?

Aplicând teorema lui Bayes, vom avea:

$$P(M|G) = \frac{P(G|M) \cdot P(M)}{P(G)} = 0,02\%.$$

G este un simptom pentru M . Dacă există simptomul, care este probabilitatea unei posibile cauze, adică $P(M)$? Rezultatul este $0,02\%$, deci o probabilitate mică, deoarece probabilitatea meningitei însăși este foarte mică în general.

3. Rețele bayesiene

În continuare, ne vom concentra asupra reprezentării informațiilor legate de evenimente probabilistice, care ne va ajuta să realizăm eficient raționamente.

Determinarea probabilității unei combinații de valori se poate realiza astfel:

$$P(x_1, \dots, x_n) = P(x_n|x_{n-1}, \dots, x_1) \cdot P(x_{n-1}, \dots, x_1).$$

Aplicând în continuare această regulă vom obține *regula de înmulțire a probabilităților* (engl. “chain rule”):

$$P(x_1, \dots, x_n) = P(x_n|x_{n-1}, \dots, x_1) \cdot P(x_{n-1}|x_{n-2}, \dots, x_1) \cdot P(x_2|x_1) \cdot P(x_1),$$

exprimată mai concis astfel:

$$P(x_1, \dots, x_n) = \prod_{i=1}^n P(x_i | x_{i-1}, \dots, x_1).$$

O rețea bayesiană arată ca în figura 2: este un graf orientat aciclic (engl. “directed acyclic graph”), în care evenimentele sau variabilele se reprezintă ca noduri, iar relațiile de corelație sau cauzalitate se reprezintă sub forma arcelor dintre noduri.

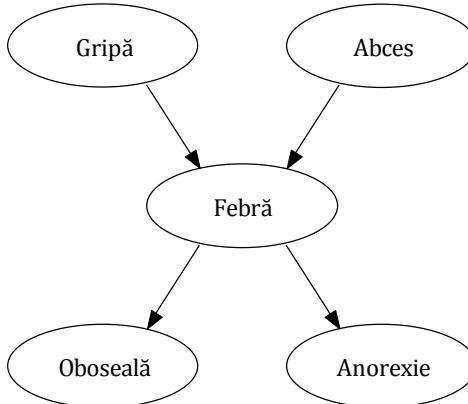


Figura 2. Rețea bayesiană

Tabelul 1. Tabelele de probabilități pentru rețeaua bayesiană

$P(Gripă = Da)$	$P(Gripă = Nu)$
0,1	0,9

$P(Abces = Da)$	$P(Abces = Nu)$
0,05	0,95

Gripă	Abces	$P(Febră = Da)$	$P(Febră = Nu)$
Da	Da	0,8	0,2
Da	Nu	0,7	0,3
Nu	Da	0,25	0,75
Nu	Nu	0,05	0,95

Febră	$P(Oboseală = Da)$	$P(Oboseală = Nu)$
Da	0,6	0,4
Nu	0,2	0,8

Febră	$P(Anorexie = Da)$	$P(Anorexie = Nu)$
Da	0,5	0,5
Nu	0,1	0,9

În acest exemplu, se consideră că atât gripa cât și abcesul pot determina febra. De asemenea, febra poate cauza o stare de oboseală sau lipsa poftei de mâncare (anorexie).

Sensul săgeților arcelor sunt dinspre părinti, cum ar fi gripa și abcesul, înspre fii, precum febra. Deși în acest exemplu relațiile sunt cauzale, în general o rețea bayesiană reflectă relații de

corelație, adică măsura în care aflarea unor informații despre o variabilă-părinte aduce noi informații despre o variabilă-fiu.

Fiecare variabilă are o mulțime de valori. În cazul cel mai simplu, variabilele au valori binare, de exemplu *Da* și *Nu*. În general însă, o variabilă poate avea oricâte valori.

Asociate cu variabilele, o rețea bayesiană conține o serie de tabele de probabilități, precum cele din tabelul 1. Pentru nodurile fără părinți se indică probabilitățile marginale ale fiecărei valori (adică fără a lua în considerare valorile celorlalte variabile). Pentru celelalte noduri, se indică probabilitățile condiționate pentru fiecare valoare, ținând cont de fiecare combinație de valori ale variabilelor părinte.

În general, o variabilă binară fără părinți va avea un singur parametru independent, o variabilă cu 1 părinte va avea 2 parametri independenți iar o variabilă cu n părinți va avea 2^n parametri independenți în tabela de probabilități corespunzătoare.

Presupunerea modelului bazat pe rețele bayesiene este că o variabilă nu depinde decât de părinții săi și deci ecuația anterioară devine:

$$P(x_1, \dots, x_n) = \prod_{i=1}^n P(x_i | \pi(x_i)),$$

unde $\pi(x_i)$ reprezintă mulțimea părinților variabilei x_i , din sirul ordonat topologic al nodurilor, în care părinții unui nod apar întotdeauna înaintea nodului respectiv.

4. Inferență prin enumerare

Folosind acest procedeu, putem răspunde practic la orice întrebare privind evenimentele codate în rețea. Având în vedere niște evidențe, adică observații sau evenimente despre care știm că s-au întâmplat, putem calcula probabilitățile tuturor celorlalte noduri din rețea.

Mai exact, scopul inferenței prin enumerare este de a calcula probabilitatea unei variabile interogate (engl. “query”), date fiind variabilele observe (evidență).

Ideea de bază este tot calcularea unui produs de probabilități condiționate, însă în cazul variabilelor despre care nu se cunoaște nimic (nu sunt nici observate și nici interogate), se sumează variantele corespunzătoare tuturor valorilor acestora.

Să considerăm următoarea întrebare: „Care este probabilitatea ca o persoană să aibă gripă, dacă prezintă simptome de oboseală și anorexie?”

Vom calcula independent $P(G_D | O_D, X_D)$ și $P(G_N | O_D, X_D)$.

Pentru $P(G_D | O_D, X_D)$, variabilele rămase sunt *Abcesul* și *Febra*. În consecință, vom suma probabilitățile corespunzătoare tuturor valorilor acestor variabile: $a \in \{A_D, A_N\}$ și $f \in \{F_D, F_N\}$. De asemenea, pentru a crește eficiența calculelor, se recomandă ca variabilele rămase să fie mai întâi sortate topologic, astfel încât părinții să apară înaintea copiilor. În acest caz, se vor putea descompune mai ușor sumele, scoțând în față factorii care nu depind de o anumită variabilă.

$$\begin{aligned}
P(G_D|O_D, X_D) &= \\
\alpha \cdot \sum_{a \in \{A_D, A_N\}} \sum_{f \in \{F_D, F_N\}} P(G_D, a, f, O_D, X_D) &= \\
\alpha \cdot \sum_a \sum_f P(G_D) \cdot P(a) \cdot P(f|G_D, a) \cdot P(O_D|f) \cdot P(X_D|f) &= \\
\alpha \cdot P(G_D) \cdot \sum_a P(a) \cdot \sum_f P(f|G_D, a) \cdot P(O_D|f) \cdot P(X_D|f) &= \\
\alpha \cdot P(G_D) \cdot \sum_a P(a) \cdot [P(F_D|G_D, a) \cdot P(O_D|F_D) \cdot P(X_D|F_D) + \\
P(F_N|G_D, a) \cdot P(O_D|F_N) \cdot P(X_D|F_N)] &= \\
\alpha \cdot P(G_D) \cdot \{P(A_D) \cdot [P(F_D|G_D, A_D) \cdot P(O_D|F_D) \cdot P(X_D|F_D) + \\
P(F_N|G_D, A_D) \cdot P(O_D|F_N) \cdot P(X_D|F_N)] + \\
P(A_N) \cdot [P(F_D|G_D, A_N) \cdot P(O_D|F_D) \cdot P(X_D|F_D) + \\
P(F_N|G_D, A_N) \cdot P(O_D|F_N) \cdot P(X_D|F_N)]\} &= \\
\alpha \cdot 0,1 \cdot \{0,05 \cdot [0,8 \cdot 0,6 \cdot 0,5 + 0,2 \cdot 0,2 \cdot 0,1] + \\
0,95 \cdot [0,7 \cdot 0,6 \cdot 0,5 + 0,3 \cdot 0,2 \cdot 0,1]\} &= \\
\alpha \cdot 0,02174.
\end{aligned}$$

În exemplul de mai sus, se observă că $P(a)$ nu depinde de f și prin urmare, suma corespunzătoare variabilei *Abces* a fost scoasă în fața sumei corespunzătoare variabilei *Febră*, evitându-se duplicarea unor calcule. Nodul *Abces*, neavând părinti, este în fața *Febrei* în sortarea topologică. Se remarcă variabila α care intervine în expresia probabilității. Vom explica sensul acesteia imediat, după ce vom considera și calculele pentru $P(G_N|O_D, X_D)$, în mod analog:

$$\begin{aligned}
P(G_N|O_D, X_D) &= \\
\alpha \cdot \sum_{a \in \{A_D, A_N\}} \sum_{f \in \{F_D, F_N\}} P(G_N, a, f, O_D, X_D) &= \\
\alpha \cdot \sum_a \sum_f P(G_N) \cdot P(a) \cdot P(f|G_N, a) \cdot P(O_D|f) \cdot P(X_D|f) &= \\
\alpha \cdot P(G_N) \cdot \sum_a P(a) \cdot \sum_f P(f|G_N, a) \cdot P(O_D|f) \cdot P(X_D|f) &= \\
\alpha \cdot P(G_N) \cdot \{P(A_D) \cdot [P(F_D|G_N, A_D) \cdot P(O_D|F_D) \cdot P(X_D|F_D) + \\
P(F_N|G_N, A_D) \cdot P(O_D|F_N) \cdot P(X_D|F_N)] + \\
P(A_N) \cdot [P(F_D|G_N, A_N) \cdot P(O_D|F_D) \cdot P(X_D|F_D) + \\
P(F_N|G_N, A_N) \cdot P(O_D|F_N) \cdot P(X_D|F_N)]\} &= \\
\alpha \cdot 0,9 \cdot \{0,05 \cdot [0,25 \cdot 0,6 \cdot 0,5 + 0,75 \cdot 0,2 \cdot 0,1] + \\
0,95 \cdot [0,05 \cdot 0,6 \cdot 0,5 + 0,95 \cdot 0,2 \cdot 0,1]\} &= \\
\alpha \cdot 0,03312.
\end{aligned}$$

Rolul coeficientului α este de a asigura faptul că $P(G_D|O_D, X_D) + P(G_N|O_D, X_D) = 1$, deoarece *Da* și *Nu* sunt singurele valori posibile pentru *Gripă*. Având în vedere că $P(G_D|O_D, X_D) = \alpha \cdot 0,02174$ și $P(G_N|O_D, X_D) = \alpha \cdot 0,03312$, există $\alpha = 1/(0,02174 + 0,03312) = 18,23$, astfel încât suma celor două probabilități să fie 1. În consecință, rezultatul interogării este:

$$P(G_D|O_D, X_D) = 0,39628 \approx 40\%,$$

$$P(G_N|O_D, X_D) = 0,60372 \approx 60\%.$$

5. Aplicații

1. Fie rețeaua bayesiană din figura 2, cu probabilitățile din tabelul 1. Folosind algoritmul de inferență prin enumerare, răspundeți, *pe hârtie*, la întrebarea: „Care este probabilitatea ca o persoană să fie obosită dacă nu are gripă, nu are abces și nu are anorexie?”

2. Desenați în programul *Belief and Decision Network Tool* (*bayes.jar*, inclus în arhiva laboratorului) aceeași rețea bayesiană (figura 2, tabelul 1). Răspundeți la întrebarea de la punctul 1 cu ajutorul programului și comparați rezultatele.

Indicații. Desenarea se face în tab-ul *Create*. Interogările se introduc în tab-ul *Solve: Make observation* pentru setarea evidențelor și *Query* pentru setarea variabilei de interogare.

3. Tot cu ajutorul programului, răspundeți la următoarele întrebări:

a) Care este probabilitatea ca o persoană să aibă febră, dacă are gripă și abces? Cum influențează variabilele *Oboseală* și *Anorexie* aceste probabilități?

b) Care sunt probabilitățile marginale ale nodurilor *Febră*, *Oboseală* și *Anorexie* (când în rețea nu sunt noduri de evidență)?

c) Care este probabilitatea nodului *Oboseală* dacă *Gripă* are valoarea *Da*?

d) Care este probabilitatea nodului *Oboseală* dacă *Gripă* are valoarea *Da* și *Febră* are valoarea *Nu*? În acest caz, care sunt variabilele irelevante pentru interogare?

4. Fie situația de trafic din figura 3. Strada incidentă în intersecție, cu sens unic, are 4 benzi de circulație (B_1, B_2, B_3, B_4). Până la intersecție sunt 4 sectoare de drum (S_1, S_2, S_3, S_4). Din fiecare sector, o mașină poate merge înainte, cu probabilitatea de 50% sau la dreapta, respectiv stânga, cu probabilitățile de 25%. De pe benzile laterale nu se poate ieși în afara drumului, prin urmare probabilitatea de a merge înainte este 75%. La capătul străzii, mașinile pot merge pe unul din cele 3 drumuri (*Stânga*, *Înainte*, *Dreapta*).

Modelați această situație cu ajutorul unei rețele bayesiene în programul *bayes.jar*.

Indicație. Fiecare sector de drum poate fi modelat ca o variabilă cu 4 valori posibile, corespunzătoare celor 4 benzi. Cele 3 drumuri de după intersecție pot fi modelate ori ca o variabilă cu 3 valori posibile, ori ca 3 variabile distințe, de data aceasta cu câte 2 valori, *Adevărat* sau *Fals*.

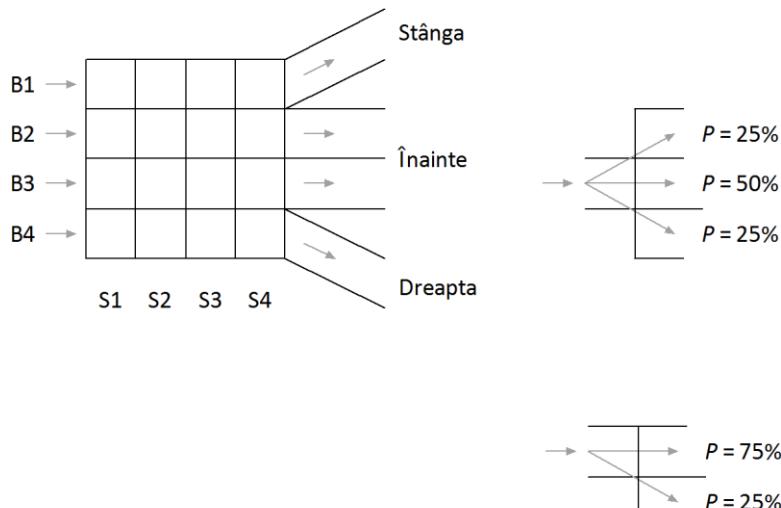


Figura 3. Model de intersecție

Cu ajutorul programului, răspundeți la următoarele întrebări:

- Dacă o mașină este pe segmentul S_1 , banda B_1 , care sunt probabilitățile ca după intersecție să meargă la *Stânga*, *Înainte* sau la *Dreapta*? Pentru direcția *Înainte*, nu ne interesează pe care din cele două benzi va merge mașina, contează doar direcția.
- Să presupunem că o mașină merge pe segmentul S_1 , banda B_1 , apoi pe segmentul S_2 , banda B_4 . Este posibil? Răspunsul se găsește calculând probabilitatea evidenței (în program, $P(e)$ Query).
- Dacă o mașină este pe segmentul S_1 , banda B_1 și o ia la *Dreapta* după intersecție, care sunt probabilitățile poziției sale pe sectoarele de drum S_3 și S_4 ?
- Dacă mașina a luat-o la *Stânga* în intersecție, care sunt probabilitățile poziției sale anterioare pe sectoarele de drum incidente: S_1 , S_2 , S_3 și S_4 ?

Rețele neuronale: regiuni de decizie

1. Obiective

Obiectivele acestui laborator sunt următoarele:

- descrierea caracteristicilor rețelelor neuronale de tip perceptron, cu unul sau mai multe straturi;
- prezentarea unor funcții de activare pentru neuroni;
- descoperirea empirică a parametrilor unor rețele neuronale simple care aproximează funcții logice elementare și reprezentarea regiunilor de decizie ale acestora.

2. Caracteristici ale rețelelor neuronale artificiale

Preocuparea pentru rețelele neuronale artificiale, denumite în mod curent „rețele neuronale”, a fost motivată de recunoașterea faptului că modul în care calculează creierul ființelor via este complet diferit de cel al calculatoarelor numerice convenționale. Spre deosebire de mașinile von Neumann, unde există o unitate de procesare care execută instrucțiunile stocate în memorie în mod serial, numai o instrucțiune la un moment dat, rețelele neuronale utilizează în mod masiv paralelismul. Fiind modele simplificate ale creierului uman, ele dețin capacitatea de a învăța, spre deosebire de calculatoarele convenționale, care rămân totuși mai eficiente pentru sarcinile bazate pe operații aritmetice precise și rapide. Rețelele neuronale nu dispun de unități de procesare puternice, dimpotrivă, acestea sunt caracterizate printr-o simplitate extremă, însă interacțiunile lor pe ansamblu produc rezultate complexe datorită numărului mare de conexiuni.

O rețea neuronală este un procesor masiv paralel, distribuit, care are o tendință naturală de a înmagazina cunoștințe experimentale și de a le face disponibile pentru utilizare. Ea se aseamănă cu creierul în două privințe:

- Cunoștințele sunt căpătate de rețea printr-un proces de învățare;
- Cunoștințele sunt depozitate nu în unitățile de procesare (neuroni), ci în conexiunile interneuronale, cunoscute drept ponderi sinaptice.

Procedura folosită pentru a executa procesul de învățare se numește *algoritm de învățare*, funcția căruia este de a modifica ponderile sinaptice ale rețelei într-un stil sistematic pentru a atinge obiectivul dorit de proiectare. Printre numeroasele proprietăți interesante ale unei rețele neuronale, cea mai semnificativă este abilitatea acesteia de a învăța prin intermediul mediului înconjurător, și prin aceasta să-și îmbunătățească performanțele; creșterea performanțelor are loc în timp și conform cu unele reguli prestabilite. O rețea neuronală își învăță mediul printr-un proces iterativ de ajustări

aplicate conexiunilor și pragurilor sale sinaptice. În mod ideal, rețeaua devine mai intelligentă după fiecare iterație a procesului de învățare.

În contextul rețelelor neuronale vom defini astfel *învățarea*: un proces prin care parametrii variabili ai unei rețele neuronale se adaptează printr-un proces continuu de stimulare din partea mediului în care este inclusă. Tipul de învățare este determinat de modul în care au loc schimbările parametrilor.

Definiția învățării implică următoarea secvență de evenimente:

- Rețeaua neuronală este stimulată de un mediu;
- Rețeaua neuronală suferă schimbări datorită acestor stimulați;
- Rețeaua neuronală răspunde în mod diferit mediului datorită schimbărilor care au apărut în structura sa internă.

Rețelele neuronale se caracterizează prin trei elemente: modelul neuronului, arhitectura rețelei și algoritmul de antrenare folosit.

3. Rețele cu un singur strat

Începutul rețelelor neuronale artificiale este legat de problema clasificării unor obiecte definite de o serie de atribuții. Cel mai simplu model era funcția $\$I$ logic între anumite atribuții (prezente sau absente), care să determine o anumită clasă. Totuși, unele clase pot avea atribuții comune, iar unele valori, în cazul în care provin dintr-un mecanism perceptual, pot fi afectate de zgomot. Soluția s-a bazat pe faptul de bun simț că unele atribuții sunt mai importante decât celelalte pentru determinarea unei anumite clase. O clasă era determinată dacă sumarea valorilor ponderate depășea un anumit prag, în bună concordanță cu legea biologică „totul sau nimic” (dacă un impuls nu depășește un prag minim, el nu produce nici un răspuns).

Frank Rosenblatt a propus un astfel de model, care rămâne până în prezent fundamentalul structural pentru majoritatea rețelelor neuronale.

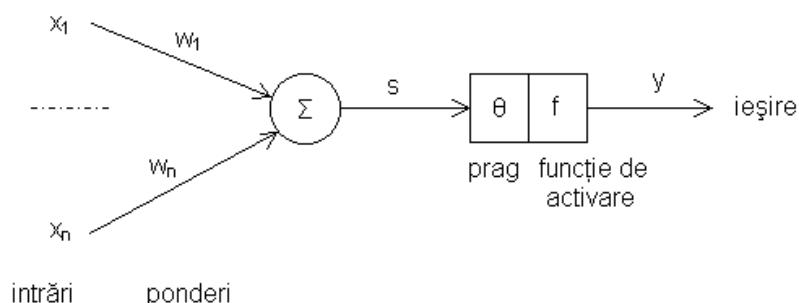


Figura 1. Modelul unui neuron

Fiecărei conexiuni îi corespunde o valoare reală, numită pondere sinaptică, care determină efectul intrării respective asupra nivelului de activare a neuronului. Suma ponderată a intrărilor poartă denumirea de *intrare netă* (engl. “net input”). În figura 1, x_i reprezintă intrările, w_i ponderile sinaptice, f o funcție de activare, θ valoarea prag iar y ieșirea, care se calculează după formula:

$$y = f\left(\sum_{i=1}^n w_i x_i - \theta\right). \quad (1)$$

Rosenblatt a imaginat și un algoritm de învățare pentru așa-numitul *perceptron* (figura 1). Ideea principală este de a face mici ajustări ale ponderilor pentru a reduce diferența dintre ieșirea reală a perceptronului și ieșirea dorită. Ponderile inițiale sunt inițializate aleatoriu (în general în intervalul [-0.5, 0.5]) și apoi actualizate treptat astfel încât ieșirea să se apropie de valorile dorite. Exemplele de antrenare sunt prezentate succesiv, în orice ordine. Dacă în pasul (epoca) p ieșirea reală este y^p iar ieșirea dorită este y_d^p , atunci eroarea este:

$$e^p = y_d^p - y^p.$$

Dacă eroarea e este pozitivă, trebuie să mărim y ; dacă este negativă, y trebuie micșorat. Având în vedere că fiecare intrare are contribuția $x_i \cdot w_i$, atunci dacă x_i este pozitivă, o creștere a ponderii w_i va avea ca efect o creștere a ieșirii perceptronului. Invers, dacă x_i este negativă, creșterea ponderii w_i va determina scăderea ieșirii y . De aici poate fi stabilită regula de învățare a perceptronului:

$$w_i^{p+1} = w_i^p + \alpha \cdot x_i \cdot e^p,$$

unde $\alpha \in (0, 1)$ este numită *rata de învățare*.

Cu ajutorul perceptronului pot fi învățate de exemplu funcții binare elementare, precum **ȘI**, **SAU** etc. (figura 2). Pe abscisă și ordonată sunt reprezentate valorile celor două intrări, iar culoarea cercurilor reprezintă rezultatul operației (alb = 0, negru = 1). Perceptronul împarte planul în două regiuni de decizie (datorită pragului funcției de activare). În cazul n -dimensional, spațiul soluțiilor va fi divizat tot în două regiuni de un hiperplan. Acestea sunt probleme *liniar separabile*. Aici poate fi observată și utilitatea pragului: în lipsa acestuia, hiperplanul separator ar trece întotdeauna prin origine, ceea ce nu este de dorit în orice situație.

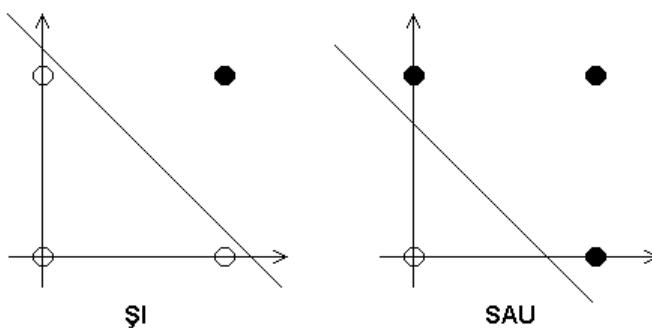


Figura 2. Probleme liniar separabile: ȘI, SAU

Algoritmul de antrenare garantează clasificarea corectă a două clase pe baza setului de antrenare, cu condiția ca acele clase să fie liniar separabile.

Algoritmul de antrenare al perceptronului este prezentat mai jos:

```
se inițializează toate ponderile  $w_i$  cu 0 sau cu valori aleatorii din intervalul [-0.5, 0.5]
se inițializează rata de învățare alfa cu o valoare din intervalul (0, 1], de exemplu 0.1
se inițializează numărul maxim de epoci P, de exemplu 100
```

```
p = 0 // numărul epocii curente (pasul curent)
erori = true // un flag care indică existența erorilor de antrenare
```

```
repetă cât timp p < P și erori == true
{
    erori = false
    pentru fiecare vector de antrenare  $x_i$  cu i = 1..N
    {
         $y_i = F(\sum(x_{ij} * w_j))$  cu j = 1..n+1
        dacă ( $y_i \neq y_{di}$ )
        {
            e =  $y_{di} - y_i$ 
            erori = true
            pentru fiecare intrare j = 1..n+1
             $w_j = w_j + alfa * x_{ij} * e$ 
        }
    }
    p = p + 1
}
```

Foarte multe probleme sunt însă neseparabile liniar. De exemplu funcția XOR (figura 3) nu poate fi învățată de un perceptron simplu. Minsky și Papert au demonstrat limitările serioase ale rețelelor de tip perceptron în aceste situații. De asemenea, ei au studiat posibilitatea utilizării perceptronilor pentru calculul predicatelor, demonstrându-le limitele în comparație cu mașina Turing. Concluzia cărții lor a fost că și rețelele multi-strat vor prezenta aceleași impiedimente, ceea ce a determinat practic pierderea interesului pentru cercetările în domeniul rețelelor neuronale pentru aproape 20 de ani, când a fost găsită o soluție fezabilă pentru problemele neseparabile liniar. Trebuie totuși menționat faptul că funcția XOR multidimensională (suma modulo 2 a argumentelor) continuă să fie o funcție greu de învățat și pentru rețelele neuronale actuale.

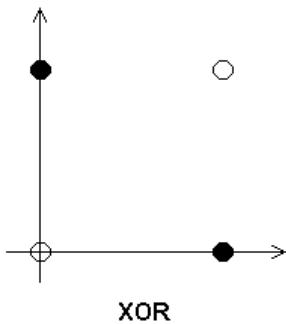


Figura 3. Problemă liniar neseparabilă: XOR

4. Perceptronul multi-strat

Încercările de rezolvare a problemelor neseparabile liniar au condus la diverse variante privind numărul de straturi de neuroni și funcțiile de activare utilizate. *Perceptronul multi-strat* (engl. “multilayer perceptron”, MLP) este tipul de rețea neuronală cel mai cunoscut și mai des folosit. De cele mai multe ori, semnalele se transmit în interiorul rețelei într-o singură direcție: de la intrare spre ieșire. Nu există bucle, ieșirea fiecărui neuron neafectând neuronul respectiv. Această arhitectură se numește *cu propagare înainte* (engl. “feed-forward”) (figura 4). Straturile care nu sunt conectate direct la mediu se numesc *ascunse*. Există în literatura de specialitate o controversă privind considerarea primului strat (de intrare) ca strat propriu-zis în rețea, de vreme ce singura sa funcție este transmiterea semnalelor de intrare spre straturile superioare, fără a face vreo prelucrare asupra intrărilor. În cele ce urmează, vom număra numai straturile formate din neuroni propriu-zisi, însă vom spune că intrările sunt grupate în stratul de intrare.

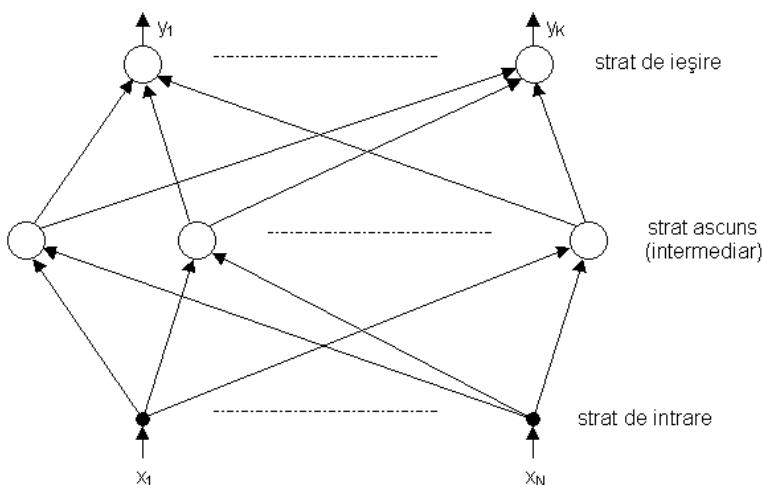


Figura 4. Rețea neuronală *feed-forward* multi-strat

Există și rețele *recurente* (cu *feed-back*), în care impulsurile se pot transmite în ambele direcții, datorită unor conexiuni de reacție în rețea. Aceste tipuri de rețele sunt foarte puternice și pot fi extrem de complicate. Sunt dinamice, starea lor schimbându-se permanent, până când rețeaua ajunge la un punct de echilibru iar căutarea unui nou echilibru are loc la fiecare schimbare a intrării.

Introducerea mai multor straturi a fost determinată de necesitatea creșterii complexității regiunilor de decizie. După cum am arătat în paragraful anterior, un perceptron cu un singur strat și o ieșire generează regiuni de decizie de forma unor semiplane. Adăugând încă un strat, fiecare neuron se comportă ca un perceptron standard asupra ieșirii neuronilor din stratul anterior și astfel ieșirea rețelei poate approxima regiuni de decizie convexe, rezultate din intersecția semiplanelor generate de neuroni. La rândul său, un perceptron cu trei straturi poate genera zone de decizie arbitrară (figura 5).

Din punct de vedere al funcției de activare a neuronilor, rețelele multi-strat nu asigură o creștere a puterii de calcul în raport cu rețelele cu un singur strat dacă funcțiile de activare sunt liniare, deoarece o funcție liniară de funcții liniare este tot o funcție liniară. Puterea perceptronului multi-strat provine tocmai din funcțiile de activare neliniare. Aproape orice funcție neliniară poate fi

folosită în acest scop, cu excepția funcțiilor polinomiale. În prezent, funcțiile cele mai des utilizate în prezent sunt *sigmoida unipolară* (sau logistică), afișată în figura 6:

$$f(s) = \frac{1}{1 + e^{-s}}$$

și *sigmoida bipolară* (tangenta hiperbolică), afișată în figura 7:

$$f(s) = \frac{1 - e^{-2s}}{1 + e^{-2s}}.$$

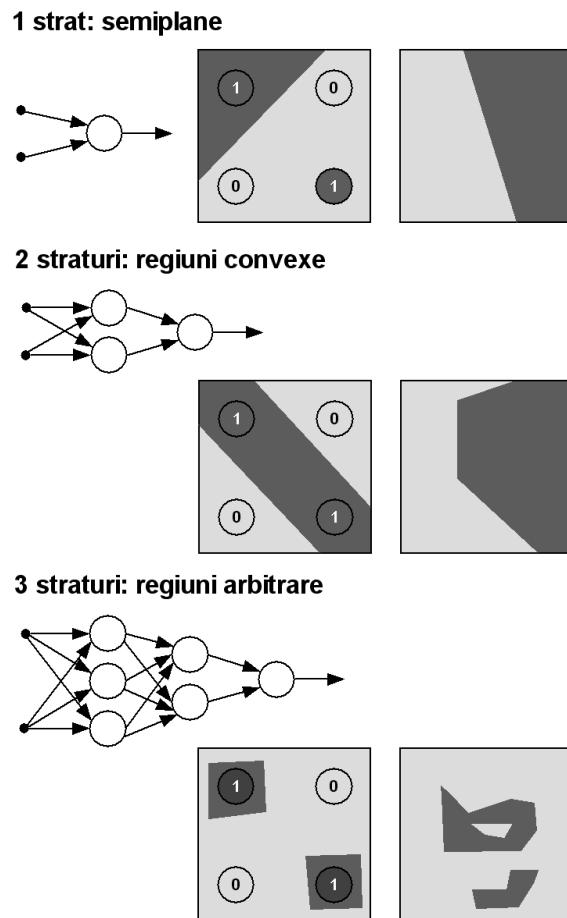


Figura 5. Regiunile de decizie ale perceptronilor multi-strat

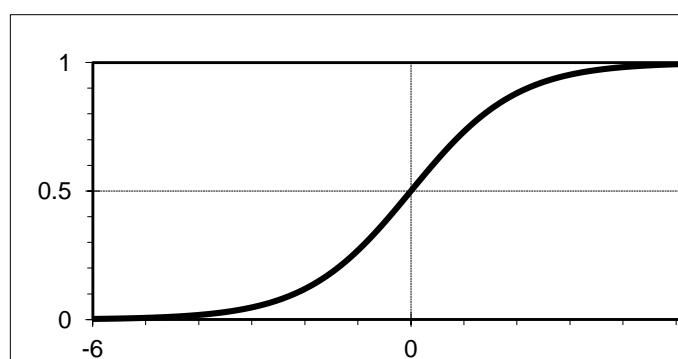


Figura 6. Funcția de activare sigmoidă unipolară

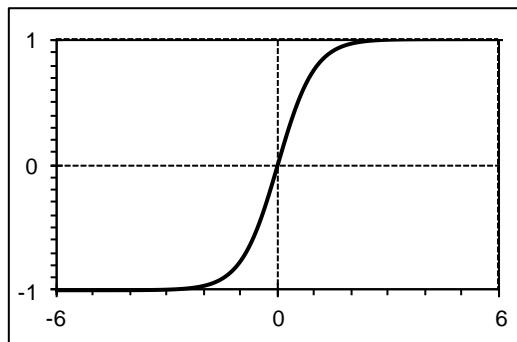


Figura 7. Funcția de activare sigmoidă bipolară

Se poate constata că funcțiile sigmoide se comportă aproximativ liniar pentru valori absolute mici ale argumentului și se saturează, preluând oarecum rolul de prag, pentru valori absolute mari ale argumentului.

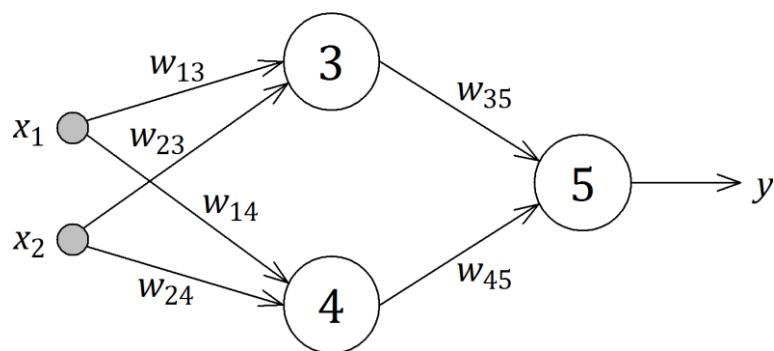
O aproximare mai simplă a sigmoidei este funcția semiliniară, definită astfel:

$$f(s) = \begin{cases} -1, & s \leq -1 \\ s, & s \in (-1, 1) \\ 1, & s \geq 1 \end{cases}$$

O rețea, posibil infinită, cu un singur strat ascuns este capabilă că aproximeze orice funcție continuă. Astfel se justifică proprietatea perceptronului multi-strat de *aproximator universal*.

5. Aplicații

Se dă un prototip de aplicație pentru implementarea unui perceptron cu un singur strat și a unui perceptron cu un strat ascuns, având configurația din figura următoare:



Prototipul include interfața grafică cu utilizatorul și desenarea regiunilor de decizie.

Întrucât aici se presupune că ieșirile neuronilor aparțin intervalului $[0, 1]$, se folosesc următoarele expresii pentru funcțiile de activare:

- funcția prag:

$$f(s) = \begin{cases} 0, & s < 0 \\ 1, & s \geq 0 \end{cases}$$

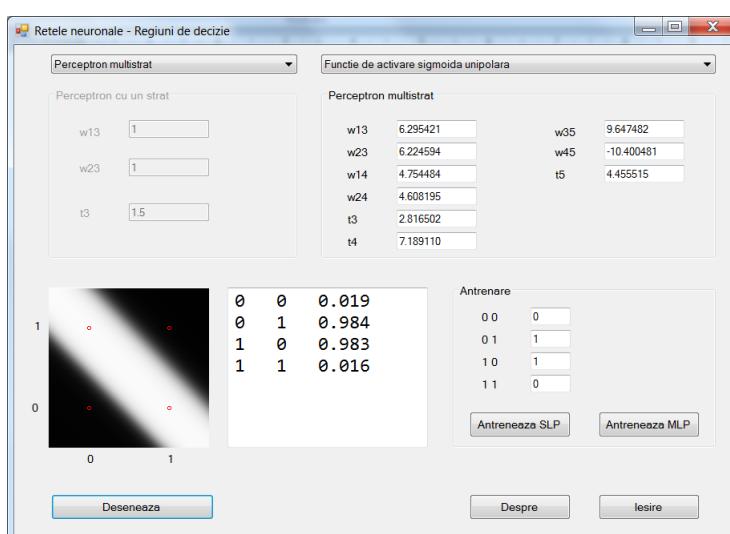
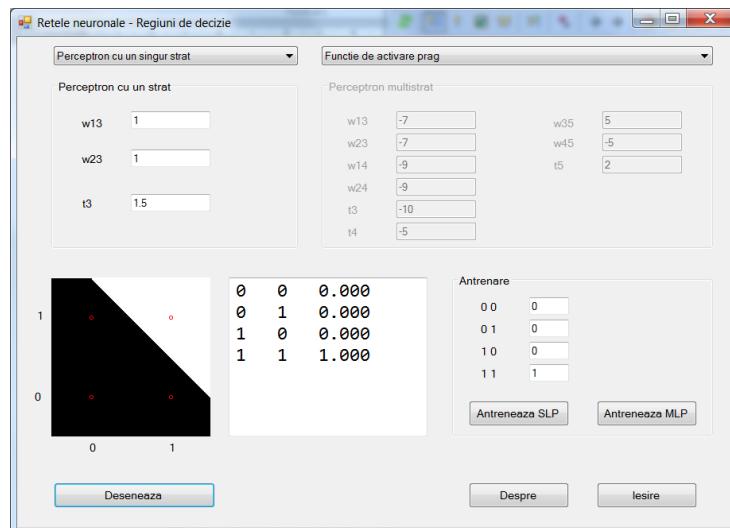
- funcția semiliniară:

$$f(s) = \begin{cases} 0, & s \leq 0 \\ s, & s \in (0, 1) \\ 1, & s \geq 1 \end{cases}$$

- funcția sigmoidă unipolară:

$$f(s) = \frac{1}{1 + e^{-s}}$$

La fel de bine, se pot proiecta rețele în care ieșirile aparțin intervalului [-1, 1].



Se dorește determinarea ponderilor conexiunilor dintre neuroni și a valorilor prag astfel încât rețeaua să aproximeze câteva funcții binare elementare. De exemplu:

Funcția de activare	Funcția de aproximat	w_{13}	w_{23}	w_{14}	w_{24}	w_{35}	w_{45}	θ_3	θ_4	θ_5
prag	nand	1	0	0	1	-0.5	-0.5	0.5	0.5	-0.5
semiliniară	or	1	1	1	1	0.5	0.5	0	0	0
sigmoidă	and	0	-2	-6	-6	0	-5	0	-9	-3
sigmoidă	or	-5	-5	6	6	-3	3	-3	3	0
sigmoidă	xor	-7	-7	-9	-9	5	-5	-10	-5	2

Scopul laboratorului este de a încerca determinarea prin încercări a parametrilor pentru un perceptron cu un singur strat, pentru probleme liniar separabile și apoi determinarea automată a acestora cu ajutorul algoritmului de antrenare al perceptronului. Pentru un perceptron cu un singur strat, valorile ponderilor determină rotația dreptei de separare a celor două clase, iar pragul controlează apropierea acestei drepte de origine.

Pentru perceptronul multi-strat și probleme neseparabile liniar, se poate folosi algoritmul *backpropagation* (inclus într-un *dll*). Acest algoritm va trebui implementat în laboratorul 13.

5.1. Completăți metodele care calculează cele trei funcții de activare și metodele care calculează ieșirile perceptronilor: StepActivation, SemiliniarActivation, SigmoidActivation, respectiv SLP și MLP.

5.2. Completăți metoda corespunzătoare antrenării perceptronului cu un singur strat: buttonTrainSLP_Click.

5.3. Determinați parametrii corespunzători pentru aproximarea funcției SAU, folosind funcția de activare prag pentru un perceptron cu un singur strat și pentru perceptronul multi-strat din figura de mai sus.

5.4. Aproximați funcția NON-XOR (XNOR) cu perceptronul multi-strat și funcția de activare sigmoidă.

5.5. Aproximați funcția NON-ȘI (NAND) folosind funcția de activare semiliniară și cele două tipuri de perceptron.

Algoritmul de retro-propagare (*backpropagation*)

1. Obiectiv

Obiectivul acestui laborator este prezentarea algoritmului de retro-propagare (backpropagation), utilizat pentru antrenarea rețelelor neuronale de tip perceptron multi-strat.

2. Descrierea algoritmului

Rețelele neuronale au capacitatea de a învăța, însă modalitatea concretă în care se realizează acest proces este dată de algoritm folosit pentru antrenare. O rețea se consideră antrenată dacă aplicarea unui vector de intrare conduce la obținerea unei ieșiri dorite, sau foarte apropiate de aceasta.

Antrenarea constă în aplicarea secvențială a diferenții vectori de intrare și ajustarea ponderilor din rețea cu ajutorul unui algoritm de antrenare. În acest timp, ponderile conexiunilor converg gradual spre anumite valori pentru care fiecare vector de intrare produce vectorul de ieșire dorit. După aplicarea unei intrări, se compară ieșirea calculată cu ieșirea dorită, după care diferența este folosită pentru modificarea ponderilor cu scopul minimizării erorii la un nivel acceptabil.

Algoritmul backpropagation este cel mai cunoscut și utilizat algoritm de antrenare pentru rețele neuronale de tip perceptron multi-strat. Numit și *algoritmul delta generalizat*, el se bazează pe minimizarea diferenței dintre ieșirea dorită și ieșirea reală, prin metoda gradientului descedent. Se definește *gradientul* unei funcții $F(x,y,z,\dots)$ drept:

$$\vec{\nabla} F = \frac{\partial F}{\partial x} \vec{i} + \frac{\partial F}{\partial y} \vec{j} + \frac{\partial F}{\partial z} \vec{k} + \dots$$

Gradientul ne spune cum variază funcția în diferite direcții. Ideea algoritmului este găsirea minimului funcției de eroare în raport cu ponderile conexiunilor. Eroarea este dată de diferența dintre ieșirea dorită și ieșirea efectivă a rețelei. Cea mai utilizată funcție de eroare este eroarea medie pătratică (mean squared error, MSE). Dacă avem în multimea de antrenare K vectori iar rețeaua are N ieșiri, atunci:

$$MSE = \frac{1}{K \cdot N} \cdot \sum_k \sum_n e_{kn}^2 = \frac{1}{K \cdot N} \cdot \sum_k \sum_n (y_{kn}^d - y_{kn})^2 ,$$

unde y_{kn}^d este valoarea dorită la ieșirea n a rețelei pentru vectorul k , iar y_{kn} este ieșirea efectivă a rețelei.

Algoritmul pentru un perceptron multistrat cu un singur strat ascuns este următorul.

Pasul 1: Inițializarea

Toate ponderile și pragurile rețelei sunt inițializate cu valori aleatorii *nenule*, distribuite uniform într-un mic interval, de exemplu $(-0.1, 0.1)$.

Dacă valorile acestea sunt 0, gradienții care vor fi calculați pe parcursul antrenării vor fi tot 0 (dacă nu există o legătură directă între intrare și ieșire) și rețeaua nu va învăța. Este chiar indicată încercarea mai multor antrenări, cu ponderi inițiale diferite, pentru găsirea celei mai bune valori pentru funcția de cost (minimul erorii). Dimpotrivă, dacă valorile inițiale sunt mari, ele tind să satureze unitățile respective. În acest caz, derivata funcției sigmoide este foarte mică. Ea acționează ca un factor de multiplicare în timpul învățării și deci unitățile saturate vor fi aproape blocate, ceea ce face învățarea foarte lentă.

Pasul 2: O nouă epocă de antrenare

O epocă reprezintă prezentarea tuturor exemplelor din mulțimea de antrenare. În majoritatea cazurilor, antrenarea rețelei presupune mai multe epoci de antrenare.

Pentru ca antrenarea să nu fie influențată de ordinea de prezentare a vectorilor de antrenare, ponderile vor fi ajustate numai după ce toți vectorii sunt aplicați rețelei. În acest scop, corecțiile ponderilor trebuie memorate și ajustate după prezentarea fiecărui vector din mulțimea de antrenare. La sfârșitul unei epoci de antrenare, se modifică ponderile o singură dată. Există și varianta online, mai simplă, în care ponderile sunt actualizate direct. În acest caz, poate conta ordinea în care sunt prezențați rețelei vectorii de antrenare.

Se inițializează corecțiile ponderilor și eroarea curentă cu 0: $\Delta w_{ij} = 0$ și $E = 0$.

Pasul 3: Propagarea semnalului înainte

3.1. La intrările rețelei se aplică un vector din mulțimea de antrenare.

3.2. Se calculează ieșirile neuronilor din stratul ascuns:

$$y_j = f\left(\sum_{i=1}^n x_i \cdot w_{ij} - \theta_j\right),$$

unde n este numărul de intrări ale neuronului j din stratul ascuns, iar f este funcția de activare sigmoidă.

Pentru simplitate, pragul este considerat ca fiind ponderea unei conexiuni suplimentare a cărei intrare este întotdeauna egală cu 1. Expresia de mai sus devine acum:

$$y_j = f\left(\sum_{i=1}^{n+1} x_i \cdot w_{ij}\right).$$

3.3. Se calculează ieșirile reale ale rețelei:

$$y_k = f\left(\sum_{j=1}^{m+1} y_j \cdot w_{jk}\right),$$

unde m este numărul de intrări ale neuronului k din stratul de ieșire.

3.4. Se actualizează eroarea medie pătratică pe epocă:

$$MSE = \frac{1}{K \cdot N} \cdot \sum_k \sum_n e_{kn}^2 = \frac{1}{K \cdot N} \cdot \sum_k \sum_n (y_{kn}^d - y_{kn})^2 .$$

Pasul 4: Propagarea erorilor înapoi și ajustarea ponderilor

4.1. Se calculează gradienții erorilor pentru neuronii din stratul de ieșire:

$$\delta_k = f' \cdot e_k ,$$

unde f' este derivata funcției de activare iar eroarea $e_k = y_k^d - y_k$.

Dacă folosim sigmoida unipolară, derivata acesteia este:

$$f'(x) = \frac{e^{-x}}{(1 + e^{-x})^2} = f(x) \cdot (1 - f(x)).$$

Vom avea:

$$\delta_k = y_k \cdot (1 - y_k) \cdot e_k .$$

4.2. Se actualizează corecțiile ponderilor dintre stratul ascuns și stratul de ieșire:

$$\Delta w_{jk} = \Delta w_{jk} + \alpha \cdot y_j \cdot \delta_k ,$$

unde α este rata de învățare.

4.3. Se calculează gradienții erorilor pentru neuronii din stratul ascuns:

$$\delta_j = y_j \cdot (1 - y_j) \cdot \sum_{k=1}^l \delta_k \cdot w_{jk} ,$$

unde l este numărul de ieșiri ale rețelei.

4.4. Se actualizează corecțiile ponderilor dintre stratul de intrare și stratul ascuns:

$$\Delta w_{ij} = \Delta w_{ij} + \alpha \cdot x_i \cdot \delta_j.$$

Pasul 5. O nouă iterare

Dacă mai sunt vectori de test în epoca de antrenare curentă, se trece la pasul 3.

Dacă nu, se actualizează ponderile tuturor conexiunilor pe baza corecțiilor ponderilor:

$$w_{ij} = w_{ij} + \Delta w_{ij}.$$

Dacă s-a încheiat o epocă, se testează dacă s-a îndeplinit criteriul de terminare ($E < E_{max}$ sau atingerea unui număr maxim de epoci de antrenare). Dacă nu, se trece la pasul 2. Dacă da, algoritmul se termină.

3. Considerente practice

În implementarea algoritmului, apar o serie de probleme practice, legate în special de alegerea parametrilor antrenării și a configurației rețelei.

În primul rând, o rată de învățare mică determină o convergență lentă a algoritmului, pe când una prea mare poate cauza eșecul (algoritmul va „sări” peste soluție). Pentru probleme relativ simple, o rată de învățare $\alpha = 0.5$ este acceptabilă, însă în general se recomandă rate de învățare în jur de 0.1 – 0.2.

Este recomandată preprocesarea și postprocesarea datelor, astfel încât rețeaua să opereze cu valori scalate, de exemplu în intervalul [0.1, 0.9] pentru sigmoida unipolară.

O altă problemă caracteristică acestui mod de antrenare este dată de minimele locale. Într-un minim local, gradienții erorii devin 0 și învățarea nu mai continuă. O soluție este încercarea mai multor antrenări independente, cu ponderi inițializate diferite la început, ceea ce crește probabilitatea găsirii minimului global. Pentru probleme mari, acest lucru poate fi greu de realizat și atunci pot fi acceptate și minime locale, cu condiția ca erorile să fie suficient de mici. De asemenea, se pot încerca diferite configurații ale rețelei, cu un număr mai mare de neuroni în stratul ascuns sau cu mai multe straturi ascunse, care în general conduc la minime locale mai mici. Totuși, deși minimele locale sunt într-adevăr o problemă, în practică nu reprezintă dificultăți de nesoluționat.

O chestiune importantă este alegerea unei configurații cât mai bune pentru rețea din punct de vedere al numărului de neuroni în straturile ascunse. În multe situații, un singur strat ascuns este suficient. Nu există niște reguli precise de alegere a numărului de neuroni. În general, rețeaua poate fi văzută ca un sistem în care numărul vectorilor de test înmulțit cu numărul de ieșiri reprezintă numărul de ecuații iar numărul ponderilor reprezintă numărul necunoscutelor. Ecuațiile sistemului sunt în general neliniare și foarte complexe și deci este foarte dificilă rezolvarea lor exactă prin mijloace convenționale. Algoritmul de antrenare urmărește tocmai găsirea unor soluții aproximative care să minimizeze erorile.

O rețea neuronală trebuie să fie capabilă de generalizare. Dacă rețeaua aproximează bine mulțimea de antrenare, aceasta nu este o garanție că va găsi soluții la fel de bune și pentru datele din altă mulțime, cea de test. Generalizarea presupune existența în date a unor regularități, a unui model

care poate fi învățat. În analogie cu sistemele liniare clasice, aceasta ar însemna niște ecuații redundante. Astfel, dacă numărul de ponderi este mai mic decât numărul de vectori de test, pentru o aproximare corectă rețeaua trebuie să se bazeze pe modelele intrinseci din date, modele care se vor regăsi și în datele de test. O regulă euristică afirmă că numărul de ponderi trebuie să fie în jur sau sub o zecime din produsul dintre numărul de vectori de antrenare și numărul de ieșiri. În unele situații însă (de exemplu, dacă datele de antrenare sunt relativ puține), numărul de ponderi poate fi chiar jumătate din produs.

Pentru un perceptron multi-strat se consideră că numărul de neuroni dintr-un strat trebuie să fie suficient de mare pentru ca acest strat să furnizeze trei sau mai multe laturi pentru fiecare regiune convexă identificată de stratul următor. Deci numărul de neuroni dintr-un strat ar trebui să fie aproximativ de trei ori mai mare decât cel din stratul următor.

După cum am menționat, un număr insuficient de ponderi conduce la „sub-potrivire” (underfitting), în timp ce un număr prea mare de ponderi conduce la „supra-potrivire” (overfitting), fenomene prezентate în figura 1. Același lucru apare dacă numărul de epoci de antrenare este prea mic sau prea mare.

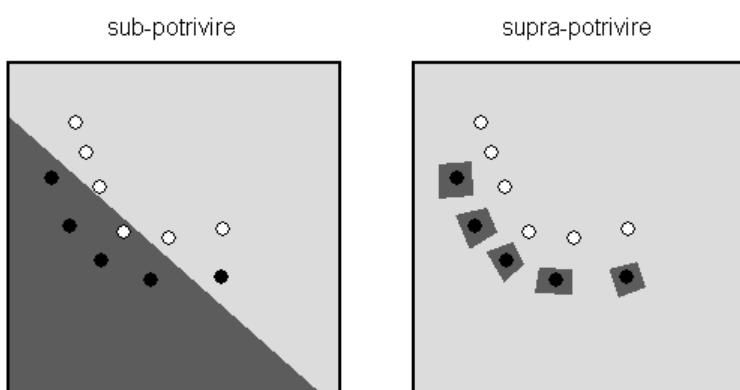


Figura 1. Capacitatea de generalizare

O metodă de rezolvare a acestei probleme este oprirea antrenării în momentul în care se atinge cea mai bună generalizare. Pentru o rețea suficient de mare, eroarea de antrenare scade în mod continuu pe măsură ce crește numărul epocilor de antrenare. Totuși, pentru date diferite de cele din mulțimea de antrenare (mulțimea de test), se constată că eroarea scade la început până la un punct în care începe din nou să crească. De aceea, oprirea antrenării trebuie făcută în momentul când eroarea pentru mulțimea de *test* este minimă.

Cea mai simplă modalitate de a estima capacitatea de generalizare este împărțirea mulțimii de date disponibile în 2/3 pentru antrenare și 1/3 pentru testare. Antrenarea se oprește atunci când începe să crească eroarea pentru mulțimea de validare, moment numit „punct de maximă generalizare”. În funcție de performanțele rețelei în acest moment, se pot încerca apoi diferite configurații, crescând sau micșorând numărul de neuroni din stratul (sau straturile) ascuns(e).

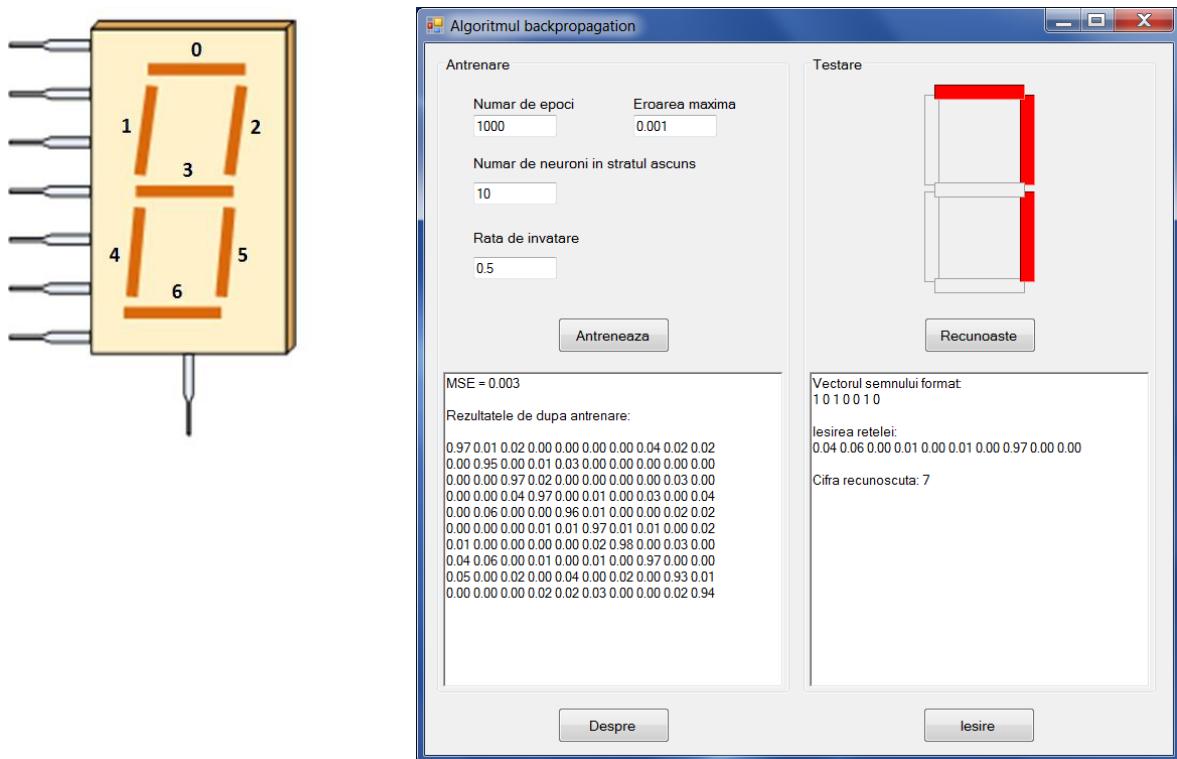
O metodă mai laborioasă, dar care este cea mai utilizată pentru evaluarea și compararea performanțelor modelelor de învățare, este *validarea încrucișată* (cross-validation), în care datele sunt împărțite în n grupuri (de obicei 10), $n - 1$ grupuri se folosesc pentru antrenare, iar al n -lea pentru testare. Procesul se repetă de n ori, schimbând grupul de test. În final, se face media performanțelor obținute pentru cele n grupuri de test.

Pentru explicații suplimentare, consultați cursul 11.

4. Aplicație

Implementați algoritmul backpropagation pentru o rețea neuronală de tip perceptron multistrat, cu un singur strat ascuns și cu un număr variabil de unități în stratul ascuns.

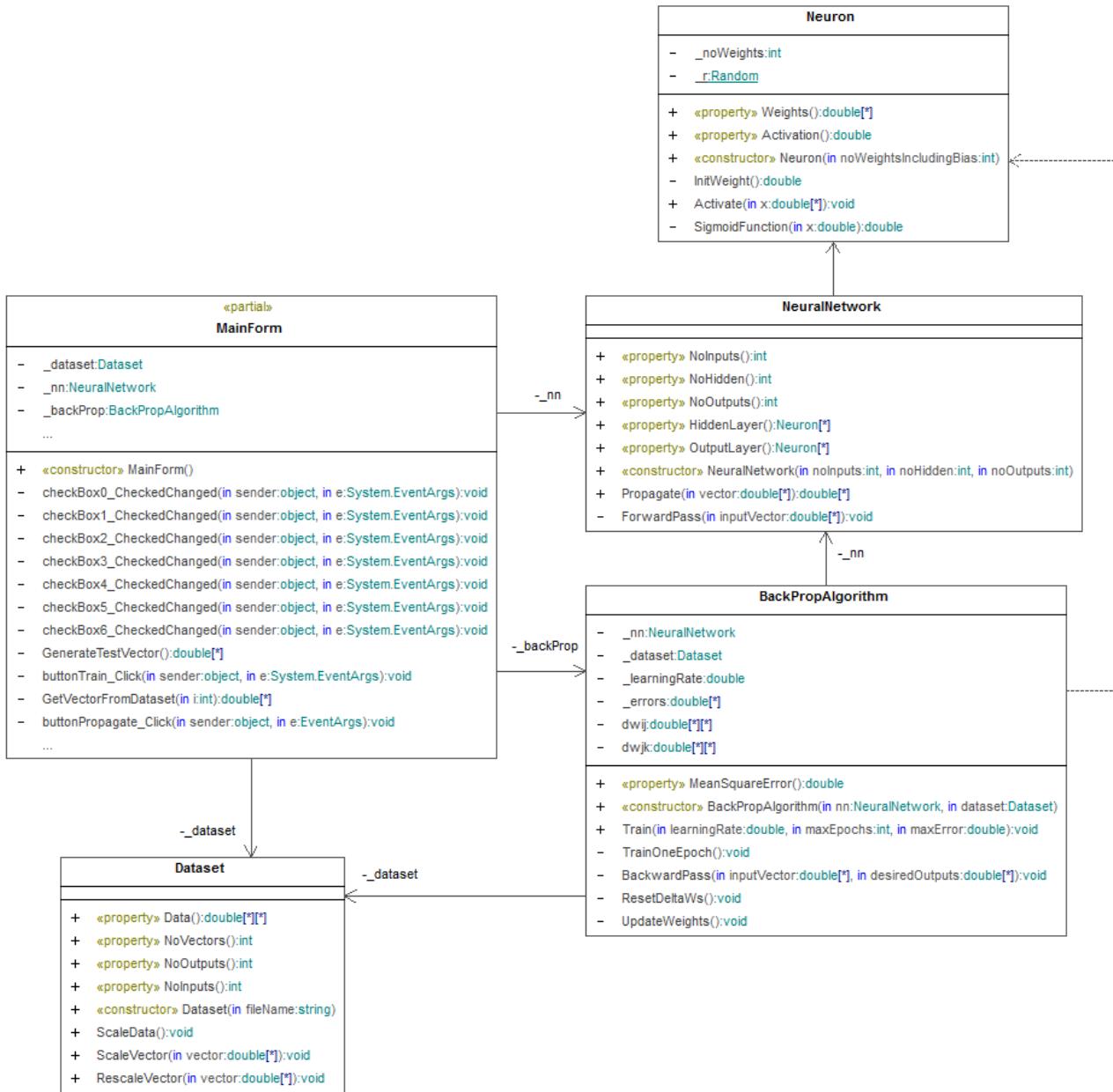
Se dă un prototip de aplicație care își propune recunoașterea cifrelor formate de utilizator pe un afișaj cu 7 led-uri.



Mulțimea de antrenare, citită din fișierul segments.data, conține 7 intrări (starea celor 7 led-uri), 10 ieșiri (1 pentru cifra formată, 0 în rest) și 10 vectori (cifrele 0-9):

7	10	10
1, 1, 1, 0, 1, 1, 1	1, 0, 0, 0, 0, 0, 0, 0, 0, 0	
0, 0, 1, 0, 0, 1, 0	0, 1, 0, 0, 0, 0, 0, 0, 0, 0	
1, 0, 1, 1, 1, 0, 1	0, 0, 1, 0, 0, 0, 0, 0, 0, 0	
1, 0, 1, 1, 0, 1, 1	0, 0, 0, 1, 0, 0, 0, 0, 0, 0	
0, 1, 1, 1, 0, 1, 0	0, 0, 0, 0, 1, 0, 0, 0, 0, 0	
1, 1, 0, 1, 0, 1, 1	0, 0, 0, 0, 0, 1, 0, 0, 0, 0	
1, 1, 0, 1, 1, 1, 1	0, 0, 0, 0, 0, 0, 1, 0, 0, 0	
1, 0, 1, 0, 0, 1, 0	0, 0, 0, 0, 0, 0, 0, 1, 0, 0	
1, 1, 1, 1, 1, 1, 1	0, 0, 0, 0, 0, 0, 0, 0, 1, 0	
1, 1, 1, 1, 0, 1, 1	0, 0, 0, 0, 0, 0, 0, 0, 0, 1	

Studiați structura aplicației. Diagrama UML de clase a programului complet este următoarea:



Implementați următoarele metode, corespunzătoare algoritmului backpropagation:

Metoda Train(double learningRate, int maxEpochs, double maxError) din clasa BackPropAlgorithm

Metoda de antrenare a rețelei. Antrenarea se realizează până la atingerea unui număr maxim de epoci (maxEpochs) sau până la atingerea unei erori acceptabile (maxError). Apelează metoda TrainOneEpoch.

Metoda TrainOneEpoch() din clasa BackPropAlgorithm

Reprezintă o epocă de antrenare pentru toți vectorii de antrenare. Calculează și eroarea medie pătratică (MSE) obținută la finalul epocii.

Metoda ForwardPass(double[] inputVector) din clasa NeuralNetwork

Pasul de propagare înainte al algoritmului.

Metoda BackwardPass(double[] inputVector, double[] desiredOutputs) din clasa BackPropAlgorithm

Pasul de propagare înapoi al algoritmului.