ReasonSharp

Rezolvitorul de probleme prin inferență predicativă cu raționament înainte

```
Cazul 1 criminalitatea lui West:

Fapte:
Owns(Nono, M1)
Missile(M1)
American(West)
Enemy(Nono, America)

Reguli:
American(X) AND Weapon(Y) AND Sells(X, Y, Z) AND Hostile(Z) => Criminal(X)
Missile(X) AND Owns(Nono, X) => Sells(West, X, Nono)
Missile(X) => Weapon(X)
Enemy(X, America) => Hostile(X)

Sells(West, M1, Nono)
Weapon(M1)
Hostile(Nono)

Criminal(West)
Propozitia este satisfiabila.
```

Realizat de:

Dancău Rareș-Andrei, 1410A

Cuprins

Descrierea problemei considerate

În cadrul programului am ales să rezolv folosind algoritmul de inferență predicativă cu raționament înainte două probleme matematice.

1. Verificarea dacă o funcție este bijectivă.

Definiţia 1.2.2 *Fie $f : X \to Y$ o funcţie.*

*(i) Spunem că f este o **injecţie** sau că este **funcţie injectivă** dacă pentru orice $x_1, x_2 \in X$ cu $x_1 \neq x_2$ avem $f(x_1) \neq f(x_2)$ sau, echivalent,*

$$(x_1, x_2 \in X, f(x_1) = f(x_2)) \Rightarrow x_1 = x_2.$$

*(ii) Spunem că f este o **surjecţie** sau că este o **funcţie surjectivă** dacă pentru orice $y \in Y$ există $x \in X$ astfel încât $f(x) = y$ sau, echivalent,*

$$f(X) = Y.$$

*(iii) Spunem că funcţia f este o **bijecţie** sau că este **funcţie bijectivă** dacă este, simultan, injectivă şi surjectivă.*

2. Lema cleștelui.

Propoziţia 2.4.13 (Lema cleştelui) *Dacă $x_n \leq y_n \leq z_n$ pentru $n \geq n_1$ şi dacă $\lim_{n\to\infty} x_n = \lim_{n\to\infty} z_n = x$ atunci $\lim_{n\to\infty} y_n = x$.*

Am ales aceste probleme deoarece au relevanță matematică și premisele lor nu sunt nici numeroase și nici complexe.

Aspecte teoretice privind algoritmul

Algoritmul de inferență predicativă implementat în cadrul aplicației ReasonSharp utilizează un set de reguli și fapte pentru a realiza raționament logic asupra predicatelor definite în baza de cunoștințe.

Algoritmul își propune să determine satisfiabilitatea unei propoziții date, astfel că, în timpul inferenței, noile predicate generate sunt integrate în lista de fapte pentru a extinde baza de cunoștințe.

Pentru a identifica substituțiile potrivite, algoritmul efectuează un proces de unificare între predicatul curent și propoziția de interes. Acest proces implică gestionarea substituțiilor variabilelor și verificarea unor condiții precum apartenența și non-ocurența, asigurând astfel consistența inferențelor. Algoritmul continuă acest ciclu până când nu mai sunt generate noi predicte sau propoziția este declarată satisfiabilă.

Prin intermediul acestei abordări, algoritmul de inferență predicativă în ReasonSharp furnizează o metodă robustă și eficientă pentru raționament logic în contextul bazei de cunoștințe definite de programator, pentru a rezolva probleme matematice date de către utilizator.

Mai jos se poate observa pseudocodul modulului de unificare din cadrul programului:

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

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

---

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

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

Acest modul compară mai întâi structurile parametrilor, element cu element. Substituția theta este argumentul care inițial nu are valoare și care este construit pe măsura executării algoritmului de unificare și este folosit pentru a asigura coerența comparațiilor ulterioare comparativ cu cele anterioare. Pentru expresiile compuse, cum ar fi F(A,B), funcția OP va lua operatorul F și ARGS va prelua lista de parametri.

Modulul de unificare este apelat din modulul principal al algoritmului, al cărui pseudocod este disponibil mai jos:

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
    $new \leftarrow \{\}$
    for each sentence r **in** KB **do**
      $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-APART(r)
      for each θ such that SUBST($\theta, p_1 \wedge \ldots \wedge p_n$) = SUBST($\theta, p'_1 \wedge \ldots \wedge p'_n$)
          for some $p'_1, \ldots, p'_n$ in KB
        $q' \leftarrow$ SUBST(θ, q)
        if q' is not a renaming of some sentence already in KB or new **then do**
          add q' to new
          $\phi \leftarrow$ UNIFY(q', α)
          if ϕ is not *fail* **then return** ϕ
    add new to KB
  return *false*

Este o implementare directă din punct de vedere conceptual, dar foarte ineficientă, a inferenței predicative cu raționament înainte. La fiecare pas, adaugă propozițiile atomice ce pot fi determinate din implicații la baza de cunoștințe dacă nu se aflau deja.

Modalitatea de rezolvare

În cadrul rezolvării, mi-am definit mai multe clase pentru a prezenta conceptele necesare implementări algoritmului.

Astfel, pentru variabile am definit o clasă care conține un parametru Name care reprezintă valoarea variabilei la un moment dat. Aceste variabile pot face parte din predicate.

Pentru predicate, am implementat o clasă care reține numele predicatului și o listă de argumente ce pot fi șiruri de caractere sau variabile.

Mi-am definit o clasă pentru a reține conjuncții de predicate, căreia i-am zis clauză. Clauza reține un antecedent și consecvent. Antecedentul este o conjuncție de predicate pozitive, reprezentată printr-o listă de predicate. Consecventul reține un predicat.

Pentru a reprezenta baza de cunoștințe, mi-am creat o clasă ce conține o listă de clauze, numite reguli, și o listă de predicate, numite fapte.

Algoritmul și funcțiile auxiliare fac parte din clasa InferenceEngine, ce sunt apelate din programul principal.

În următoarea secțiune se pot vedea implementările efective pentru cele mai importante părți ale programului.

Părți semnificative din codul sursă

Implementarea clasei pentru variabile:

```
//Clasa pentru variabila; Variabila este definita printr-un nume.
    public class Variable
    {
        public string Name { get; set; }

        public Variable(string name)
        {
            Name = name;
        }

        override public string ToString()
        {
            return Name;
        }
        public override bool Equals(object obj)
        {
            if (obj is Variable)
                return (obj as Variable).Name == this.Name;
            return false;
```

```
        }
    }
```

Implementarea clasei pentru predicate:

```
//Clasa pentru predicate. Predicatul are un nume si o lista de argumente care poate fi compusa
din variabile sau string-uri
    public class Predicate
    {
        public string Name { get; set; }
        public List<object> Arguments { get; set; }
        public Predicate()
        {
            Name = "";
            Arguments = new List<object>();
        }
        public Predicate(string name, List<object> arguments)
        {
            Name = name;
            Arguments = arguments;
        }
        public string stringify()
        {
            //Functia de transformare a unui predicat in string; Este folosita pentru a afisa
la consola predicatul
            if (this.Arguments.Count == 0)
                return "";
            if (this.Arguments.Count == 1)
                return this.Name + "(" + this.Arguments[0].ToString() + ")";
            string a=this.Name+"("+this.Arguments[0].ToString();
            foreach(var arg in this.Arguments.Skip(1))
            {
                a += ", " + arg.ToString();
            }
            a += ")";
            return a;
        }
    }
```

Implementarea clasei pentru clauze:

```
 public class Clause
    {
        //Clauza este definita de ordinul intai; Antecedentul este o conjunctie de literali
pozitivi
        //Consecventul este un singur literal pozitiv
        public List<Predicate> ps { get; set; }
        public Predicate q { get; set; }
        public Clause()
        {
            ps = new List<Predicate>();
            q = new Predicate();
        }

        public void Addp(Predicate predicate)
        {
            ps.Add(predicate);
        }
```

```
        public void Setq(Predicate predicate)
        {
            q = predicate;
        }

        public string stringify()
        {
            var predicateStrings = ps.Select((p, index) =>
            {
                // Nu afișez operatorul pentru primul predicat sau pentru un singur predicat
                if (index == 0 || ps.Count == 1)
                {
                    return p.stringify();
                }

                return $"AND {p.stringify()}";
            });
            string ret = string.Join(" ", predicateStrings);
            ret += " => " + q.stringify();
            return ret;
        }
    }
```

Clasa pentru baza de cunoștințe:

```
 //Clasa pentru baza de cunostinte, contine o lista de reguli si o lista de fapte
    //Faptele sunt propozitii atomice, iar regulile sunt clauze definite de ordinul intai.
    public class KnowledgeBase
    {
        public List<Clause> Rules { get; set; }
        public List<Predicate> Facts { get; set; }
        public KnowledgeBase()
        {
            Rules = new List<Clause>();
            Facts = new List<Predicate>();
        }

        public void AddRule(Clause clause)
        {
            Rules.Add(clause);
        }
        public void AddFact(Predicate predicate)
        {
            Facts.Add(predicate);
        }
        public void Clear()
        {
            Facts.Clear();
            Rules.Clear();
        }
    }
```

Algoritmul principal:

```
public Dictionary<string, string> FOL_FC_ASK(KnowledgeBase KB, Predicate alpha)
```

```
        {
            var newPredicates = new List<Predicate>();
            do {
                newPredicates.Clear();
                foreach (Clause r in KB.Rules)
                {
                    Clause standard = r;
                    var substitutionsList = IdentifySubstitutions(KB.Facts, standard);
//Determinare substitutii posibile=egalitate intre substitutii
                    foreach (var theta in substitutionsList)
                    {
                            Predicate q_prime = SubstPredicate(theta, standard.q);
                            if (!IsRenaming(q_prime, newPredicates) && !IsRenaming(q_prime,
KB.Facts))
                            {
                              //Afisare propozitii noi la fiecare pas
                               Console.WriteLine(q_prime.stringify());
                            newPredicates.Add(q_prime);
                                var phi = Unify(q_prime, alpha,new Dictionary<string,
string>());
                            if (phi != null)
                            {
                                return phi;
                            }
                            }
                    }
                }
                KB.Facts.AddRange(newPredicates);
                Console.WriteLine("");
            } while (newPredicates.Count > 0);
            return null;
        }
```

Funcția de determinare a substituțiilor posibile:

```
        private List<Dictionary<string, string>> IdentifySubstitutions(List<Predicate> facts,
Clause standard)
        {
            var substitutionsList = new List<Dictionary<string, string>>();
            foreach(Predicate predicate in standard.ps)
            {
                //Daca pentru un predicat din regula nu exista echivalent in fapte, atunci nu
pot face substitutie
                if (!facts.Exists(x => x.Name == predicate.Name))
                    return new List<Dictionary<string, string>>();
                foreach (Predicate predicate1 in facts)
                {//Daca am un predicat in baza de fapte si unificarea este nula, atunci nu pot
face substitutia
                    if (predicate.Name == predicate1.Name)
                    {
                        var theta = Unify(predicate, predicate1, new Dictionary<string,
string>());
                        if (theta != null)
                            substitutionsList.Add(theta);
                    }
                }
            }
```

```
            return substitutionsList;
        }
```

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

```
public bool IsRenaming(Predicate predicate, List<Predicate> predicates)
        {
            //Parcurg lista de predicate si verific daca exista o egalitate cu unul din
predicate
            foreach (Predicate existingPredicate in predicates)
            {
                if (ArePredicatesEqual(predicate, existingPredicate))
                {
                    return true;
                }
            }
            return false;
        }

        private bool ArePredicatesEqual(Predicate predicate1, Predicate predicate2)
        {
            // Verifică dacă numele și argumentele predicatelor sunt identice
            return predicate1.Name == predicate2.Name &&
predicate1.Arguments.SequenceEqual(predicate2.Arguments);
        }
```

Funcția care aplică substituția theta asupra unui predicat:

```
 private Predicate SubstPredicate(Dictionary<string, string> theta, Predicate predicate)
        {
            //Aplic substitutia theta pe fiecare argument
            var substitutedArguments = predicate.Arguments.Select(arg => SubstArgument(theta,
arg)).ToList();
            return new Predicate(predicate.Name, substitutedArguments);
        }
        private object SubstArgument(Dictionary<string, string> theta, object argument)
        {
            //Daca argumentul meu este o variabila si am in theta o substitutie, intorc o noua
variabila cu numele din substitutie
            if (argument is Variable var && theta.TryGetValue(var.Name, out var substitution))
            {
                return new Variable(substitution);
            }
            //Daca nu sunt in cazul de mai sus, fie am string ca argument fie nu exista
substitutie in theta pentru variabila
            return argument;
        }
```

Funcțiile din algoritmul de unificare

```
public Dictionary<string,string> Unify(object x,object y,Dictionary<string,string> theta)
        {
            //Algoritmul clasic de unificare
            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 (IsList(x) && IsList(y))
            {
                if ((x as List<object>).Count == 0 && (y as List<object>).Count == 0)
                    return theta;
                else if ((x as List<object>).Count>0 && (y as List<object>).Count > 0)
                {
                    var unifiedFirst = Unify((x as List<object>).First(), (y as
List<object>).First(), theta);
                    if (unifiedFirst != null)
                        return Unify((x as List<object>).Skip(1).ToList(), (y as
List<object>).Skip(1).ToList(), unifiedFirst);
                }
                return null;
            }
            else return null;
        }
        public Dictionary<string, string> UnifyVar(Variable v,object
x,Dictionary<string,string> theta)
        {
            //Algoritmul clasic de unificare intre o variabila si un obiect
            if (x == null)
                return null;
            if(theta.TryGetValue(v.Name, out var val))
            {
                return Unify(val, x, theta);
            }
            if(theta.TryGetValue(x.ToString(),out var existingVal))
            {
                return Unify(v, val, theta);
            }
            else if (OccurCheck(v, x))
            {
                return null;
            }
            else
            {
                theta.Add(v.Name, x.ToString());
                return theta;
            }
        }
```

Rezultatele obținute în urma rezolvării unor probleme matematice

1. Verificarea bijectivității funcției f:{3, 4}->{6, 8}, f(3)=6 f(4)=8

```
Cazul 1 verificarea bijectivitatii unei functii

Fapte:
Inegal(3, 4)
Diferit(f, 3, 4)
Egal(f, 3, 6)
Egal(f, 4, 8)

Reguli:
Inegal(X1, X2) AND Diferit(f, X1, X2) => injectiva(f)
Egal(f, X1, Y) => surjectiva1(f)
Egal(f, X2, B) AND Inegal(X1, X2) => surjectiva2(f)
surjectiva1(f) AND surjectiva2(f) => surjectiva(f)
injectiva(f) AND surjectiva(f) => bijectiva(f)

injectiva(f)
surjectiva1(f)
surjectiva2(f)

surjectiva(f)

bijectiva(f)
Propozitia este satisfiabila.
```

2. Lema cleștelui

```
Cazul 2 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) AND MaiMare(zn, yn) AND valoare(xn, infinit, a) AND valoare(zn, infinit, a) => lim(yn, a)

valoare(xn, infinit, a)
valoare(zn, infinit, a)

lim(yn, a)
Propozitia este satisfiabila.
```

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

1. Determină dacă West este criminal

The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.

```
Cazul 1 de test:

Fapte:
Owns(Nono, M1)
Missile(M1)
American(West)
Enemy(Nono, America)

Reguli:
American(X) AND Weapon(Y) AND Sells(X, Y, Z) AND Hostile(Z) => Criminal(X)
Missile(X) AND Owns(Nono, X) => Sells(West, X, Nono)
Missile(X) => Weapon(X)
Enemy(X, America) => Hostile(X)

Sells(West, M1, Nono)
Weapon(M1)
Hostile(Nono)

Criminal(West)
Propozitia este satisfiabila.
```

2. Verifică dacă John este malefic.

$$\forall x \;\; King(x) \wedge Greedy(x) \;\Rightarrow\; Evil(x)$$
$$King(John)$$
$$Greedy(John)$$

```
Cazul 2 de test:

Fapte:
King(John)
Greedy(John)

Reguli:
King(X) AND Greedy(X) => Evil(X)

Evil(John)
Propozitia este satisfiabila.
```

Concluzii

În concluzie, ReasonSharp rezolvă câteva probleme din domeniul matematicii, dar și probleme de logică generale. Algoritmul implementat gestionează bazele de cunoștințe, identifică substituții relevante și produce rezultate consistente.

Bibliografie

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

http://math.etc.tuiasi.ro/rstrugariu/cursuri/AM2020/c1.pdf pentru accesare parola este csdfitp

http://math.etc.tuiasi.ro/rstrugariu/cursuri/AM2020/c2.pdf pentru accesare parola este csdfitp