Agent-Based Systems
Tutorial 2 Solutions

Michael Rovatsos

Suggestions for solutions are printed in italics below each question

Q1 Formally define the function $new : D \times Per \rightarrow D$ that updates the agent's knowledge base for the vacuum-world example (using a schematic tabular representation or pseudo-code, if desired).
Solution suggestions: *This is a very simple exercise. A simple pseudo-code description of the new function is. (We assume that in the first time step the knowledge base contains information about being in (0,0) and facing north.)*

```
procedure new(list of facts D; percept P; new action A) outputs D'
    initialise D' to the empty set
    forall d in D {
      if d=In(x,y) then xpos = x;  ypos = y;
      else if d=Facing(f) then dir=f;
    }
    if A=forward then {
        if dir=north and ypos<2 then newposx=xpos; newposy=ypos+1;
        else if dir=west and xpos>0 then newposx=xpos-1; newposy=ypos;
        else if dir=east and xpos<2 then newposx=xpos+1; newposy=ypos;
        else if ypos>0 then newposx=xpos; newposy=ypos-1;
    }
    if A=turn then {
        newposx=posx; newposy=posy;
        if dir=north then dir=east;
        else if dir=east then dir=south;
        else if dir=south then dir=west;
        else dir=north;
    }
    add Facing(dir) to D';
    add In(newposx,newposy) to D';
    if P=dirt then add Dirt(posx,posy) to D';
end
```

You can point out that knowledge about the previous action is necessary and is missing from the definition of the "new" function in the lecture slides. The above definition assumes that the percept is just "dirt" or "null" and infers the direction from previous knowledge and own action. Alternatively, the direction might be available as a percept.

Q2 Suggest a compact and elegant decision making algorithm for the vacuum world using first-order logic that works for arbitrary grid sizes. You can use the usual quantifiers $\exists$ and $\forall$, equality $=$, integers and normal operations on them as well as a constant S

which denotes the size of the grid.
Solution suggestions: *Many different designs are possible, but the core insight should be that using rules that are as general as possible minimises the number of rules necessary to describe a strategy. The problem can be made a bit more interesting if we allow an additional action "move in random direction" to avoid having to specify conditions for each and every situation and to analyse whether the robot can get stuck. Here is a very simple example robot design with this additional non-deterministic action:*

$$\begin{aligned} \forall x\ \forall y\ .In(x,y) \wedge Dirt(x,y) &\Rightarrow Do(suck) \\ \forall x\ \forall y\ .In(x,y) \wedge (y = S \vee x = S) &\Rightarrow Do(turn) \\ \forall x\ \forall y\ .In(x,y) &\Rightarrow Do(randomMove) \end{aligned}$$

Attention should also be drawn to the fact that the order of these rules in the knowledge base matters in terms of optimal choices if the simple control loop is used that was shown in the lecture. You can discuss whether having mutually exclusive preconditions in the above rules solves this problem, and also what the importance of exhaustive preconditions is in this context.

Q3 The following specification describes the famous "Snow White" example in Concurrent MetateM:

$$\begin{aligned} &\mathsf{SnowWhite(ask)[give]}: \\ &\boldsymbol{\circledcirc} ask(x) \Rightarrow \Diamond give(x) \\ &give(x) \wedge give(y) \Rightarrow (x = y) \\ &\mathsf{eager(give)[ask]}: \\ &start \Rightarrow ask(eager) \\ &\boldsymbol{\circledcirc} give(eager) \Rightarrow ask(eager) \\ &\mathsf{greedy(give)[ask]}: \\ &start \Rightarrow \Box ask(greedy) \\ &\mathsf{courteous(give)[ask]}: \\ &((\neg ask(courteous)\ \mathcal{S}\ give(eager)) \wedge \\ &(\neg ask(courteous)\ \mathcal{S}\ give(greedy))) \Rightarrow ask(courteous) \\ &\mathsf{shy(give)[ask]}: \\ &start \Rightarrow \Diamond ask(shy) \\ &\boldsymbol{\circledcirc} ask(x) \Rightarrow \neg ask(shy) \\ &\boldsymbol{\circledcirc} give(shy) \Rightarrow \Diamond ask(shy) \end{aligned}$$

Describe what the programme does and trace its operation in a table for the first three time steps. For reference, the following table summarises the MetateM operators:

$\circledcirc\varphi$	φ is true tomorrow
$\boldsymbol{\circledcirc}\varphi$	φ was true yesterday
$\Diamond\varphi$	φ now or at some point in the future
$\Box\varphi$	φ now and at all points in the future
$\blacklozenge\varphi$	φ was true sometimes in the past
$\blacksquare\varphi$	φ was always true in the past
$\varphi\,\mathcal{U}\,\psi$	ψ some time in the future φ until then
$\varphi\,\mathcal{S}\,\psi$	ψ some time in the past, φ since then (but not now)
$\varphi\,\mathcal{W}\,\psi$	ψ was true unless φ was true in the past
$\varphi\,\mathcal{Z}\,\psi$	like "$\mathcal{S}$" but φ may have never become true

Solution suggestions: *An example run is shown in the table below (the first three steps are not enough to illustrate the system's behaviour as suggested in the question). Note that for readability some messages irrelevant for the "dwarves" are not shown in this table.*

time	*SnowWhite*	*eager*	*greedy*	*courteous*	*shy*
0		*ask(eager)*	*ask(greedy)*		*ask(shy)*
1	*ask(eager), ask(greedy)*		*ask(greedy)*		
2	*ask(greedy), give(eager)*		*ask(greedy)*		
3	*ask(greedy), give(greedy)*	*give(eager)*	*ask(greedy)*		
4	*ask(greedy), ask(shy)*	*ask(eager)*	*ask(greedy) give(greedy)*		
5	*ask(greedy), ask(eager)*		*ask(greedy)*	*ask(courteous)*	

The workings of this system are quite clear: the greedy agent will always ask for the resource, the eager agent will ask at the beginning and whenever it has just received it, the courteous agent will only ask if it hasn't asked since neither the greedy or the eager agent got it, and the shy agent will only ask if no other agent has just asked for it (in fact, the way the system is set up, the shy agent will never ask for the resource unless it does so in the very first step). SnowWhite will give the resource eventually to anyone, but can only provide it to one at a time. It is important to appreciate the difference between the behaviour intended for the system by the programmer here, and whether there is actually any way of satisfying all commitments and producing a run where all commitments have been satisfied.