Djordje Simovic 2013/0162

Inteligentni Sistemi , Januar 2016, projekat: Simulacija ekspertskog sistema.

**O projektu:**

Tema projekta je rad na ekspertskom sistemu u neizvesnom okruzenju, simulacije njegovog izvrsavanja, kao i modeliranja celokupnog sistema.

**Tehnologije:**

Projekat je napisan u C# programskom jeziku, kao WPF (windows presentation form) aplikacija. Zbog radoznalosti i zelje za vezbanjem novog jezika izabrao sam C#.

**Pretpostavka:**

U narednom delu teksta podrazumeva se da je citaoc vec upoznat sa svim proracunima i terminima koje se traze u zadatku, te cu ovde navesti glavne formule, koje ce kasnije biti refersirane:

**Kumulativni Faktor:**

* Cf(z) = Mb(z) - Md(z)

**Mera Poverenja:**

* za konkretno opazanje se ocekuje da bude zadata od strane korisnika, default vrednost je 0. Za Mb() se vraca vrednost ObservedFactor-a koji se zadaje, a za Md() vrednost NotObservedFactor-a koji se takodje zadaje.
* za zakljucak Mb(z, uslovi) = MbStrength() \* max(Cf(uslovi), 0)
* za operator ILI Mb(ILI(e1,e2)) = max(Mb(e1),Mb(e2))
* za operator I isto kao ILI samo umesto max => min
* za operator - (negacija) Mb(-(e1)) = Md(e1)

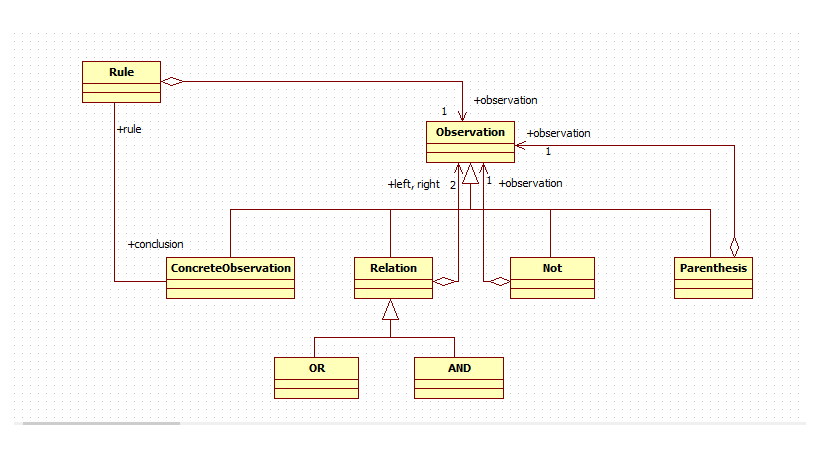
**Mera Nepoverenja:**  Sve isto kao mera poverenja, razlike:

* za operator ILI Md(ILI(e1,e2)) = min(Mb(e1),Mb(e2))
* za operator I isto kao ILI samo umesto min => max
* za operator - (negacija) Md(-(e1)) = Mb(e1)

**Model:**

Na prvi pogled sistem mi je delovao kao stvoren za objektno orijentisan pristup. Naime, logika kojom se vodi ovaj sistem, odnosno pravila koja prati su zapravo najobicniji logicki izrazi, odakle mi je i dosla ideja za ovakav model.

U nastavku se nalazi kratak prikaz glavnog dela sistema koji se odnosi na srz programa.



Interfejs Observation klase ima tri metode: Cf() - sablonska metoda. Mb(), Md() - apstraktne

ConcreteObservation predstavlja klasu objekata opazanja i zakljucaka. Nakon dosta razmisljanja dosao sam do sledeceg zakljucka: svako opazanje je potencijalan zakljucak. Koristeci ovakav model korisnik nije primoran da sam topoloski sortira svoj sistem pravila da bi ispravno uneo podatke i zapoceo simulaciju, vec moze da unosi pravila po bilo kom redosledu, a program ce se pobrinuti za pravilnu identifikaciju objekata unutar sistema. OR i AND su klasicni binarni logicki predikati koje prate interfejs Relation klase. NOT je dekorater koji samo obrce Mb() i Md() dekorisanog objekta(za Md() vraca Mb(), i obrnuto). Parenthesis je dekorater koji ispisuje otvorene i zatvorene zagrade oko imena dekoirsanog objekta.

Dakle, sta se tu desava: Svako pravilo odvajamo u zaseban objekat koji sadrzi jedan zakljucak, i jednu observaciju koja predstavlja rezultujuci logicki izraz. Do tog logickog izraza smo dosli pomoci parsiranja izvornog infix izraza u postfix, pomocu klase PostfixConverter o kojoj cu kasnije pricati, i iz postfixa se veoma lako i brzo dolazi do sjedinjenog objekta koji predstavlja observaciju. Nakon pravljenja observacije, moramo napraviti i zakljucak. Ok to je lako, samo parsiramo sve posle "ONDA" dela, izvucemo ime i faktor i napravimo objekat. Hmm, sta se onda desava kada dodamo jos jedno pravilo koje ima potpuno drugaciji izraz a isti zakljucak. Za to ce nam biti potrebne dodatne metode kao sto su MbStrength() i MdStrength() koje odredjuju jacinu zakljucka, kao i MbCumulative() i MdCumulative() zarad odredjivanja kumulativnih verovatnoca izmedju svih pojava zakljucka u pravilima (naravno, kao zakljucka). Ok sada imamo nacin da izracunamo kumulativne faktore Cf za zakljucke. Sta jos moze da se popravi? Na primer kada bi imali p1 = AKO a ILI b ONDA c, p2 = AKO c ILI d ONDA f, p3 = AKO c ILI f ONDA g , i pozvali Cf(g) Mb() i Md() zakljucka odnosno opazanja C bi se pozvale 2 puta(u detalje necu da ulazim, ali i Cf() unutar proracuna jacine opazanja zakljucka C bi se takodje pozivao vise puta itd), a dobili bi isti rezultat. Za resenje tog problema koristicemo cache, naime sacuvacemo poslednju izracunatu vrednost u objektu, i nju cemu koristiti nadalje. Uvodimo 2 nove metode MbCached() i MdCached() koje pozivaju Mb() ili Md() u zavisnosti da li je rezultat kesiran. Takodje moramo paziti da kada god se obrise bilo kakva observacija ili zakljucak vezan za taj proracun, da ga resetujemo. Tako smo dobili ogroman napredak u brzini kada su u pitanju veoma umrezeni sistemi.

Logiku obrade gresaka implementirao sam pomocu regex izraza i sopstvenih funkcija. Ideja je da korisnik moze da unese izraz, gde za svaku gresku u pozadinskoj sintaksi dobija upozorenje tako sto ce nedozvoljeni deo teksta biti predstavljen crvenom bojom, vecim i debljim fontom. Nakon toga korisnik moze da regulise izraz, ili da pritisne Ctrl + W dok ima fokus na tekstu i dobice kao rezultat potpuno cist izraz sa kojim dalje moze da nastavi.

Dvolikom na jedan od sadrzaja liste pravila sa strane mozete uci u dijalog za modifikaciju gde vaze ista pravila kao i za obican unos pravila, sve isto potpuno. Desnim klikom na sadrzaj brise taj sadrzaj iz liste.

**Najbitnije Klase i metode koriscenje u programu:**

public abstract class Observation

{

public abstract double Md();

public abstract double Mb();

public abstract string DisplayName();

public virtual double MbCached();

public virtual double MdCached();

public double Cf();

}

public class ConcreteObservation : Observation, INotifyPropertyChanged

{

public double ObservedFactor; // faktor pozivitvnog opazanja

public double NotObservedFactor;// faktor negativnog opazanja

// lista pravila u kome je trenutna observacija posmatrana kao zakljucak

public List<Rule> ConclusionRuleList;

// lista pravila u kome je trenutna observacija posmatrana kao konkretna observacija

public List<Rule> ObservationRuleList;

public double CummulativeM;// kumulativna formula Cum = P(a) + P(b) - P(a)\*P(b)

public double MbStrength;// formula za jacinu opazanja

public double MdStrength;// -||-

}

// klasa sluzi kao unikat koji sadrzi referncu na MainWindow klasu pomocu koje pristupamo graficki komponentama i statickim poljima MainWindow-a

public class Core{};

// klasa glavnog prozora koja sadrzi glavne GUI komponente

public partial class MainWindow : Window

{

// klasa koja sluzi kao wrapper standardne klase Semaphore, koji ima dodatnu opciju koja predstavlja mod u kom se semafor nalazi: Step ili Finish

public class MySemaphore;

// metoda koja vraca FlowDocument objekat u kom se nalazi parsiran i obojen izraz,

kao i vrednost uspesnosti parsiranja(boolean)

private Tuple<FlowDocument, bool> ValidateExpression(string expression)

// kolekcija koja sadrzi sve observacije

private static readonly List<ConcreteObservation>

Observations = new List<ConcreteObservation>();

// kolekcija pravila

public static readonly TrulyObservableCollection<Rule>

Rules = new TrulyObservableCollection<Rule>();

// kolekcija koja sadrzi sve trenutne konkretne observacije, tj objekte koji nijednom nisu referisani kao zakljucak

private static readonly TrulyObservableCollection<ConcreteObservation> ConcreteObservations = new TrulyObservableCollection<ConcreteObservation>();

}

public partial class Rule : INotifyPropertyChanged

{

public Observation Observation;

public ConcreteObservation Conclusion;

// lista svih observacija koje se nalaze u izrazu Observation

public List<ConcreteObservation> ConcreteObservations;

public double ObservedConclusionFactor; // faktor zakljucka u tom pravilu

}

// klasa koja se koristi kao atomicni deo pri parsiranju izraza iz infix u postfix

public class AtomicExpression;

public class PostfixConverter

{

// metoda za konverziju Pravila

public static Rule RuleConversion(string ruleString);

// metoda za konverziju Zakljucka

public static ConcreteObservation ConclusionConversion(string conclusionString, double factor);

// metoda za konverziju Observacije

public static Tuple<Observation, List<ConcreteObservation>> ObservationConversion(List<AtomicExpression> list);

// metoda za infix -> postfix

public static List<AtomicExpression> Postfix(string expression);

// provera balansiranosti zagrada u izraza pomocu steka

public static List<Tuple<int,int>> CheckParenthesis(string expression);

// provera illegalnih kombinacija operatora u izrazu, pomocu regexa

public static List<Tuple<int, int>> IllegalOperators(string expression);

// provera ilegalnih kombinacija reci u izrazu, pomocu regexa

public static List<Tuple<int, int>> InvalidWordCombinations(string expression);

// provera svih invalidnih kombinacija, poziva InvalidWord i IllegalOperators

public static List<Tuple<int,int>> CheckInvalidCombinations(string expression);

// metoda koja uzima pocetni izraz(string), i vraca string kao pocetni, samo razdovjen

jednim razmakom izmedju svakog znacajnog dela, u nasem slucaju {')', '(', 'ILI', 'I'}

public static string Tokenize(string sentence);

// metoda koja se koristi za infix -> postfix, vraca prioritet unetog operatora

private static int Priority(string x)

}

Ovo je samo tracak izvornog koda koji sam izdvojio kako bi najbrze i najlakse objasnio organizaciju sistema. Naravno pored ovih klasa postoje i pomocne klase, koje trenutno nisu dovoljno bitne da bi ih naglasio, ali se svejedno nalaze u izvornom kodu.

Simovic Djordje 2013/0162

Softversko Inzenjerstvo