Informe del Proyecto de Simulación II

Carlos Bermudez Porto C.BERMUDEZ@ESTUDIANTES.MATCOM.UH.CU
Grupo C412

Tutor(es): Dr. Yudivian Almeida, *Universidad de La Habana*

1. Características del Sistema de Inferencia e Implementación

Para la implementación del Sistema difuzo se tuvieron en cuenta diferentes definiciones de la lógica difusa. Estas definiciones fueron implementadas usando el lenguaje de programación *Python*. A continuación se explicarán las principales componentes, asi como detalles sobre su implementación:

1.1 Conjunto Difuso

Como base de todo el sistema se encuentra el conjunto difuso. Este cuenta con 4 propiedades principales:

1. Nombre: Este será el identificador de dicho conjunto.
2. Función de Membresía: Esta será la función que describa el grado pertenencia a dicho conjunto de un elemento x cualquiera.
3. Conorma: La función utilizada para realizar la operación de unión sobre este conjunto.
4. Norma: La función utilizada para realizar la operación de intersección sobre este conjunto.

Un ejemplo de dicha estructura en una clase de *Python* sería la siguiente:

```
class FSet:
   def __init__(self, name, membership,
        umethod, imethod):
      self.name = name
      self.membership = membership
      self.umethod = umethod
      self.imethod = imethod

   def __add__(self, other):
      # ...

   def __mul__(self, other):
      #...
```

Como en se observa en el código anterior las funciones de unión e intersección fueron implementadas como las operaciones de suma y multiplicación, respectivamente, de un objeto de *Python*.

El código de la clase anterior se puede ver en el archivo *fuzzy_set.py*.

1.2 Funciones de Membresía

La función de membresía es la principal característica de los conjuntos difusos anteriormente explicados. Estas funciones son definidas a través de la clase de *Python* siguiente:

```
class MembershipFunc:
   def __init__(self, ipoints, func):
      self.ipoints = ipoints
      self.func = func

   def __call__(self, x):
      return self.func(x)
```

La propiedad *func* contiene a la propia función de membresía que se define en la lógica difusa.

Por su parte la propiedad *ipoints* es una lista que contiene los puntos importantes para la función, los cuales son asumidos como valores reales. Ejemplo de estos puntos serían los límites analizados por esta función.

Si bien hasta este punto todas las propiedades permitían elegir un dominio cualquiera para los conjuntos difusos, la propiedad *ipoints* fuerza que el dominio este restringido a los reales. Esto se hace necesario para el correcto funcionamiento de algunas de las funciones de desfusificación que se verán más adelante.

1.2.1 Funciones de Membresía Básicas

Como funciones básicas de membresía fueron implementadas se implementaron métodos que construyen las funciones más utilizadas en la lógica difusa. Una lista de las funciones implementadas fue:

1. Función Triángular
2. Función Trapezoidal
3. Función Gaussiana
4. Función smf
5. Función zmf

El código de la clase anterior, asi como de las funciones básicas, se puede ver en el submódulo *functions*.

1.3 Variables Lingüísticas

Para la representación de las variables lingüísticas se utilizó la siguiente clase de *Python*:

```
class Variable:
   def __init__(self, name, *descriptors):
      self.name = name
      self.descriptors = dict()
      for fset in descriptors:
         self.descriptors[fset.name] = fset

   def fuzzify(self, x, dname):
      # ...
```

La propiedad *name* se refiere al nombre de la variable. La propiedad *descriptors* es un diccionario con todos los conjuntos difusos que describen a la variable. Un ejemplo sería $Variable('Persona', conjunto\_joven, conjunto\_viejo)$ donde los conjuntos viejo y joven son los que describen a esa persona.

Adicionalmente las variables cuentan con la función $fuzzify$ que permite dado un valor y el nombre de uno de los descriptores calcular su grado de pertenencia al conjunto con ese nombre.

El código de la clase anterior se puede ver en el archivo *variable.py*.

1.4 Reglas

Las Reglas en si varían en su implementación dependiendo del tipo de método de inferencia al que se vayan a usar, por ejemplo en los métodos de *Larsen* y de *Tsukamoto* la aplicación de dichas reglas a la entrada produce como resultado un conjunto o un valor respectivamente. Sin embargo para su implementación todas tuvieron como base la siguiente clase:

```
class Rule:
   def __init__(self, antecedent):
      self.antecedent = antecedent

   def eval(self, values:dict):
      raise NotImplementedError()
```

Sobre esta clase base se puede deducir que toda regla tendrá un antecedente, contenido en la propiedad *antecedent*. Y un método *eval* el cual recivirá los valores de la entrada de cada variable en el antecedente, el cual será evaluado para luego conformar y devolver el resultado de dicha regla.

1.4.1 Antecedentes

Para el manejo y evaluación de los antecedentes se implementó una clase base Antecedente, de la heredan 3 nuevas clases. De ellas dos son para manejar las relaciones de $\wedge$ y $\vee$ entre antecedentes, siendo la unidad básica de los antecedentes las expresiones $var\ is\ A$ donde var es una variable lingüística y A es un conjunto disfuso que la describe.

La implementación utilizada para estas clases es similar a la siguiente:

```
class Antecedent:
   def __and__(self, other):
      return AndAntecedent(self, other)

   def __or__(self, other):
      return OrAntecedent(self, other)

   def eval(self, values:dict):
      raise NotImplementedError()

class BinaryAntecedent(Antecedent):
   def __init__(self, left:Antecedent,
       right:Antecedent):
      self.left = left
      self.right = right

class AndAntecedent(BinaryAntecedent):
   def eval(self, values:dict):
      return min(self.left.eval(values),
          self.right.eval(values))

class OrAntecedent(BinaryAntecedent):
   def eval(self, values:dict):
      return max(self.left.eval(values),
          self.right.eval(values))

class IsStatementAntecedent(Antecedent):
   def __init__(self, var:Variable,
       descriptor:str):
      self.var = var
      self.desc = descriptor

   def eval(self, values:dict):
      return
          self.var.fuzzify(values[self.var.name],
          self.desc)
```

1.5 Métodos de Inferencia

De los 4 métodos estudiados se decidió implementar los métodos de Mamdani y de Larsen. Debido a la alta similaridad entre ellos se decició implementar una clase base que encapsulara el comportamiento de ambos, asi como para sus reglas ya que ellas solo varían en el conjunto resultante de la evaluación de las mismas. La implementación común de las reglas es la siguiente:

```
class Defuzzy_Rule(Rule):
   def __init__(self, antecedent:Antecedent,
       con_var:list, con_desc:list):
      super(Defuzzy_Rule,
          self).__init__(antecedent)
      self.con_vars = con_var
      self.con_descs = con_desc

   def merge(self, value, fset:FSet):
      raise NotImplementedError()

   def eval(self, values:dict):
      result = self.antecedent.eval(values)
      transformed = dict()
      for con_var, con_desc in
          zip(self.con_vars, self.con_descs):
```

```
                to_trans:FSet =
                    con_var.descriptors[con_desc]
                fset = self.merge(result, to_trans)
                transformed[con_var.name] = fset
            return transformed
```

Esta nueva regla agrega como propiedad las listas con_var y con_desc las cuales contendran ambas en la posición i la variable y el conjunto que la describe respectivamente en el consecuente de dicha regla. El resultado de evaluar la regla será un diccionario donde a cada nombre de variable en el consecuente se le asociará el conjunto de resultante de aplicar la función *merge* sobre ese conjunto que la describe y el valor resultante de evaluar el antecedente, dando esto como resultado un nuevo conjunto.

Para la implementación especifica de las reglas de los métodos de Mamdani y de Larsen solo hizo falta sobreescribir la función *merge* correcta para cada uno. Para Mamdani una que devuelva el conjunto cortado por el valor y en el caso de Larsen el conjunto escalado.

La clase en común para ambos métodos es la siguiente:

```
class Defuzzy_Model:
   def __init__(self, rules:list):
      self.rules = rules

   def infer(self, input_values:dict,
       def_method=centroid_defuzzification):
      final =
          self.rules[0].eval(input_values)
      for rule in self.rules[1:]:
         update = rule.eval(input_values)
         for var_name in update:
            final[var_name] =
                final[var_name] +
                update[var_name]
      for var_name in final:
         final[var_name] =
             def_method(final[var_name])
      return final
```

Ambos métedos contarán con la propiedad rules que contiene a todas las reglas que se usarán. El método $infer$ dado el diccionario con los valores de la entrada calculará el resultado correspondiente de cada regla y los cuales se unirán todos en otro diccionario, luego se desfusificarán cada uno con el método correspondiente.

1.5.1 Desfusificación

Para la desfusificación se implementaron varios métodos conocidos. Estos fueron:

1. Bisección
2. Centroide
3. Mínimo de los Máximos
4. Media de los Máximos
5. Máximo de los Máximos

El código de la clases anteriores se puede ver en los submódulos *inference* y *defuzzification*.

2. Problema Propuesto

El problema propuesto a resolver es el siguiente. Se quiere saber la operación a realizar por una presa automática en la desembocadura de un río para evitar que ocurran inundaciones en las tierras bajas que se encuentran río adentro, esto dado el estado de diferentes variables meteorológicas y el la diferencia de altura del nivel del mar con su estado normal.

Las Variables que se usaron son:

1. Temperatura(T) en F°. Los conjuntos que la describen son:
   - Frio
   - Templado
   - Caliente
2. Presión Atmosférica(PA) en hPa. Los conjuntos que la describen son:
   - Baja
   - Alta
3. Humedad del Aire(HA) en %. Los conjuntos que la describen son:
   - Baja
   - Promedio
   - Alta
4. Nivel del Mar(NM) en m. Los conjuntos que la describen son:
   - Normal
   - Medianamente Alto
   - Alto
5. Operacíon de la Presa(OP). Los conjuntos que la describen son:
   - Abrir
   - Cerrar

Las reglas usados segun estas variables son fueron las siguientes 5:

1. Si NM es $Normal$ entonces OP es $Abrir$.
2. Si NM es $Alto$ entonces OP es $Cerrar$.
3. Si T es $Alta \wedge HA$ es $Alta \wedge PA$ es $Alta$ entonces OP es $Cerrar$.
4. Si (HA es $Promedio \vee HA$ es $Baja$) $\wedge$ T es $Alta$ $\wedge$ PA es $Baja \wedge NA$ es $MedianamenteAlto$ entonces OP es $Abrir$.
5. Si (HA es $Baja \vee T$ es $Frio \vee PA$ es $Alta$) $\wedge$ NA es $MedianamenteAlto$ entonces OP es $Abrir$.

El código que describe este problema utilizando las herramientas antes descritas puede ver se en el archivo *main.py*.

3. Resultados

Se ejecutó el sistema para los siguientes casos siendo estos algunos de sus resultados:

Table 1: Resultados

T	PA	HA	NM	$Mamdani$	$Larsen$
70	989	89	2.5	*Cerrar*	*Cerrar*
58	1005	65	1	*Abrir*	*Abrir*
50	1000	70	3	*Cerrar*	*Cerrar*
65	992	89	2	*Abrir*	*Abrir*

Según estos resultados el Sistema pareció comportarse correctamente de acuerdo a las reglas definidas. De todas formas para una aproximación precisa del problema se debería contar con un experto en el tema para la correcta definición de las variables, la asignación de las funciones de membresía más correctas así como la definición de las reglas asociadas. Tambiíen se pudo observar la gran capacidad que tienen los sitemas de inferencias para dar soluciones aproximadas a problemas donde un algoritmo de solución no sea fácil de implementar o que de respuesta de la forma adecuada.

References

[1] Enlace al Proyecto en Github. URL: https://github.com/cbermudez97/fuzzy-logic-project.