Systems Theory - Homework 4

Ryan Spangler

February 1, 2011

Chapter 5

I spent all my time writing this fuzzy logic program so didn't get to the questions for chapter 5. However, I am hoping my fuzzy program is spectacular enough to carry me through.

Fuzzy Excellence

For this problem I wrote some python code to implement fuzzy logic, which turned this question into a straightforward application of querying the fuzzy logic oracle:

```
variables = {
  'excellent': {8:0.2,9:0.6,10:1.0},
  'good': {6:0.1,7:0.5,8:0.9,9:1.0,10:1.0},
  'fair': {2:0.3,3:0.6,4:0.9,5:1.0,6:0.9,7:0.5,8:0.1},
  'bad': {1:1.0,2:0.7,3:0.4,4:0.1}
}

class FuzzyNode:
  def __init__(self, key=''):
    self.key = key
    self.membership = {}
    self.remaining = 0
    self.operands = 0

  def evaluate(self, context):
    return self

  def complete(self):
```

```
    return self.remaining == 0

  def keyTree(self):
    return self.key

  def __repr__(self):
    return self.key

class FuzzyVariable(FuzzyNode):
  def __init__(self, key, membership, context=None):
    self.key = key
    self.membership = membership
    self.remaining = 0
    self.operands = 0

    if not context:
      self.context = self.membership.keys()
    else:
      self.context = context

  def keyContext(self, keys):
    self.context = keys

  def contains(self, key):
    return self.membership.has_key(key)

  def member(self, key):
    if self.contains(key):
      return self.membership[key]
    else:
      return 0.0

  def fuzzNot(self):
    notted = {}
    for key in self.context:
      value = 0.0
      if self.membership.has_key(key):
        value = self.membership[key]
      if value < 1.0:
        notted[key] = 1.0 - value

    result = FuzzyVariable('not '+self.key, notted, self.context)
    return result

  def fuzzVery(self):
```

```
    veryd = {}
    for key in self.membership.keys():
      veryd[key] = pow(self.membership[key], 2)

    result = FuzzyVariable('very '+self.key, veryd, self.context)
    return result

  def fuzzBinary(self, other, operation, combine):
    anded = {}
    for key in self.context:
      value = combine(self.member(key), other.member(key))
      if value > 0.0:
        anded[key] = value

    result = FuzzyVariable(self.key+' '+operation+' '+other.key,
                           anded, self.context)
    return result

  def fuzzAnd(self, other):
    return self.fuzzBinary(other, 'and', lambda x,y: min([x,y]))

  def fuzzOr(self, other):
    return self.fuzzBinary(other, 'or', lambda x,y: max([x,y]))

  def fuzzBut(self, other):
    return self.fuzzBinary(other, 'but', lambda x,y: min([x,y]))

  def fuzzTo(self, other):
    return self.fuzzBinary(other, 'to', lambda x,y: max([x,y]))

  def evaluate(self, context):
    return self

class FuzzyFunction(FuzzyNode):
  def __init__(self, key, function, operand=None):
    self.key = key
    self.function = function
    self.operand = operand or FuzzyNode()
    self.remaining = 0 if operand else 1
    self.operands = 1

  def bind(self, operand):
    self.operand = operand
    self.remaining = 0
```

```
  def evaluate(self, context):
    return self.function(self.operand.evaluate(context), context)

  def keyTree(self):
    return '(' + self.key + ' ' + self.operand.keyTree() + ')'

class FuzzyOperator(FuzzyNode):
  def __init__(self, key, operator, head=None, tail=None):
    self.key = key
    self.operator = operator
    self.head = head or FuzzyNode()
    self.tail = tail or FuzzyNode()

    self.remaining = 0
    self.operands = 2
    if not head: self.remaining += 1
    if not tail: self.remaining += 1

  def bind(self, target):
    if self.remaining == 2:
      self.head = target
    else:
      self.tail = target

    if self.remaining > 0:
      self.remaining -= 1

  def evaluate(self, context):
    return self.operator(self.head.evaluate(context),
                         self.tail.evaluate(context),
                         context)

  def keyTree(self):
    return '(' + self.head.keyTree()
         + ' ' + self.key + ' '
         + self.tail.keyTree() + ')'

class Fuzzy:
  def __init__(self, variables):
    self.variables = {}
    self.keys = set()

    self.lexicon = {
      'not': lambda v: FuzzyFunction('not', lambda x, context: x.fuzzNot()),
      'very': lambda v: FuzzyFunction('very', lambda x, context: x.fuzzVery()),
```

```
      'and': lambda v: FuzzyOperator('and', lambda a, b, context: a.fuzzAnd(b)),
      'or': lambda v: FuzzyOperator('or', lambda a, b, context: a.fuzzOr(b)),
      'but': lambda v: FuzzyOperator('but', lambda a, b, context: a.fuzzBut(b)),
      'to': lambda v: FuzzyOperator('to', lambda a, b, context: a.fuzzTo(b))
    }

    for v in variables.keys():
      self.variables[v] = FuzzyVariable(v, variables[v])
      self.keys = self.keys.union(variables[v].keys())

    for v in self.variables.keys():
      self.variables[v].keyContext(self.keys)

  def variable(self, key):
    return self.variables[key]

  def fuzzy(self, key):
    if self.lexicon.has_key(key):
      return self.lexicon[key](key)
    elif self.variables.has_key(key):
      return self.variables[key]

  def parse(self, statement):
    tokens = statement.split(' ')
    complete = []
    incomplete = []
    seen = []

    while len(tokens) > 0:
      head = tokens[0]
      tail = tokens[1:]
      focus = self.fuzzy(head)

      while not focus.complete()
            and len(seen) > 0
            and focus.operands == 2:
        target = seen.pop()
        focus.bind(target)
        if target in complete:
          complete.remove(target)

      while focus.complete() and len(incomplete) > 0:
        incomplete[-1].bind(focus)
        focus = incomplete.pop()
```

```
      stack = complete if focus.complete() else incomplete
      stack.append(focus)

      if not focus in seen:
        seen.append(focus)

      tokens = tail

    return (complete, incomplete)

  def evaluate(self, statement):
    tree, incomplete = self.parse(statement)

    if len(incomplete) > 0:
      for conundrum in incomplete:
        print 'could not parse ' + conundrum.key
        print incomplete.membership

    return tree[0].evaluate(self)

theory = Fuzzy(variables)

def parse(statement):
  return theory.evaluate(statement)

def output(statement):
  print statement
  print parse(statement).membership
  print

output('not bad but not very good')
output('good but not excellent')
output('fair to excellent')
```

The last three statements print the membership values for the composite statements. These are:

not bad but not very good:
{2: 0.3, 3: 0.6, 4: 0.9, 5: 1.0, 6: 0.99, 7: 0.75, 8: 0.19}
good but not excellent:
{6: 0.1, 7: 0.5, 8: 0.8, 9: 0.4}
fair to excellent:
{2: 0.3, 3: 0.6, 4: 0.9, 5: 1.0, 6: 0.9, 7: 0.5, 8: 0.2, 9: 0.6, 10: 1.0}

All of these seem to fit fairly well with their intuitive notions. One of the benefits of this approach is that "fuzzy and" and "fuzzy or" are both almost identical except for "or" uses max() where "and" uses min(). This similarity is reflected in the fuzzyBinary() function above which also takes a supplied "combine" function which produces a result based on the values of the respective membership functions of the two fuzzy variables being compared. This allows for a whole range of logical functions based on two fuzzy sets beyond just and and or.

For instance, rather than max or min, what if the supplied combination function is multiplication? This is a sort of fuzzy convolution, and could be useful in fuzzy applications.

The program is also a natural language parser for the given system (or any given fuzzy logic system) which bases its translation on whatever linguistic variables are supplied at the beginning. The program converts statements in natural language into executable trees reflecting values in the given fuzzy logic system.

In addition to the notion of fuzzy variables there is also added the idea of a fuzzy function and fuzzy operators. A fuzzy function takes a fuzzy variable (or function or operator) and returns another fuzzy variable. A fuzzy operator is a fuzzy function which has two inputs and produces a single fuzzy variable output. In this way the input statement is converted into a tree of nested fuzzy nodes, with fuzzy operators as branches, fuzzy functions as extenders and fuzzy variables as leaves.