Northeastern University – Silicon Valley

CS 5150 Game Artificial Intelligence

Homework #6

Sushruth Bhandary

Note : I worked on this assignment as a pair along with Michelle Lee and hence both our submissions shall have the same codes for some questions

1. Markov State Machines: Basic Implementation

The code for this question can be found in the file 'hw6_q1.py'. It is a basic implementation of a 2 state FSM. Press 's' on the keyboard when the program is running to start or stop the rat sound and that will cause the transition from one state to another. To run the code, use the command :

> python3 hw6_q1.py

2. Fuzzy logic and fuzzy sets

I used the skfuzzy library in python 2.7 for this question. I used Jupyter Notebook for this question since skfuzzy uses a non-GUI backend for matplotlib for visualization and hence I couldn't get the visualizations working without using Jupiter. The code can be found in the file 'hw6_q2.ipynb'. I implemented a fuzzy rule set to calculate the winning probability of a character given the character's health and the number of enemies. The code includes two test examples.

3. Goal oriented behavior (GOAP)

The code for this question can be found in the directory GOAP in the file GOAP.py

The GOAP algorithm works by reviewing all possible actions that are provided to it. For each action it looks at whether the prerequisites for it are satisfied, then adds it to the plan if it is cheaper than the existing plan. It hands the list back to itself to update recursively for the entire action list. In main, I declared two ultimate goals that the player can achieve. I declared several classes that represent the actions that could be taken. Each of these actions have a prerequisite that must be satisfied before it can be executed, an effect if it is executed, and a cost associated with the entire action. The goal, list of states, and possible actions are then passed off to the GOAP algorithm which returns the sequence of actions to be performed.

The first goal is to create a tomato soup and has a straight forward path that only has one sequence available: get the tomato, chop the tomato, make the soup. The second goal is to make onion soup but there are options to get to the same state where the player has the chopped onion.

The algorithm iterates through the onion soup tree and determines the shortest path which is: look in the pantry for the chopped onion and make the soup

4. Rule-based systems - RETE Algorithm

RETE Algorithm is a Rule Based System which is used for matching rules against a database. The Algorithm uses a special data structure called the Rate, which is basically a directed acyclic graph. The RETE algorithm represents the patterns in all the rules using this data structure. There are three important types of nodes :

- Pattern nodes
  These nodes represent the individual patterns. Rules in a Rule based system are made up of multiple smaller rules or facts. These facts or sub-rules are called patterns and can be repeated in multiple rules in the system. We identify these patterns and represent them as pattern nodes. Pattern nodes should not be repeated
- Join Nodes
  Join Nodes represent an operation between two patterns. For example, Logical operations like AND and OR between two facts can be depicted using a Join Node
- Rule Nodes
  Rule Nodes are the functions that need to be fired if a rule is satisfied

Once the Rete is Generated using the nodes mentioned above, we then feed the database to this algorithm. The pattern nodes try to find a match in the database.They find all the facts that match and pass them down to the join nodes. The nodes also pass down variable bindings. Variable bindings help us to keep track of the context as we go down the Rete. For example if we have multiple objects of the enemy class and one of those objects satisfy the pattern node, the algorithm will take the information of which objects satisfied that pattern and pass it down the Rete. Using these variable bindings from the pattern nodes, the join nodes take a decision and then pass down it's own set of variable binding down the Rete. This happens till we reach the bottom of the Rete and the function is fired.

5. Blackboard Architectures

The code for this question can be found in the file 'bb_architecture.py'. I implemented a simple game which includes a non playable character, three enemy characters(Red, Blue, Green circles) and an ammo stockpile(Black circle) where the character can refill the ammo. The bottom right corner of the screen is the safe zone. The character has a certain ammo to begin with. The enemies have a health count. One ammo reduces the enemy health by 1. So the character can kill an enemy only if it has at least as much ammo as the enemy character's health.

The code uses a blackboard architecture to make decisions in runtime. The Balckboard class has 2 important variables called 'mode' and 'executing'. Mode shows the action that the character is performing. The modes can be 'attack', 'restock', 'flee' or 'idle'. The 'executing' variable is used to manage access to the blackboard class. The function strategyExpert is used to decide at runtime the next move for the character.

The character first checks if it can defeat any of the enemies and if it can, it attacks the nearest enemy that can be defeated. If none of the enemies can be defeated, it tries to find an ammo stock pile to restock ammo and then checks if it can now defeat any of the enemies. If there are still no enemies that can be defeated, it'll flee to the safe zone. The health, character mode and ammo values are displayed on the game screen.

To run the program, use the instruction :

> python3 bb_architecture.py