Informatics 2D Tutorial 4

First-Order Logic and Generalised Modus Ponens*

Week 5

1 The Crop Allocation Problem

Consider the following problem in bio-dynamic farming (where some crops grow better next to particular crops)[1] for the specific land division shown in Figure 1.

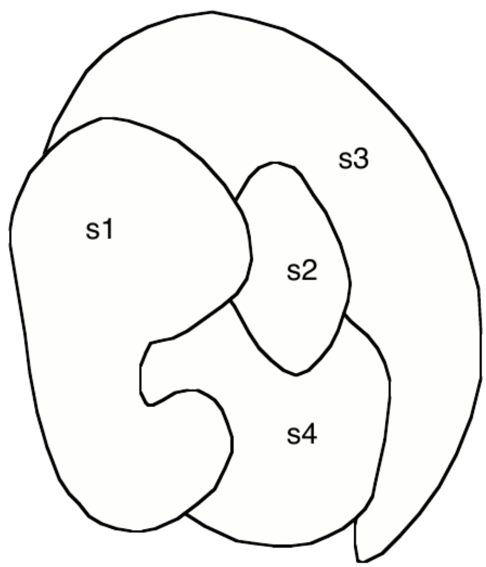

Figure 1: The Bio-Dynamic Farming Problem.

The figure shows the allocation of a piece of land for planting four different crops using the constraints of bio-dynamic farming. In this kind of farming, the idea is that there are groups of crops that develop better if set in particular arrangements. Also the balance of nutrients in the soil is used to decide what to plant where. Here are the constraints according to the current levels of nutrients in the soil:

*Credits: Kobby Nuamah, Michael Rovatsos

[1]Adapted from an original problem set by Mellish & Fisher.

1. Sector 1 (s1) can be planted with one of the following crops: {cabbage, kale, broccoli, cauliflower}

2. Sector 2 (s2) can be planted with one of the following crops: {cabbage, kale, broccoli}

3. Sector 3 (s3) can be planted with one of the following crops: {kale}

4. Sector 4 (s4) can be planted with one of the following crops: {kale, broccoli}

The constraint here is that we do not want two sectors that are adjacent to each other to be planted with the same crops.

How does this look when expressed as a constraint satisfaction problem (CSP)? What are the stages that the AC-3 algorithm goes through in obtaining arc consistency for this example? (see Figure 2 for the AC-3 algorithm)

```
function AC-3( csp) returns false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components (X, D, C)
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (Xi, Xj) ← REMOVE-FIRST(queue)
    if REVISE(csp, Xi, Xj) then
      if size of Di = 0 then return false
      for each Xk in Xi.NEIGHBORS - {Xj} do
        add (Xk, Xi) to queue
  return true
---------------------------------------------------------------------
function REVISE( csp, Xi, Xj) returns true iff we revise the domain of Xi
  revised ← false
  for each x in Di do
    if no value y in Dj allows (x,y) to satisfy the constraint between Xi and Xj then
      delete x from Di
      revised ← true
  return revised
```

Figure 2: The AC-3 algorithm.

Answer:

Variables: s1, s2, s3 and s4

Domains:
domain(s1, [cabbage, kale, broccoli, cauliflower])

domain(s2, [cabbage, kale, broccoli])

domain(s3, [kale]),

domain(s4, [kale, broccoli]).

Constraints: $s_1 \neq s_2 \neq s_3 \neq s_4$

AC-3:

The initial queue is:

[s1→s2, s1→s3, s1→s4, s2→s3, s2→s4, s3→s4, s2→s1, s3→s1, s4→s1, s3→s2, s4→s2, s4→s3]

The stages of the following can be shown incrementally on the board. You will want to show the values lists (domains) for the variables being updated as this all takes place.

queue is

[s1→s2, s1→s3, s1→s4, s2→s3, s2→s4, s3→s4, s2→s1, s3→s1, s4→s1, s3→s2, s4→s2, s4→s3]

Revise (s1→s2) = False

queue is

[s1→s3, s1→s4, s2→s3, s2→s4, s3→s4, s2→s1, s3→s1, s4→s1, s3→s2, s4→s2, s4→s3]

Revise (s1→s3) = True, domain(s1,[cabbage,broccoli,cauliflower]) add []

queue is

[s1→s4, s2→s3, s2→s4, s3→s4, s2→s1, s3→s1, s4→s1, s3→s2, s4→s2, s4→s3]

Revise (s1→s4) = False

queue is

[s2→s3, s2→s4, s3→s4, s2→s1, s3→s1, s4→s1, s3→s2, s4→s2, s4→s3]

Revise (s2→s3) = True, domain(s2,[cabbage,broccoli]) add [s1→s2]

queue is

[s2→s4, s3→s4, s2→s1, s3→s1, s4→s1, s3→s2, s4→s2, s4→s3, s1→s2]

Revise (s2→s4) = False

queue is

[s3→s4, s2→s1, s3→s1, s4→s1, s3→s2, s4→s2, s4→s3, s1→s2]

Revise (s3→s4) = False

queue is

[s2→s1, s3→s1, s4→s1, s3→s2, s4→s2, s4→s3, s1→s2]

Revise (s2→s1) = False

queue is

[s3→s1, s4→s1, s3→s2, s4→s2, s4→s3, s1→s2]

Revise (s3→s1) = False

queue is

[s4→s1, s3→s2, s4→s2, s4→s3, s1→s2]

Revise (s4→s1) = False

queue is

[s3→s2, s4→s2, s4→s3, s1→s2]

Revise (s3→s2) = False

queue is

[s4→s2, s4→s3, s1→s2]

Revise (s4→s2) = False

queue is [s4→s3, s1→s2]

Revise (s4→s3) = True, domain(s4,[broccoli]) add [s1→s4, s2→s4]

queue is

[s1→s2, s1→s4, s2→s4]

Revise (s1→s2) = False

queue is

[s1→s4, s2→s4]

Revise (s1→s4) = True, domain(s1,[cabbage,cauliflower]) add [s2→s1, s3→s1]

queue is

[s2→s4, s2→s1, s3→s1]

Revise (s2→s4) = True, domain(s2,[cabbage]) add [s1→s2, s3→s2]

queue is

[s2→s1, s3→s1, s1→s2, s3→s2]

Revise (s2→s1) = False

queue is [s3→s1, s1→s2, s3→s2]

Revise (s3→s1) = False

queue is

[s1→s2, s3→s2, s4→s2]

Revise (s1→s2) = True, domain(s1,[cauliflower]) add [s3→s1, s4→s1]

queue is

[s3→s2, s4→s2, s3→s1, s4→s1]

Revise (s3→s2) = False

queue is

[s4→s2, s3→s1, s4→s1]

Revise (s4→s2) = False

queue is [s3→s1, s4→s1]

Revise (s3→s1) = False

queue is [s4→s1]

Revise (s4→s1) = False

queue is []

2 First-Order Logic

Part 1: Represent the following sentences in first-order logic. You will have to define a vocabulary (which should be consistent between sentences).

1. Some students took French in spring 2001.
2. Every student who takes French passes it.
3. Only one student took Greek in spring 2001.
4. The best score in Greek is always higher than the best score in French.
5. There is a male barber who shaves all the men who do not shave themselves.

Part 2: Write down a first-order logic sentence such that every world in which it is true contains exactly one object.

Answer:

Part 1: An example vocabulary might be:

- *French* - a constant denoting the subject French
- *Greek* - a constant denoting the subject Greek
- *student*/1 - a unary relation, student(x) iff constant x is a student (unnecessary if the took relation is defined to only apply to students)
- *took*/3 - a ternary relation, $took(x, y, t)$ iff x took the subject y during time interval t (there are alternatives to having time as an argument to the $took$ relation. You could represent this as a course event, e.g. $course(e) \wedge during(e, t) \wedge took(x, e) \wedge subject(y, e)$)
- *pass*/2 - a binary relation, $pass(x, y)$ iff x passes the subject y
- *Spring2001* - a constant denoting the time interval spring 2001
- *bestScore*/2 - a binary function which identifies the best score in a subject during a given time interval.
- *greaterThan*/2 - a binary relation which has the same meaning as $>$
- *equals*/2 - a binary relation which has the same meaning as $=$ (note that it's acceptable to assume that we are using first-order logic with equality, provided that the student knows what this means).
- *numOfStudents*/2 - a binary function which identifies the number of students in taking a subject during a given time interval.
- *barber*/1 - a unary relation, $barber(x)$ iff x is a barber.
- *shaves*/2 - a binary relation, $shaves(x, y)$ iff x shaves y.

Given this vocabulary you can represent the sentences as follows:

1. $\forall x.\, student(x) \wedge took(x; French; Spring2001)$
2. $\forall x, t:\ student(x) \wedge took(x; French; t) \Rightarrow pass(x; French)$ (this is slightly vague, since a student could fail and then pass on the second attempt)
3. $3equals(numOfStudents(Greek; Spring2001); 1)$ (the challenge here is to represent the cardinality of the set of students taking Greek during spring 2001). An alternative is $\exists x.\, student(x) \wedge took(x; Greek; Spring2001) \wedge (\forall y.\, took(y; Greek; Spring2001) \Rightarrow x = y)$. This formula asserts that there exists a student who took Greek in spring 2001 and if there is anything else which took Greek in spring 2001 then it must be this student. So it would be impossible to satisfy this sentence if less than one student took Greek in spring 2001, since we have asserted the existence of at least one student with this property. But also impossible to satisfy it if more than one student took Greek in spring 2001, since in that case there would be a y such that $took(y; Greek; Spring2001) \wedge y \neq x$.
4. $\forall t.\, greaterThan(bestScore(Greek, t), bestScore(French, t))$
5. $\exists x.\, barber(x) \wedge \forall y.\, \neg shaves(y; y) \Rightarrow shaves(x; y)$ – (almost) Russell's paradox, there is no barber with this property as if there was then it would be possible to prove that the $shaves(Barber, Barber)$ and $\neg shaves(Barber, Barber)$ (N.B. Russell's paradox is that there is a barber who shaves all $andonly$ the men who do not shaves themselves – this is an equivalence, $\Leftrightarrow$, rather than an implication)

Part 2: One possible sentence is $\forall x.\, P(x) \wedge \neg\exists x.\, x \neq A \wedge P(A)$; which means that for all objects property P holds and there are no objects not equal to object A such that property P holds. So if this sentence is true then there can only be one object, A, in the domain of interpretation. If there were any other objects then they would have to have property P and not have property P in order to satisfy this sentence, which is impossible. A simpler alternative is $\exists x \forall y.\, x = y$; which means that there exists an object such that all other objects are equivalent to this object, so there is only one unique object in the domain.

3 Most General Unifier (MGU)

The most general unifier (MGU) is the least constrained substitution that makes two clauses unify with each other. What is the MGU for each pair of clauses below? If there is no MGU, explain why.

The Unify algorithm in figure 3 (also in R&N Section 9.2, p.328.)

1. $p(A, B, B)$ and $p(x, y, z)$
2. $q(y, g(A, B))$ and $q(g(x, x), y)$
3. older(father(y), y) and older(father(x), John)
4. knows(father(y), y) and knows(x, x)

```
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} ∈ θ then return UNIFY(val, x, θ)
   else if {x/val} ∈ θ then return UNIFY(var, val, θ)
   else if OCCUR-CHECK?(var, x) then return failure
   else return add {var/x} to θ
```

Figure 3: Unification Algorithm.

Note that, constants are upper case (e.g. A, B) and variables are lower case (e.g. x, y, z).

Answers

1. x/A, y/B, z/B

2. Unification fails. Start with the (partial) substitution $y/g(x,x)$, then add x/A to get $y/g(x,x)$, x/A. At this point, one clause is $q(g(A,A), g(A,B))$, and the other $q(g(A,A), g(A,A))$. Since $q(A,A)$ cannot be unified with $q(A,B)$ unification fails.

3. $x/John$, $y/John$

4. Unification fails. Start with (partial) substitution $x/father(y)$. One clause is now $knows(father(y), y)$, and the other $knows(father(y), father(y))$. Unification fails here because we can't unify y and $father(y)$, due to the occurs check.