Video Description

In the video two teams of artificial agents play against each in a hostile environment known as Unreal Tournament (2004 edition). The objective of the match is kill members, of which there are four, of the opposite team 60 times. The team in blue are controlled by the UT game engine, while the team in red are controlled by an external system. The team in red win by a ~20 point margin.

The Engineering

The system used to control the red team is a layered architecture using a combination of machine learning, behaviour modules and multi-agent co-ordination.

Layer 1 – Machine Learning

At the bottom level a variety of machine learning techniques are implemented which model certain parts of the environment. In the example in the video only the fight and weapon models were used.

A K-Nearest Neighbours fight model with a Mahalanobis distance was used to generate a probability of surviving a confrontation with an enemy, given current situation (distance from enemy, current health, current weapon, current ammunition).

A weapon model was used to generate a utility for each weapon available in the level which was then used to select which weapon to go for and which of the currently held weapons to use. The modelling method was a heuristic rule for updating weapon utility based on outcomes of fights using that weapon.

Layer 2 – Behaviour Modules

The middle layer contained a selection of behaviour modules, built upon the machine learning layer, designed to represent basic behaviours. These could then be combined at higher levels to provide a full bot behaviour. The two modules used in the video example were explorer and exploiter.

The explorer module's behaviour was to run around picking up weapons. When an enemy became visible they would select the last weapon which they picked up and run towards the enemy firing at them.

The exploiter module's behaviour was to select the best weapon using the weapon model, go pick it up and then search for the enemy. If an enemy was visible they would first select the best weapon in their current possession and begin shooting at the enemy, before querying the fight model to decide whether to approach the enemy or not.

In the video only the explorer module was updating the models.

Layer 3 – Multi-Agent Co-ordination

The top layer combined these lower layers into a system for controlling the bots. To do this a variant of lightweight co-ordination calculus (LCC) , which I will now refer to as REA-LCC, was used. This was a reactive adaptation which selects a rule from the strategy, using prolog semantics, and uses the constraints in the rule as a way of turning on or off certain modules.

The formalisation used for the rules was:

```
Framework := {Clause,...}
      Clause := Agent :: Dn
        Agent := agent(Type,Id)
          Dn := Message | Dn then Dn | Dn or Dn | null <- C
   Message := M => Agent | M => agent <- C | M <= Agent | C <- M <= Agent
             C := Term | C and C | C or C
         Type := Term
            M := term
```

And the example used in the video was:

```
%%The stuck exploiter module, reassigns members to the correct module based on performance

a(exploit,sh)::engageModule(exploit) <-- dontReply(sh) <= a(_,_)
a(exploit,sh)::changeToExplore => a(exploit,Id) <-- engageModule(exploit) and performance(exploit,death)
a(exploit,sh)::changeToExploit => a(explore,Id) <-- engageModule(exploit) and performance(exploit,kill)
a(exploit,sh)::null<--engageModule(exploit)

%% The hunter module,

a(exploit,Id)::engageModule(exploit) <-- dontReply(ID) <= a(exploit,_) then changeToExplore <= a(exploit,ID)
a(exploit,Id)::engageModule(explore) and changeToRole(explore) <-- changeToExplore <= a(exploit,ID) then
                dontReply(ID) => a(exploit,IDS)
a(exploit,Id)::null<--engageModule(exploit)

%% The baseliner module
a(explore,Id)::engageModule(explore) <-- dontReply(ID) <= a(explore,_) then changeToExploit <= a(exploit,ID)
a(explore,Id)::engageModule(exploit) and changeToRole(exploit) <-- changeToExploit <= a(exploit,ID) then
                 dontReply(ID) => a(explore,IDS) then dontReply(ID) => a(exploit,ID)
a(explore,Id)::null<--engageModule(explore)
```

The idea of this strategy is that we have one bot who is always playing as an exploiter and several other bots which can play either as exploiters or explorers. When the 'sh' bot sees that the performance of the 'exploit' role is bad he will send out messages which will turn the first responding explorer bot into an exploiter. They will then send back a message which stops him sending out any more messages until that one decays (Messages are time limited, providing some weak notion of temporality to the system).

The measurement of the performance of the roles was to report the outcome of the majority of the last 5 confrontations for that role.

The Theory

The reason this strategy works well is that it contains a novel approach to the exploitation/exploration problem from reinforcement learning. The problem is that, when a situated agent is learning, there is a possibility for a feedback loop which can reinforce certain low utility conclusions through action.

The traditional approach to the problem is to have a learning agent split its time between exploring (trying alternative options) and exploiting (only following what the model says to do). In our case the approach we have taken is to treat the whole team as an agent and split the exploitation and exploration play of the team in response to changes in the performance of particular roles.

This adaptive multi-agent treatment is made simple by the use of REA-LCC and shows how this trade-off consideration can be thought of in a situated multi-agent sense.