# Motor de Inferență predicativă prin raționament înainte

Realizat de:

Diaconu Rareș-George, 1409B
Huminiuc Simona, 1409B

# Cuprins

# Descrierea problemei considerate

În cadrul programului au fost alese spre rezolvare folosind algoritmul de inferență predicativă cu raționament înainte două probleme matematice.

1. Verificarea dacă o funcție este bijectivă

### 3. Bijecție

**Def.** O funcție f:A→B este funcție bijectivă dacă ea este și injectivă și surjectivă, ecuația $f(x) = y$ are o singură soluție x ϵ A.

**Obs.**

Funcția f este BIJECTIVĂ dacă pentru orice y ∈ B, ecuația $f(x) = y$ are O SINGURĂ SOLUȚIE.

2. Lema cleștelui

**Teoremă (criteriul „cleștelui").** Fie $(x_n)$, $(u_n)$, $(v_n)$ trei șiruri de numere reale care satisfac condițiile:

**1)** $u_n \leq x_n \leq v_n$, $\forall n \geq n_0$, $n_0$ fixat;

**2)** $\lim_{n\to\infty} u_n = \lim_{n\to\infty} v_n = a$.

Atunci șirul $(x_n)$ are limită și $\lim_{n\to\infty} x_n = a$.

# Aspecte teoretice privind algoritmul

Algoritmul de inferență predicativă implementat folosește principiul înlănțuirii înainte (forward chaining) în contextul logicii predicative de ordinul întâi. Baza algoritmului stă într-un set de date (fapte) cunoscute, căruia îi sunt aplicate reguli de inferență pentru a deriva concluzii noi și procesând informațiile pâna la găsirea unei domenstrații sau epuizarea posibilităților.

Motorul aplicației implementate este funcția de unificare (Unify) care compară expresiile pentru identificarea substituțiilor care le pot face identice. Aceasta tratează mai multe cazuri specifice, spre exemplu când unda din expresii este o variabilă, când există predicate compuse sau liste cu variabile. În acest proces include și verificări pentru a preveni substituțiile recursive invalide cu metoda occur-check.

Căutarea și aplicarea substituțiilor se realizează prin analiza sistematică a listei de fapte pentru a găsi predicatele care se potrivesc cu antecedentele regulilor stabilite. Dacă se găsește o potrivire, algoritmul încearcă să unifice și să colecteze substituțiile valide, verificând în același timp dacă predicatele noi nu sunt deja existente în baza de cunoștințe.

În continuare, este exemplificat pseudocodul care stă la baza funcției Unify. Metoda funcționează comparând structurile de input, element cu element. Substituția ($\boldsymbol{\theta}$) care este argumentul metodei, este construit pe parcurs și este utilizat pentru a asigura faptul că ulterioarele comparații sunt consistente cu legăturile stabilite anterior. Într-o expresie compusă, cum ar fi F(A, B), funcția OP extrage simbolul funcției F, iar funcția ARGS extrage lista de argumente (A, B).

**function** UNIFY($x, y, \theta$) **returns** a substitution to make $x$ and $y$ identical
  **inputs**: $x$, a variable, constant, list, or compound
    $y$, a variable, constant, list, or compound
    $\theta$, the substitution built up so far (optional, defaults to empty)

  **if** $\theta$ = failure **then return** failure
  **else if** $x = y$ **then return** $\theta$
  **else if** VARIABLE?($x$) **then return** UNIFY-VAR($x, y, \theta$)
  **else if** VARIABLE?($y$) **then return** UNIFY-VAR($y, x, \theta$)
  **else if** COMPOUND?($x$) **and** COMPOUND?($y$) **then**
    **return** UNIFY(ARGS[$x$], ARGS[$y$], UNIFY(OP[$x$], OP[$y$], $\theta$))
  **else if** LIST?($x$) **and** LIST?($y$) **then**
    **return** UNIFY(REST[$x$], REST[$y$], UNIFY(FIRST[$x$], FIRST[$y$], $\theta$))
  **else return** failure

---

**function** UNIFY-VAR($var, x, \theta$) **returns** a substitution
  **inputs**: $var$, a variable
    $x$, any expression
    $\theta$, the substitution built up so far

  **if** $\{var/val\} \in \theta$ **then return** UNIFY($val, x, \theta$)
  **else if** $\{x/val\} \in \theta$ **then return** UNIFY($var, val, \theta$)
  **else if** OCCUR-CHECK?($var, x$) **then return** failure
  **else return** add $\{var/x\}$ to $\theta$

Metoda UNIFY parcurge următorii pași:

1. se verifică dacă substituția curentă $\boldsymbol{\theta}$ este null. dacă da, atunci returnează null.
2. altfel dacă x și y sunt identice, se returnează substituția curentă $\boldsymbol{\theta}$.
3. altfel dacă x este variabilă, se apelează funcția UNIFY-VAR care unifică un obiect și o variabilă.
4. altfel dacă y este variabilă, se apelează funcția UNIFY-VAR care unifică un obiect și o variabilă.
5. altfel dacă ambele expresii sunt compuse, atunci se face unificarea operatorilor și unificarea argumentelor.
6. altfel dacă x și y sunt liste de variabile atunci se aplică recursiv unificarea element cu element a celor două liste dacă au lungimi egale.
7. dacă nu se potrivește niciunul din cazurile anterioare se returnează null, deci unificarea nu este posibilă.

În cadrul algoritmului anterior este apelată o metodă UNIFY-VAR pentru unificarea unei variabile cu o expresie și parcurge următorii pași:

1. se verifică dacă substituția curentă $\boldsymbol{\theta}$ este null. dacă da, atunci returnează null.
2. dacă variabila are deja o valoare în substituția curentă, se unifică valoarea cu x.
3. altfel dacă variabila are deja o valoare în substituția curentă, unifică variabila cu acea valoare.
4. altfel dacă variabila apare în expresia x, atunci returnează null
5. dacă verificările anterioare nu au trecut, se adaugă noua substituție la lista $\boldsymbol{\theta}$ și se returnează $\boldsymbol{\theta}$.

Modulul de unificare este prezentat de pseudocodul care urmează.

```
function FOL-FC-ASK(KB, α) returns a substitution or false
  inputs: KB, the knowledge base, a set of first-order definite clauses
          α, the query, an atomic sentence
  local variables: new, the new sentences inferred on each iteration

  repeat until new is empty
    new ← { }
    for each sentence r in KB do
      (p1 ∧ ... ∧ pn ⇒ q) ← STANDARDIZE-APART(r)
      for each θ such that SUBST(θ, p1 ∧ ... ∧ pn) = SUBST(θ, p'1 ∧ ... ∧ p'n)
                  for some p'1, ..., p'n in KB
        q' ← SUBST(θ, q)
        if q' is not a renaming of some sentence already in KB or new then do
          add q' to new
          φ ← UNIFY(q', α)
          if φ is not fail then return φ
    add new to KB
  return false
```

Aceasta este o implementare a unui algoritm de înlănțuire înainte conceptual simplu, dar foarte ineficient. La fiecare iterație, acesta adaugă în baza de cunoștințe (KB) toate propozițiile atomice care pot fi deduse într-un singur pas din propozițiile de implicație și propozițiile atomice deja existente în KB.

Metoda FOL-FC-ASK parcurge următorii pași:

1. se declară o nouă listă de fapte.
2. iterează până când nu mai pot fi generate fapte noi. Controlează procesul de inferență până la epuizarea posibilităților.
3. se golește lista de fapte
4. parcurge toate regulile din baza de cunoștințe
5. standardizează variabilele pentru evitarea conflictelor și asigură nume unice
6. caută substituții care unifică antecedentele reguli cu faptele din KB și verifică dacă există potriviri pentru toate condițiile regulii
7. pentru fiecare $\boldsymbol{\theta}$ în lista de substituții, aplică substituțiile găsite la consecventul regulii și generează un nou potențial fapt.
8. dacă noul fapt nu este redundant, atunci se adaugă noul fapt în mulțime și pregătește faptul pentru a fi adăugat în KB.
9. se încearcă unificarea faptului cu noul query și se verifică dacă s-a găsit o soluție.
10. dacă nu s-a găsit o soluție, se returnează substituția găsită.
11. se adaugă toate faptele noi în baza de cunoștințe.
12. dacă nu s-a găsit nicio soluție, se returnează null, deci rezultă că acel query nu poate fi demonstrat.

# Modalitatea de rezolvare

Algoritmul implementat are la bază o structură orientată pe obiect și conține următoarele elemente principale:

- Variabile (clasa Variable): entități logice care conțin un nume și reprezintă elementele de bază ale sistemului.
- Predicate (clasa Predicate): structuri care conțin numele propoziției și o listă de argumente
- Clauze (clasa Clause): construcții logice formate din antecedent (listă de predicate) și consecvent (un predicat).
- Baza de cunoștințe (clasa KnowledgeBase): colecție de date organizată formată dintr-o listă de fapte (predicate) și o listă de reguli (clauze).

Motorul de inferență este implementat în clasa InferenceEngine, care conține algoritmul principal, funcțiile necesare pentru raționamentul respectiv. Clasa este responsabilă pentru aplicarea regulilor de inferență și derivarea faptelor noi.

Acest mod de organizare permite o delimitare clară a responsabilităților fiecărei clase și funcții.

# Părți semnificative din codul sursă

Clasa Variable:

```
/// <summary>
/// reprezinta o variabila in motorul de inferenta predicativa
/// clasa este folosita pentru a stoca numele variabilelor in expresiile logice
/// </summary>
99+ references
public class Variable
{
    5 references
    public string Name { get; set; }

    21 references
    public Variable(string name)
    {
        this.Name = name;
    }
    0 references
    override public string ToString()
    {
        return Name;
    }
}
```

Clasa Predicate:

```
/// <summary>
/// reprezinta o propozitie logica in motorul de inferenta predicativa
/// clasa este folosita pentru a stoca numele propozitiei si argumentele sale
/// </summary>
88 references
public class Predicate
{
    10 references
    public string Name { get; set; }
    12 references
    public List<Variable> Arguments { get; set; }

    1 reference
    public Predicate()
    {
        Name = string.Empty;
        Arguments = new List<Variable>();
    }

    56 references
    public Predicate(string name, List<Variable> arguments)
    {
        Name = name;
        Arguments = arguments ?? throw new ArgumentNullException(nameof(arguments));
    }
```

```
5 references
public override string ToString()
{
    return $"{Name}({string.Join(", ", Arguments)})";
}

 /// <summary>
 /// metoda override pentru a compara doua propozitii
 /// </summary>
 /// <param name="obj">propozitia cu care se compara</param>
 /// <returns>true daca propozitiile sunt egale, altfel false</returns>
1 reference
public override bool Equals(object obj)
{
    if(obj is Predicate otherPredicate)
    {
        return Name == otherPredicate.Name && Arguments.Count == otherPredicate.Arguments.Count && Arguments.SequenceEqual(otherPredicate.Arguments);
    }
    return false;
}
}
```

Clasa Clause:

```
/// <summary>
/// reprezinta o clauza in motorul de inferenta predicativa
/// clasa este folosita pentru a stoca antecedentul si consecventul unei clauze
/// </summary>
32 references
public class Clause
{
    6 references
    public List<Predicate> Antecedent {  get; set; }
    5 references
    public Predicate Consecvent {  get; set; }

    13 references
    public Clause()
    {
        Antecedent = new List<Predicate>();
        Consecvent = new Predicate();
    }

    0 references
    public Clause(List<Predicate> antecedent, Predicate consecvent)
    {
        Antecedent = antecedent ?? throw new ArgumentNullException(nameof(antecedent));
        Consecvent = consecvent ?? throw new ArgumentNullException(nameof(consecvent));
    }

    /// <summary>
    /// adauga un predicat la antecedent
    /// </summary>
    /// <param name="predicate">predicatul care se adauga</param>
    24 references
    public void AddToTheLeft(Predicate predicate)
    {
        if (predicate == null)
            throw new ArgumentNullException(nameof(predicate));
        Antecedent.Add(predicate);
    }
```

```
    /// <summary>
    /// seteaza consecventul unei clauze
    /// </summary>
    /// <param name="consecvent">predicatul care se seteaza</param>
    13 references
    public void SetConsecvent(Predicate consecvent)
    {
        if(consecvent == null)
            throw new ArgumentNullException(nameof(consecvent));
        Consecvent = consecvent;
    }

    4 references
    public override string ToString()
    {
        string antecedentStr = Antecedent.Count > 0 ? string.Join("^", Antecedent) : "True";
        return $"{antecedentStr} => {Consecvent}";
    }
}
```

Clasa KnowledgeBase:

```
/// <summary>
/// clasa pentru reprezentarea unei baze de cunostinte
/// </summary>
10 references
public class KnowledgeBase
{
    22 references
    public List<Predicate> Facts {  get; set; }
    19 references
    public List<Clause> Rules {  get; set; }

    4 references
    public KnowledgeBase()
    {
        Facts = new List<Predicate>();
        Rules = new List<Clause>();
    }
}
```

Algoritmul principal:

```
/// <summary>
/// reprezinta motorul de inferenta predicativa
/// clasa contine metode pentru inferenta si unificare
/// </summary>
8 references
public class InferenceEngine
{
    /// <summary>
    /// metoda pentru inferenta folosind metoda inlatuirii inaite (forward chaining) pentru gasirea
    /// unei substitutii pentru o propozitie data
    /// </summary>
    /// <param name="KB">baza de cunostinte</param>
    /// <param name="alpha">propozitia ce trebuie demonstrata</param>
    /// <returns>substitutiile daca propozitia este demonstrabila, altfel null</returns>
    4 references
    public Dictionary<string, object>? FOL_FC_Ask(KnowledgeBase KB, Predicate alpha)
    {
        var newPredicates = new List<Predicate>();
        do
        {
            newPredicates.Clear();
            foreach(Clause r in KB.Rules)
            {
                var subst_list = FindSubstitutions(KB.Facts, r);
                foreach (var theta in subst_list)
                {
                    Predicate q_prime = SubstPredicate(theta, r.Consecvent);
                    if(!IsRenaming(q_prime, newPredicates) && !IsRenaming(q_prime, KB.Facts))
                    {
                        Console.WriteLine(q_prime.ToString());
                        newPredicates.Add(q_prime);
                        var phi = Unify(q_prime, alpha, new Dictionary<string, object>());
                        if(phi != null)
                        {
                            return phi;
                        }
                    }
                }
            }
            KB.Facts.AddRange(newPredicates);
        }
        while (newPredicates.Count > 0);
        return null;
    }
```

Funcția de găsire a substituțiilor:

```
/// <summary>
/// metoda pentru gasirea substitutiilor pentru un antecedent si un consecvent
/// </summary>
/// <param name="facts">lista de fapte</param>
/// <param name="clause">clauza</param>
/// <returns>lista de substitutii</returns>
1 reference
private List<Dictionary<string, object>>? FindSubstitutions(List<Predicate> facts, Clause clause)
{
    var substitutionsList = new List<Dictionary<string, object>>();

    foreach (var antecedentPredicate in clause.Antecedent)
    {
        foreach (var fact in facts)
        {
            if (fact.Name == antecedentPredicate.Name)
            {
                var theta = Unify(antecedentPredicate, fact, new Dictionary<string, object>());
                if (theta != null)
                {
                    substitutionsList.Add(theta);
                }
            }
        }
    }
    return substitutionsList;
}
```

Funcția care aplică substituția $\boldsymbol{\theta}$ asupra unui predicat:

```
/// <summary>
/// metoda pentru aplicarea unei substitutii la un predicat
/// </summary>
/// <param name="theta">substitutiile</param>
/// <param name="predicate">predicatul</param>
/// <returns>predicatul cu substitutii</returns>
1 reference
private Predicate SubstPredicate(Dictionary<string, object> theta, Predicate predicate)
{
    var substitutes = new List<Variable>();
    foreach (var arg in predicate.Arguments)
    {
        if (arg is Variable var && theta.TryGetValue(var.Name, out var substitution))
        {
            if (substitution is Variable subVar)
            {
                substitutes.Add(subVar);
            }
            else
            {
                substitutes.Add(new Variable(substitution.ToString()));
            }
        }
        else
        {
            substitutes.Add(arg);
        }
    }

    return new Predicate(predicate.Name, substitutes);
}
```

Funcția care verifică dacă un predicat deja face parte dintr-o listă de predicate:

```
/// <summary>
/// metoda pentru verificarea daca un predicat este o redenumire din lista
/// </summary>
/// <param name="predicate">predicatul</param>
/// <param name="Arguments">lista de predicate</param>
/// <returns>true daca predicatul este o redenumire, altfel false</returns>
2 references
private bool IsRenaming(Predicate predicate, List<Predicate> Arguments)
{
    return Arguments.Any(existingPredicate => predicate.Equals(existingPredicate));
}
```

Funcțiile din algoritmul de unificare:

```
/// <summary>
/// metoda pentru unificarea a doua expresii
/// </summary>
/// <param name="x">expresia 1</param>
/// <param name="y">expresia 2</param>
/// <param name="theta">substitutiile</param>
/// <returns>substitutiile daca unificarea este posibila, altfel null</returns>
7 references
public Dictionary<string, object>? Unify(object x, object y, Dictionary<string, object> theta)
{
    if (theta == null)
    {
        return null;
    }
    else if (x.Equals(y))
    {
        return theta;
    }
    else if (IsVariable(x))
    {
        return UnifyVar(x as Variable, y, theta);
    }
    else if (IsVariable(y))
    {
        return UnifyVar(y as Variable, x, theta);
    }
    else if (IsCompound(x) && IsCompound(y))
    {
        return Unify((x as Predicate).Arguments, (y as Predicate).Arguments, Unify((x as Predicate).Name, (y as Predicate).Name, theta));
    }
    else if (x is List<Variable> listX && y is List<Variable> listY)
    {
        if (listX.Count != listY.Count)
        {
            return null;
        }

        for (int i = 0; i < listX.Count; i++)
        {
            theta = Unify(listX[i], listY[i], theta);
            if (theta == null)
            {
                return null;
            }
        }
        return theta;
    }
    else return null;
}
```

Funcție pentru unificarea unei variabile cu o expresie:

```
/// <summary>
/// metoda pentru unificarea unei variabile cu o expresie
/// </summary>
/// <param name="var">variabila</param>
/// <param name="x">expresia</param>
/// <param name="theta">substitutia</param>
/// <returns>substitutia daca unificarea este posibila, altfel null</returns>
2 references
public Dictionary<string, object>? UnifyVar(Variable var, object x, Dictionary<string, object> theta)
{
    if(x == null)
    {
        return null;
    }
    if(theta.TryGetValue(var.Name, out object? val))
    {
        return Unify(val, x, theta);
    }
    else if(theta.TryGetValue(x.ToString(), out object? valX))
    {
        return Unify(var, valX, theta);
    }
    else if(OccurCheck(var, x))
    {
        return null;
    }
    else
    {
        theta.Add(var.Name, x);
        return theta;
    }
}
```

Funcție pentru verificare dacă o variabilă apare într-o expresie:

```
/// <summary>
/// metoda pentru verificarea daca o variabila apare in expresie
/// </summary>
/// <param name="var">variabila</param>
/// <param name="x">expresia</param>
/// <returns>true daca variabila apare in expresie, altfel false</returns>
1 reference
private bool OccurCheck(Variable var, object x)
{
    return x is Predicate predicate && predicate.Arguments.Contains(var);
}
```

Funcție pentru verificare dacă un obiect este variabilă:

```
/// <summary>
/// metoda pentru verificarea daca un obiect este o variabila
/// </summary>
/// <param name="obj">obiectul</param>
/// <returns>true daca obiectul este o variabila, altfel false</returns>
2 references
private bool IsVariable(object obj)
{
    return obj is Variable;
}
```

Funcție pentru verificare dacă un obiect este predicat compus:

```
/// <summary>
/// metoda pentru verificarea daca un obiect este un predicat compus (cu argumente)
/// </summary>
/// <param name="obj">obiectul</param>
/// <returns>true daca obiectul este un predicat compus, altfel false</returns>
2 references
private bool IsCompound(object obj)
{
    return obj is Predicate && (obj as Predicate).Arguments != null;
}
```

# Rezultatele obținute în urma rezolvării unor probleme matematice și din viața cotidiană

1. Verificarea bijectivității funcției f:{1, 2}->{10, 20}, f(1)=10 f(2)=20

```
Bijectivitatea unei functii

Fapte:
Distinct(1, 2)
Map(F, 1, 10)
Map(F, 2, 20)

Reguli:
Distinct(X, Y)^Map(F, X, Z)^Map(F, Y, Z) => Injective(F)
Map(F, X, Z) => Surjective(F)
Injective(F)^Surjective(F) => Bijective(F)

Injective(F)
Surjective(F)
Bijective(F)
Propozitia poate fi demonstrata: Functia este bijectiva.
```

2. Lema cleștelui

```
Lema clestelui

Fapte:
lim(xn, a)
lim(zn, a)
MaiMare(yn, xn)
MaiMare(zn, yn)

Reguli:
lim(xn, a) => valoare(xn, infinit, a)
MaiMare(yn, xn)^MaiMare(zn, yn)^valoare(xn, infinit, a)^valoare(zn, infinit, a) => lim(yn, a)
lim(zn, a) => valoare(zn, infinit, a)

valoare(xn, infinit, a)
valoare(zn, infinit, a)
lim(yn, a)
Propozitia poate fi demonstrata.
```

Programul folosește algoritmul și pentru a rezolva următoarele două cazuri de test:

1. Determină dacă o persoană este eligibilă pentru un premiu

- Regula 1: dacă o persoană are performanțe academice excelente, este calificată academic;
- Regula 2: dacă o persoană participă la activități de voluntariat, este implicată comunitar;
- Regula 3: dacă o persoană demonstrează abilități de conducere, este lider;
- Regula 4: dacă o persoană este calificată academic, implicată comunitar și lider, este eligibilă pentru premiu.

```
Testarea eligibilitatii pentru un premiu

Fapte:
HasExcellentPerformance(Person)
ParticipatesInCommunityService(Person)
DemonstratesLeadership(Person)

Reguli:
HasExcellentPerformance(Person) => AcademicallyQualified(Person)
ParticipatesInCommunityService(Person) => CommunityEngaged(Person)
DemonstratesLeadership(Person) => Leader(Person)
AcademicallyQualified(Person)^CommunityEngaged(Person)^Leader(Person) => EligibleForAward(Person)

AcademicallyQualified(Person)
CommunityEngaged(Person)
Leader(Person)
EligibleForAward(Person)
Propozitia poate fi demonstrata: Persoana este eligibila pentru premiu.
```

2. Verifică dacă o companie este vinovată de poluarea mediului

- Regula 1: dacă o companie emite un poluant găsit într-o locație, este responsabilă de poluarea acelei locații;
- Regula 2: dacă un poluant cauzează un impact negativ, este considerat poluare;
- Regula 3: dacă o companie este responsabilă pentru poluare, este vinovată

```
Testarea vinovatiei intr-un caz de poluare industriala

Fapte:
Emits(Company, Pollutant)
FoundAtLocation(Pollutant, Location)
CausesImpact(Pollutant, Impact)
NegativeImpact(Impact)

Reguli:
Emits(Company, Pollutant)^FoundAtLocation(Pollutant, Location) => ResponsibleForPollution(Company, Location)
CausesImpact(Pollutant, Impact)^NegativeImpact(Impact) => Pollution(Pollutant)
ResponsibleForPollution(Company, Location)^Pollution(Pollutant) => GuiltyOfPollution(Company)

ResponsibleForPollution(Company, Location)
Pollution(Pollutant)
GuiltyOfPollution(Company)
Propozitia poate fi demonstrata: Compania este vinovata de poluare.
```

# Concluzii

În concluzie, Motorul de inferență predicativă prin raționament înainte rezolvă două probleme matematice și probleme din viața cotidiană. Algoritmul implementat gestionează faptele și regulile, identifică substituții relevante și produce rezultate consistente și robuste.

# Bibliografie


http://aima.cs.berkeley.edu/4th-ed/pdfs/newchap09.pdf

https://www.isjsalaj.ro/red/resurse/liceal_postliceal/matematica_si_stiinte_ale_naturii/16-07-2020-Lucaciu_Simona_Criteriul-clestelui-in-calculul-limitelor-de-siruri-integrale_cls-XII_Matematica.pdf

https://schoolmodeon.com/topic/injectivitate-surjectivitate-bijectivitate-si-inversabilitate-2/

# Listă cu ce a făcut fiecare

Diaconu Rareș-George:

- Partea de cod: clasele Clause, KnowledgeBase, PollutionGuiltDeterminationTest, AwardEligibilityTest, InferenceEngine: funcțiile FOL_FC_Ask, FindSubstitutions, SubstPredicate, IsRenaming;
- Documentație: Descrierea problemei considerate, Modalitatea de rezolvare, Părți semnificative din codul sursă, Rezultatele obținute în urma rezolvării unor probleme matematice și din viața cotidiană, Concluzii, Bibliografie.

Huminiuc Simona:

- Partea de cod: clasele Variable, Predicate, BijectiveFunctionTest, PincerLemmaTest, InferenceEngine: funcțiile Unify, UnifyVar, OccurCheck, IsVariable, IsCompound;
- Documentație: Aspecte teoretice privind algoritmul.