# SET10117 Multi-Agent Tutorial Timetable Generation

Calum Hamilton

40205163

### Introduction:

In this coursework, I was assigned the task of prototyping a decentralised multi-agent system which would assign timetable slots to students based on their preferences. Timetable slots are initially assigned to student agents (acting on behalf of a student) who attempt to swap their slot until they receive preferential times.

### Design:

**The design choices are appropriate in all areas. Justification of design choices is excellent, possibly going beyond the taught material by considering issues such as scalability or applicability to a more realistic scenario.**

My program design and report will address the following five design requirements:

1. A meaningful and relevant ontology, allowing agents to communicate
2. A communication protocol, allowing agents to communicate without revealing their preferences
3. A utility function, allowing the student to know when they’re satisfied with their slots
4. A strategy to determine which exchange requests to make, which to accept and which to reject
5. A metric to evaluate system effectiveness in terms of satisfying student preferences

#### 1. Ontology Design:

I include my Ontological Relationships in Appendix A. and examine the Ontological elements here.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Element** | **Properties** | **Data Type** | **Restrictions** | **Type** |
| **Pleased With** | Student Identifier | AID | Mandatory | Predicate |
| **Timeslot** | Day  Time | Integer  Integer | Mandatory  Mandatory | Concept |
| **Tutorial Group** | **Timeslot**  Tutorial ID  Class Size | Ontology Element  String  Integer | Mandatory  Mandatory | Concept |
| **Swap Initial** | **Tutorial Group**  Agent From | Ontology Element  AID | Mandatory  Mandatory | Agent Action |
| **Swap Final** | **Swap Initial**  **Tutorial To**  Agent To | Ontology Element  Ontology Element  AID | Mandatory  Mandatory  Mandatory | Agent Action |
| **Unhappy Slot** | **Swap Initial** | Ontology Element |  | Agent Action |
| **Slots Requested** | ArrayList<**Swap Final**> | List<Ontology Elements> | Mandatory | Predicate |
| **Happy With** | ArrayList<**Swap Final**> | List<Ontology Elements> | Mandatory | Agent Action |
| **Message Board** | ArrayList<**Swap Initial**> | List<Ontology Elements> |  | Concept |
| **Slots Available** | **Message Board** | Ontology Element | Mandatory | Predicate |

#### 2. Communication Protocol:

There were several ways I envisaged the communication protocol for my multi-agent system:

* A free-for-all messaging system
* Students sending unwanted slots to an agent that broadcasted the slots to students
* Having a message board where slots could be posted and requested from

My biggest focus when planning the communication protocols, was allowing the most communication in the fewest messages. This included live messages and total messages.

My foremost concern with a free-for-all messaging system is that it would pass too many live messages– meaning that messages may not be received. Additionally, I was concerned it would create too many messages overall – especially if you the solution is scaled. It may be reasonable for 10 agents to send and request slots at the same time, but the number of messages sent is nn per iteration, where n is the number of agents. Plus, anyone who has been at a large Christmas gathering can tell you that if everyone is speaking to everyone else at the same time, it leads to inefficient communication.

An Agent receiving and broadcasting slots to Student Agents would help limit the number of messages that a free-for-all messaging system would cause. Allowing for one slot per iteration to be offered, the complexity of this is still 2n per timestep, where n is the number of agents (one message to the broadcaster from the sender, one message sent to other agents, one reply from each of the agents and one reply from the broadcaster). You could attempt to limit the messages further by having agents not respond if they’re not interested in the slot, but it still runs into a lot of the issues that the free-for-all messaging system would run into.

An underlying reason for these issues is because Student Agents aren’t allowed to expose their timeslot preferences (e.g. can only respond yes/no). Since Agents can’t expose their preferences, the receiver is unable to know which agents would be best benefitted, given a certain slot (can’t choose the result that maximises social welfare).

This led me to understand the best approach is to give a Student Agent a selection of slots and let them choose the slot best suited to them, as opposed to passing single slots and having them return yes or no. This also points towards a centralised approach – having the ability to store and pass multiple slots simultaneously.

I realised that the Communication Protocol would require a controller agent to limit the number of live messages and store a bank or ‘message board’ of unwanted slots. I started using a ‘Timetabling Agent’, to hold a list of slots for agents to swap.

I detail the Communication Protocol design in the Sequence Diagram in Appendix B.

#### 3. Utility Function:

There were several ways that I saw the agent utility being calculated:

* Mean utility
* Specialised calculation

Using the mean utility as the Agent Utility function appealed to me, since you could calculate a mean happiness for each tutorial and quit if it wasn’t above a threshold. The mean utility function worked very well on the 1-2 tutorial 1-3 group test cases. After testing the mean utility on my third test case, I realised I was still accepting Student Agents if they had a slot they couldn’t attend. I tried to fix this by raising and dropping the accepted threshold for ‘happy students’, but this didn’t solve the issue.

I then settled on a specialised calculation. It was adapted from the mean value of each slot, with a twist. The specialised calculation I created was:

Where *p* = Slot Preference (on a scale from 0 – 5). 0 means that the StudentAgent loves that slot and 5 means that the student can’t attend that slot. Since a student will never have >3/4 tutorials, -5 is a low enough value that the student can never accept the slots.

Output values (for a single example):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Slot Pref:** | 0 - loves the slot | 1 - likes the slot | 2 - indifferent | 3 - doesn't like | 4 - can't attend |
| **Value:** | 1.0 | 0.5 | 0.33° | 0.25 | -5 |

As you can see, slots the student loves and hates are heavily weighted.

For the utility function, I also found that if I started the target happiness at a reachable level, Student Agents would swap several slots and then exit, without considering there may be better slots available. This left less Student Agents in the environment, with less opportunity for Student Agents to swap bad slots. I tested various happiness thresholds and found that 1.2 with a reduction of 0.2 upon each time they were notified to request slots was a good level. It didn’t run for too long and allowed Student Agents that had selected optimal slots to exit the environment early.

#### 4. Strategy for Considering Requests and Exchanges to Make

In the system I designed, there were three opportunities for students to swap slots:

1. Informing the Timetable Agent of slots they were interested in
2. Agreeing/Refusing slots on the message board that other agents had requested
3. Posting new slots on the message board they didn’t want

I examine these more deeply below:

1. The student needed to evaluate its slots against slots on the message board. The student would then be able to select good slots/the best slot from and request those slots.  
   I limited the number of unique tutorials slots the student could request to one tutorial per Set ID. I found this a good balance between requesting too few slots each cycle and slowing down the system with a list of requests to be verified one-by-one. If an Agent could request multiple tutorials with the same Set ID, the Timetable Agent would need to verify if the first Agent was happy to swap, then the second if the first wasn’t happy etc.
2. I had Student Agents refuse swaps if the new slot fitness was worse or equal to their current. They would only accept a new slot if it was better than their current slot. This meant that Agents would post slots they didn’t like but wouldn’t swap them if the one offered was worse.
3. I originally had agents posting their worst slot and then had the Timetable Agent decline it if the slot was already present on the message board. This however meant that not enough slots were being posted to swap quickly and efficiently, and that Agents would still be stuck with slots they didn’t want. I updated this by having the Agents post slots they were unhappy with, that they hadn’t already posted – alternately post no slots.

#### 5. A Metric to Evaluate Overall Effectiveness of the System

I thought the best way to evaluate the system was a somewhat Utilitarianist approach in trying to understand the happiness of Agents across the system. I evaluate the ‘happiness’ of each Actor (Actor fitness) through a custom evaluation function, and this is what I want the system to maximise.

Since this is what I want to maximise, it seemed appropriate to average the custom evaluation function across Agents and get a Mean. The metric I’m using to evaluate overall system effectiveness is the mean value of the Utility Function – I’m calling Mean System Fitness. I examine this Mean System Fitness in each of my test cases I present below.

### Implementation

***This section must be supported by screenshots of each conversation in your communication protocol running from the JADE sniffer, and code listings for the agent and ontology classes in an appendix at the end of the report.***

My Implementation Report will address the following three requirements:

1. Show the utility calculation, where it decides which swap requests to accept and which to reject, and state where the student’s preferences are represented
2. State where in the code the timetabling agent ensures that each student attends exactly one tutorial for each module.
3. For each conversation in your communication protocol, you should reference the relevant screenshot from the JADE sniffer, and state which agent behaviours implement it.

#### 1. Student Utility Calculation and Preference Exchange:

### Testing

**Describe your test cases, and justify why you have chosen them. Present the results from running your system on each test case, and justify the output metrics that you have chosen**

**AND SHOW MEAN SYSTEM FITNESS**

**TOP MARK:**

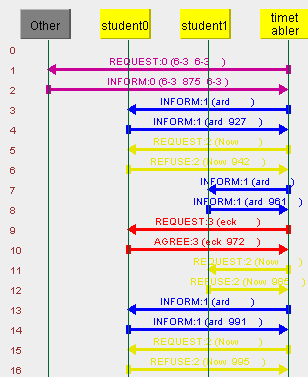
At least three test cases are chosen where their justification demonstrates an excellent understanding of the problem. The results are clearly presented with appropriate and clearly explained output metrics. The test cases clearly demonstrate how the system performs with increasing difficulty of the problem.

### Evaluation and Future Work:

1. How will the effectiveness of your system in will change as the problem becomes more difficult?
2. What are the advantages and disadvantages of taking a multi-agent systems approach to this problem?
3. In light of 1 and 2, suggest and justify an improvement to your system.

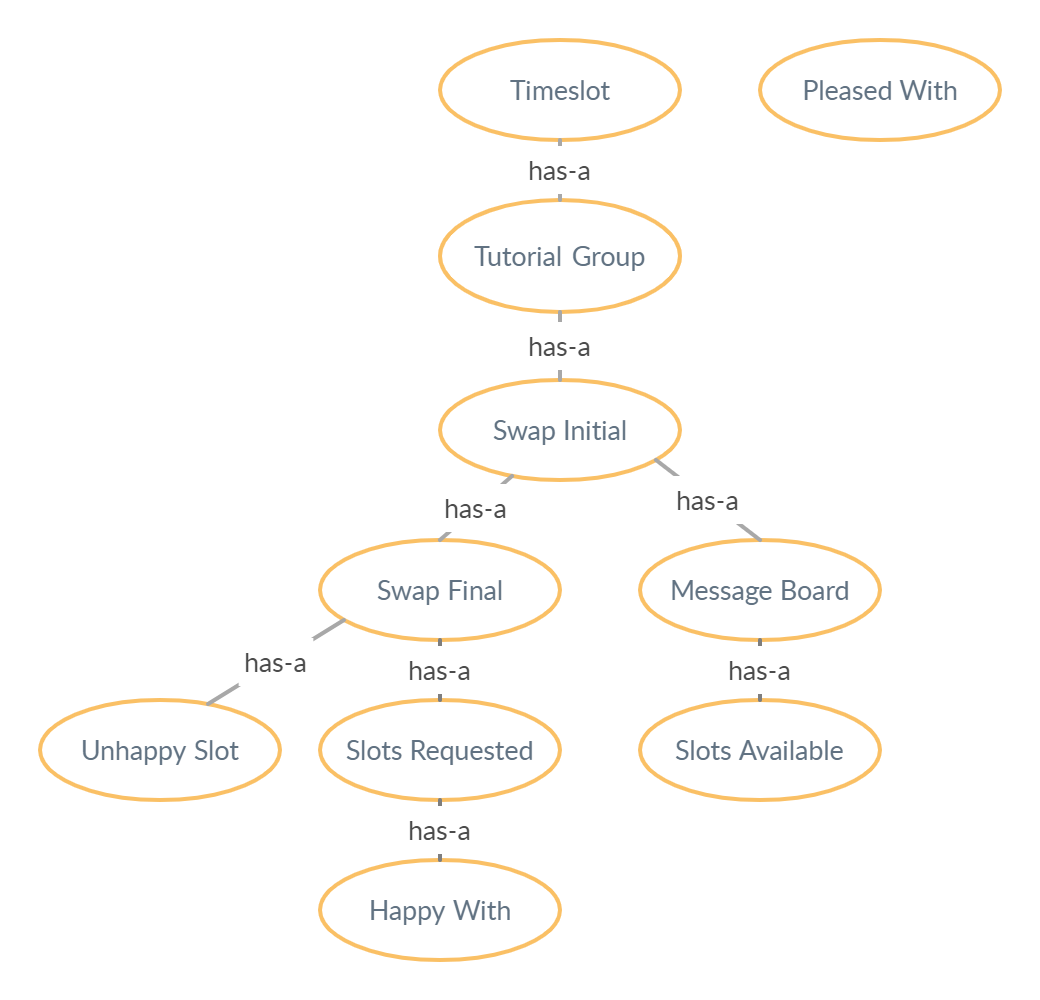
### References

1. S N, Omkar. (2003). Cricket team selection using genetic algorithm.
2. Ahmed, Faez & Deb, Kalyanmoy & Jindal, Abhilash. (2013). Multi-objective optimization and decision making approaches to cricket team selection. Applied Soft Computing. 13. 402–414. 10.1016/j.asoc.2012.07.031.



### Appendix:

#### Appendix A – Ontology Association



#### Appendix B – Message Sequence Diagram

