Smart Applications 2020/2021

Software Architecture report of the Group AI 7

Antonio Di Mauro, Fabio Murgese, Gabriele Pisciotta

1. Introduction

Objective of the course is to produce an artifact that puts together most of our knowledge in order to create an intelligent product from scratch. The *Among AIs* idea is something that allows us to build in a creative way a game where, by design, it needs multiple and different intelligent strategies to be implemented and combined together for an entertaining yet intelligent experience.

In the following sections it will be explained how it works and which further improvements could be made to make this product a better one.

2. Diagrams

The software architecture of the components of the AI 7 agent is described in detail using the UML Component and Class Diagrams.

The main structure of our agent is made by a generic user, implemented by the class **PlayerInterface**, that represents a generic user of the game with the basic functionalities. For our purposes we extend the interface with the class **AI_Agent** that represents the AI Player.

The AI agent is composed of a set of pluggable components that allow our agent to understand the map and act appropriately in different ways, as you can see from the components diagram below.

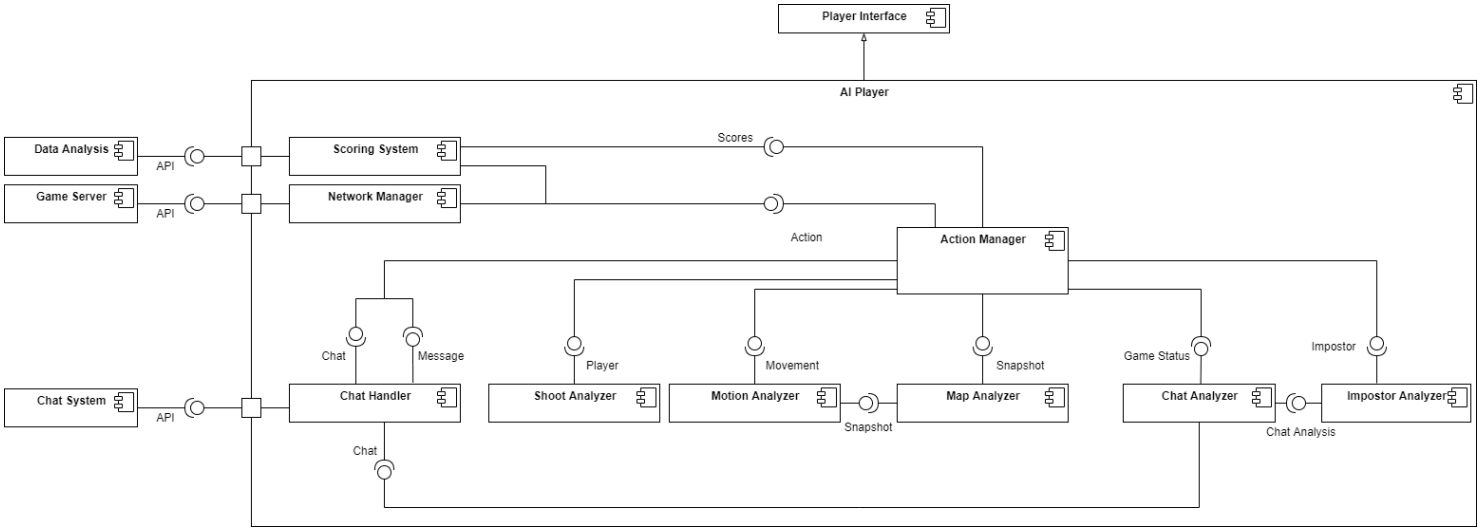
The core component of the AI agent is the Action Manager, implemented by the abstract class **ActionManagerInterface**. The role of the action manager is to retrieve information about the state from the others components and act with a specific strategy. The class ActionManagerInterface is extended by two classes:

- **SimpleActionManager**, used for the *proto* release
- **FuzzyActionManager**, used for the *alpha* and *beta* releases

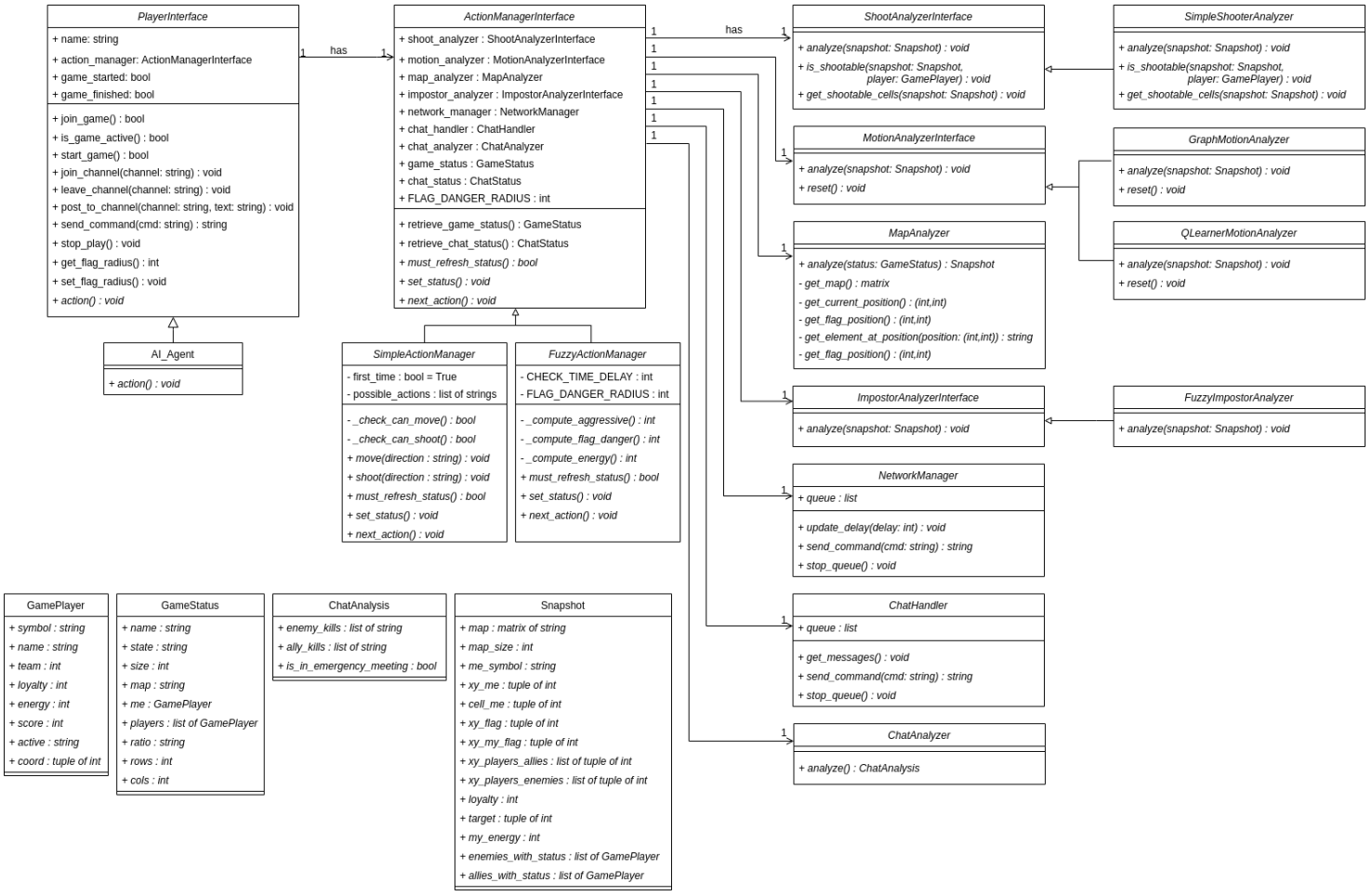
The different types of Action Manager could use the components between these:

- Shoot Analyzer, implemented by the abstract class **ShootAnalyzerInterface** and is able to understand who to shoot.
- Motion Analyzer, implemented by the abstract class **MotionAnalyzerInterface** and is able to understand where to go.
- Map Analyzer, implemented by the class **MapAnalyzer** and is able to manipulate in a useful way the map from the server.
- Impostor Analyzer, implemented by the abstract class **ImpostorAnalyzerInterface** is able to judge and accuse other players.
- Network Manager, implemented by the class **NetworkManager** is responsible for the communications between client and the game server.
- Chat Handler, implemented by the class **ChatHandler** is responsible for the communication between the client and the chat server.
- Chat Analyzer, implemented by the class **ChatAnalyzer** is able to analyze the chat.

2.1. Component diagram



2.2. Class Diagram



3. Theory of operations

We have designed a modular infrastructure that is composed of different components each one responsible for a different aspect of the agent and of the game. We decided to do this according to the software engineering principle of “separation of concerns”. Each component is described by an interface, that you can see in the class diagram. Then each interface can be implemented in multiple ways, letting us easily change the algorithms' nature behind the solutions. Each component calls the others exploiting the public methods available, so the internal aspects are completely hidden for it.

We decided to experiment with many concepts related to Graph Analysis, Fuzzy Logic and Reinforcement Learning, because we thought in the brainstorming phase that these techniques could be adapted to our aim. Our wish, for the final product, has been to keep a high explainability of the internal behavior of the agent. We discarded in a first step the usage of Machine Learning techniques in order to keep a sort of determinism in the behavior of our agent, and also because there was a lack of exploitable information (before the public availability of the logs).

From an algorithmic perspective, we noticed that our *Reinforcement-Learning* trained agents were much slower in finding the right path to the opposite flag, also displaying quite a dumb behavior in the firsts steps of the game. Also, once a game is finished and another one is started, the learning process of our agents needs to restart too. This translates into an impractical solution to this kind of problem, so we finished preferring almost always the graph approach, exploiting graph algorithms to compute the path to be followed by our agent plus some kind of reasoning made by fuzzy logic for other purposes (i.e. shooting other players).

We also discarded the option of the Cellular Automata because we were satisfied with the results obtained with the movement thanks to the graph analysis, trying to focus our efforts on fuzzy reasoning to give more intelligent explainable behaviour to our agents; more randomness, in that stage, would lead us to not understand what was happening.

Everything is fully customizable and we changed the “best solution” step by step according to the results obtained in friendly matches and tournaments. This let us continue to expand the behavior of the agent by simply plugging in different implementations of the same components or by adding other components and creating a way of communication between the existing ones.

4. Technical choices

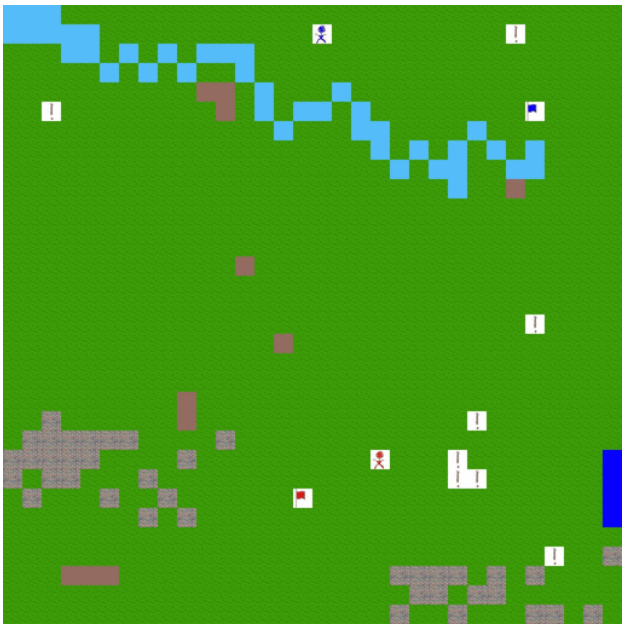
As the implementation language we decided to use Python to code our AI agent during a first brainstorm phase, thinking straightforward to the several useful libraries that are available and could be useful to our needs:

- ***socket*** to connect our client to the Game Server and make possible the communication between the latter and our agents
- ***NetworkX*** to manage graphs and compute the best path between our agent (source) and the opposite flag (target) exploiting many different algorithms
- ***scikit-fuzzy*** to implement an intelligent “action manager” that exploits fuzzy logic to reason about the best action among a set of possible outcomes

We defined a class *entity.py* where we declared all the objects useful for our different components, like *GamePlayer* containing all the information about the players in game, *GameStatus* that contains all the information about the current game, *GameChat* a combination of the messages written in the chat and the *GameStatus*, *ChatAnalysis* that concerns statistics about allies and enemies kills and also for emergency meetings; finally *Snapshot* in which we extract all the information from the *GameStatus,* manipulating this one to have more useful contents.

4.1. Visualization

In the first *Proto* phase of the implementation we used the *pygame library* to create a full environment for the visualization of the matches, linked to our components. At the beginning this visual interface gave us a great speed up for the development, letting us see our agents moving in the map and so visualizing faster problems and solutions.



We created our personal GUI for AmongAIs while waiting for the Web App and Desktop Client to be released; then we removed this component from our software because of the better developed solutions provided by their specific teams.

By the way we noted that the rendering impacted on the speed of our agents, slowing them down because of the computations needed for the visualization and the animation of graphical details present in the map.

5. Algorithmic and implementation details

5.1. Baselines components

Before involving different kind of techniques to handle the movement of the agent and its action manager, we created two baseline classes, defined as follows:

5.1.1. Simple Action Manager

This class, each time that a status is retrieved (being it a tuple of MAP and STATUS results), it first checks if in this position can shoot. This is done by calling the SimpleShootAnalyzer that checks the shootable cells and then returns the first direction that can be shooted. Then it computes the next move by involving a MotionAnalyzer component. All the possible actions are stored in a list, and the action that will be performed is chosen randomly. This component, in the end, is a really dumb one that acts as a baseline to be improved with smarter action managers.

5.1.2. Simple Shoot Analyzer

This is responsible for the analysis of the possible shots. First of all, starting from the actual cell in which the agent is, it first checks if this kind of terrain is possible to shoot. If possible, it gets the shootable cells. A shootable cell is a one that is between the agent and the end of the map, or an obstacle that blocks the bullet, or a team-mate that we don't want to shoot. We match in the end the shootable cells with the cells on which the enemies are. This component is really simple and can be improved, in the future, by considering a range of predicted next moves of the agents, in order to anticipate their moves and shoot them.

5.2. Network Manager

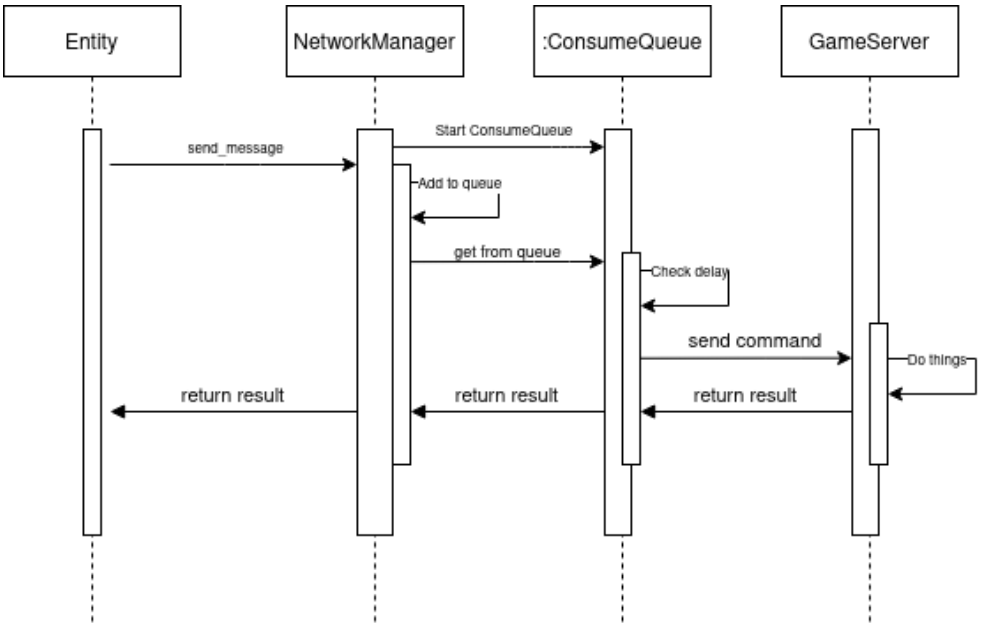
The rules of the game expect for the commands to be sent each a variable amount of time, such as 0.5s or 0.3s. This is perfectly tunable from our system, in fact before the joining of the agents the commands are sent each 0.5s and after it becomes 0.3s. If the match has the proper flag that signals the "training" kind, it goes down to 0.05s.

In order to manage everything automatically we created a *NetworkManager* component that is responsible for the communication between any entity and the *GameServer*.

It exposes the send_command() method, and we exploited the "producer-consumer" parallelism pattern, so we have a queue in the *NetworkManager*, and when the

NetworkManager object is created it starts a new process that consumes the queue sending the commands each proper delay, or sending the NOP command if there's anything to send after some time, to avoid to be kicked out from the match.

When any entity sends a message, it's created an object for that request containing a specific ID, it's added to the queue and then we wait for an answer. If the time of the requests exceeds the awaitable, the request is resent, and after a max amount of resent, is in the end dropped, raising a TimeOut exception.



In this sequence diagram it's possible to see all the phases. The NetworkManager is a "singleton" for our agent, so we would expect that in the future it could accept multiple requests at the same time, processing them of course sequentially. The "producer-consumer" pattern allows us to accomplish our aim.

5.3. Chat Handler

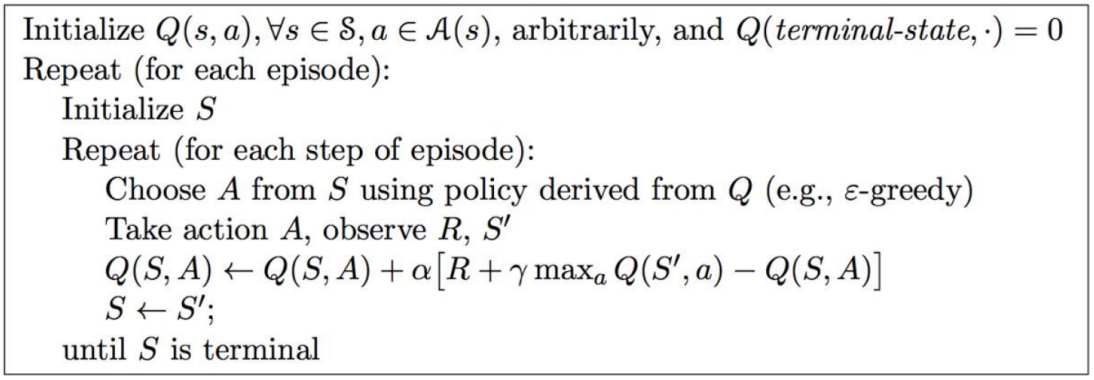
The *ChatHander* component allows us to manage the interaction with the Chat Server. Differently from the *NetworkManager*, this component can interact with the *ChatServer* completely asynchronously, with no response expected from the latter. Likewise the already seen *NetworkManager*, the *ChatHandler* exploits a queue to send messages to the *ChatServer*, this time without waiting for a response from it but, instead, listening for all the messages coming from the *ChatServer* thanks to the *read_chat()* method.

5.4. QLearnerMotionAnalyzer

The QLearnerMotionAnalyzer component is responsible for finding the best path maximizing the total discounted reward.

It uses the Q-Learning, an Off-policy TD learning algorithm. This algorithm uses a Q-function, that returns the next action to perform with the maximum Q-value, given a particular state. The algorithm converges to the optimal policy during iteratively update of the Q-values using the Bellman equation.

In the picture below is shown the algorithm.



Q-Learning Algorithm for Off-policy Control

The reward values for the objects in the map are the following:

- if the cell is the **target** the reward is **+5**
- if the cell is the **bonus** the reward is **-1**
- **otherwise** the reward is **-5**

The parameters chosen for the algorithm are the following:

- alpha: **0.99**
- number of episodes: **1000**
- epsilon: **0.35**
- epsilon decay: **0.00001**
- gamma: **0.99**
- maximum number of iterations: **1000**

The *alpha* parameter is a sort of learning rate used to look ahead in time, as a sort of window on the future, to understand what to do. The *epsilon* parameter is used in the greedy-function to set a probability to accomplish exploration and exploitation, if a random probability is less than *epsilon* performs exploration with a random action, otherwise performs exploitation giving the next action with the maximum Q-value.

5.5. GraphMotionAnalyzer

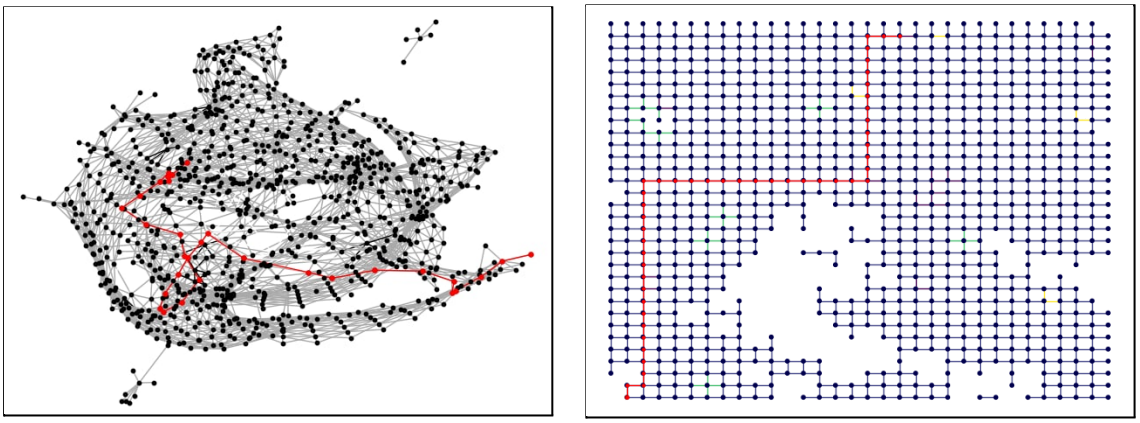
The GraphMotionAnalyzer component is responsible for the motion of the agent exploiting graph algorithms.

In order to manage graphs, we used the *NetworkX* library. We extracted a graph from the map, considering each cell as a node, linked to its neighbors.

Each edge has been weighted considering the following rules:

- 0, if the target is the bonus or the enemy flag
- 0.2, if the target is grass or river
- 1, if the target is the trap

If the target is a barrier, we'll see if it's possible to move that, and if possible, its weight has been 0.7 until Beta release, in order to discourage this kind of action, if it's possible to go somewhere else; but, in the end, we chose to set it at 0.1 in order to exploit this barriers as shield to be carried throughout the map as a defense method.
If a cell is a wall or an ocean, it won't be walkable and so won't be inserted into the graph.



Here we can see an example of the graph generated from the map in two different views, one that involves embedding of the nodes into a different space and one with a grid layout, similar to the real map. The red line represents the path that the agent is going to traverse, from its original position to the flag. It's also possible to see edges having different colors, such as green and yellow: these represent edges that go to bonuses and to traps.

The optimal path will be computed with the help of A* algorithm, considering the weights described before. Once that it's computed, the optimal path is stored and retrieved iteratively from the agent, and will be recomputed if its target cell is changed according to its strategy (e.g.: attack the enemy flag or defend its flag).

We considered the Hamming distance as heuristic for the A*, concluding that it wasn't really useful as expected. We in the end left A* without heuristics, being properly a Greedy Best-First search algorithm. If no path exists from the actual cell to the target cell, the agent will simply move to a random walkable cell, in order to change it's possibility to encounter an enemy agent to shoot even if it can't reach the flag.

5.6. Chat Analyzer

Thanks to the Chat Analyzer we perform an analysis of the messages written in the chat to recollect useful information about kills and emergency meetings. In particular, we count the number of killed allies and enemies for each player, and if there are emergency meetings already called and ongoing. These data will then be useful to the fuzzy impostor analyzer for trying to find who is the impostor “Among AIs”.

5.7. Fuzzy Action Manager

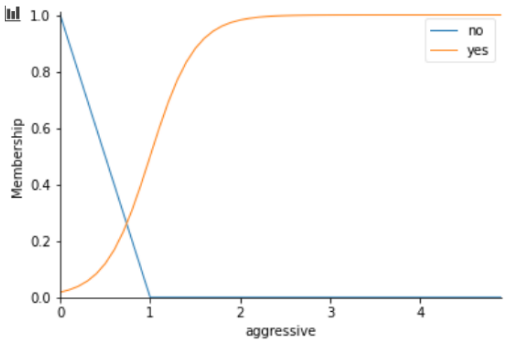
This component exploits fuzzy logic to determine the probability of the next action to be performed (using python library *scikit-fuzzy*). This component basically query each other one in order to obtain all the information it needs (e.g.: start the analysis of the map, the analysis of the chat, the possibility to shoot, ...).

Here we involve 3 main concepts to drive our strategy:

- ***aggressivity****:* computed considering the amount of shootable enemies in a radius (that can be the simple directions shootable from the actual cell or a more large one considering the min number of cell needed to shoot an enemy)
- ***danger to our flag***: computed considering the amount of enemy that are surrounding our flag, according to a specific radius that may be tunable in the configuration and that changes according to the size of the map. This can be used in order to choose if it’s better to go back to defend our flag or to keep going to attack the enemies’ one.
- ***energy***: each agent has an energy that varies from 0 to 256, and it loses energy by shooting and can gain it by taking bonuses. If the energy goes to 0, an agent won’t be able to shoot, so it’s better to consider this value in order to choose to be more a “stealth” agent or a “killer” agent.

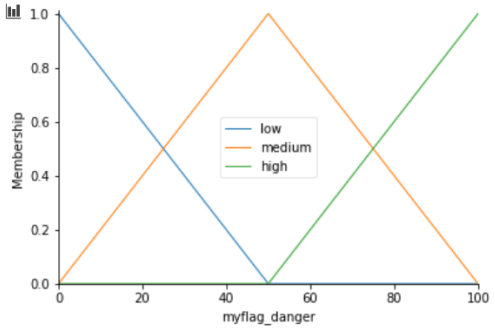
The membership function of the antecedents of our rules set are the following:

1. **aggressive** membership function accepts as input the number of enemies killable (up to 4) in the cardinal points: N, S, E and W

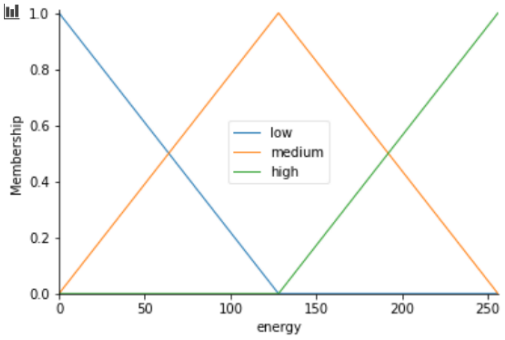


2. **myflag_danger** membership function accepts as input the percentage of enemies in a specific radius around the team’s flag, if the agent is an impostor

and is closer to the enemy flag instead the team flag, the percentage of danger is 0

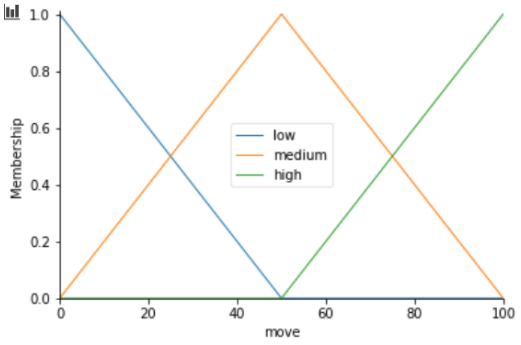


3. **energy** membership function accepts as input the energy of the player (maximum 256)

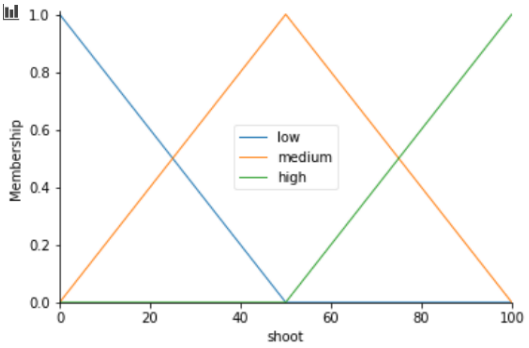


The membership function of the consequences of our rules set are the following:

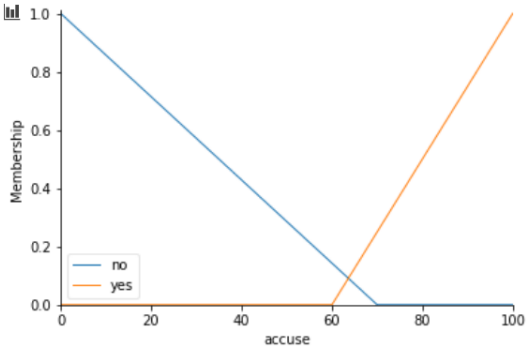
1. **move** membership function return the percentage of moving



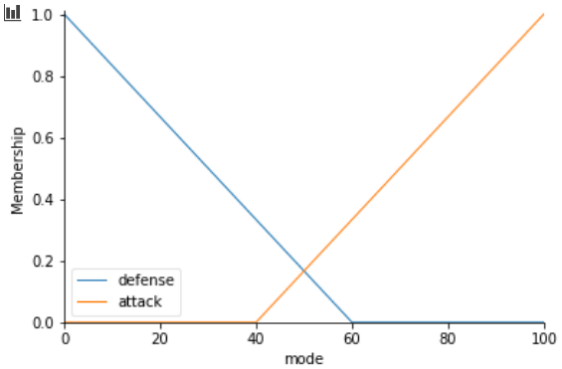
2. **shoot** membership function return the percentage of shooting



3. **accuse** membership function return the percentage of accusing



4. **mode** membership function returns the percentage of the agent modality, if membership value is greater or equal to 0.5 the agent is setted to **attack** mode, otherwise in **defense** mode



A rule in the rules set is composed by antecedents and consequents, described above. The rules are the following:

1. **IF** aggressive[‘yes’] **OR** myflag_danger[‘low’] **OR** energy['low'] **THEN** move['high']
2. **IF** aggressive['yes'] **OR** myflag_danger['medium'] **OR** energy['medium'] **THEN** move['medium']
3. **IF** aggressive['no'] **OR** myflag_danger['high'] **OR** energy['high'] **THEN** move['low']
4. **IF** aggressive['yes'] **AND** energy['high'] **THEN** shoot['high']
5. **IF** aggressive['yes'] **AND** energy['medium'] **THEN** shoot['high']
6. **IF** aggressive['yes'] **AND** energy['low'] **THEN** shoot['low']
7. **IF** aggressive['no'] **OR** myflag_danger['high'] **OR** energy['low'] **THEN** shoot['low']
8. **IF** aggressive['no'] **AND** myflag_danger['low'] **AND** energy['low'] **THEN** accuse['yes']
9. **IF** aggressive['no'] **OR** myflag_danger['low'] **OR** energy['low'] **THEN** accuse['no']
10. **IF** aggressive['yes'] **OR** myflag_danger['medium'] **OR** energy['low'] **THEN** accuse['no']
11. **IF** aggressive['yes'] **OR** myflag_danger['high'] **OR** energy['low'] **THEN** accuse['no']
12. **IF** aggressive['yes'] **OR** myflag_danger['low'] **THEN** mode['attack']

13. **IF** aggressive['no'] **AND (**myflag_danger['medium'] **OR** myflag_danger['high']**)** **THEN** mode['defense']

After that the Fuzzy Action Manager found the next action to do, it automatically performs it by calling the proper method responsible for that.

The agent will refresh the STATUS needed for the Action Manager, by cross-check information coming from different parts in order to save some useful milliseconds: the map will be taken only one time with the LOOK command and then it'll be reconstructed by moving the agents according to their coordinates taken from the STATUS.
The STATUS will be refreshed according to the following rules:

- it can be forced by some routines that need a fresh STATUS
- it's expired (e.g.: it's older than 3s)

As **future work** we plan to exploit intelligent metric in order to select the best STATUS expiring time, considering for example the minimum distance from an enemy in terms of difference of the X or Y axis individually if we/it are/is shootable. This could let us save more seconds, if we are sure that can go without check for the possibility to shoot or escape. This could also be mixed with machine learning techniques to predict the possible moves of the enemies.

5.7.1 Playing the Turing Game

Our first approach for solving the Turing Game has been to use a trivial random choice where we gave the 80% of the probability of being an AI while the remaining of being a human, taking into account the fact that the agents were always the majority in tournaments. For the *Beta* release, we exploited the report of the Data Science team to observe an interesting thing: **Humans are much slower than AIs**.
In the DataScience report the values are 27.716 for the AIs and 13.756 for the humans.
Even if it's not clear what measurement unit is, we clearly saw a ratio of 2:1, so we decided to involve this feature in order to discriminate and judge.

In order to evaluate the speed of a player, we consider the difference between each single STATUS update that we do. Each update can be done in different time intervals, so we first evaluate the maximum number of moves that could have been done in that period. For example, we get the STATUS at time 0s and then we get the status at time 1.3s. The possible moves according to the specified delay must vary, for the sake of the explainability the delay is 0.3s. The maximum possible moves will be actually 1.3/0.3 = ~ 4.

We then evaluate the distance of each agent, from the previous position, according to a simple hamming (cityblock) distance. In the end we save its *instantaneous speed.* This computation is done each time we refresh the STATUS.
When we perform the judgement, we take the mean speed of each agent by computing the mean values of each instantaneous speed saved. Considering the fact that humans are slower than AIs, we take all the means computed and we extract the **first quartile** (25-percentile). This value is our threshold: if the mean speed of a player is below that value, for us is a Human, otherwise it's AI.

According to the tests performed in various 3 H vs 3 AI matches (battle of species), we found a **precision of up to 100%**. When we perform unbalanced matches, e.g. 3Hvs12AI or 1Hvs6AI we can observe a slight downgrade, having few agents that are actually classified as AI. As we can see, this method, in an unbalanced set, can be fragile due to the fact that being a statistical method it's influenced by the amount of players: for example if an agent is actually slow because it's performing a JUDGE or it's shooting, and it could be classified as human. This can be mitigated by performing a judgement after some time from the starting of the match. Another flaw could be that if an agent goes forward and then backward, it won't be recognized by the simple hamming distance. This is intrinsic in the kind of distances considered to evaluate the speed, but can be mitigated by doing more frequent refresh of the STATUS or performing the JUDGE lately.

As **future works**, we could also consider the *shooting precision* as further feature: a combination of it with the speed of the agent could push up the precision and mitigate the fragilities described before.

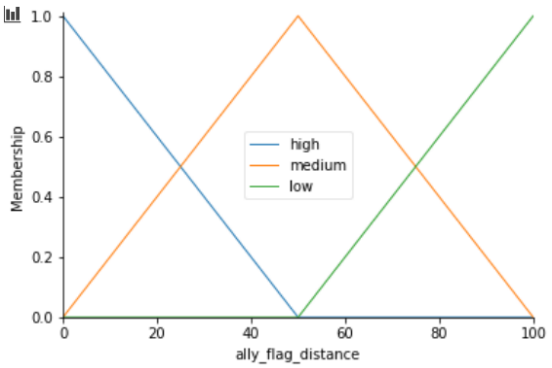
5.8. Fuzzy Impostor Analyzer

This component exploits fuzzy logic to determine the probability of a player to be an impostor (using python library *scikit-fuzzy*).
To judge who's the impostor we consider three concepts:

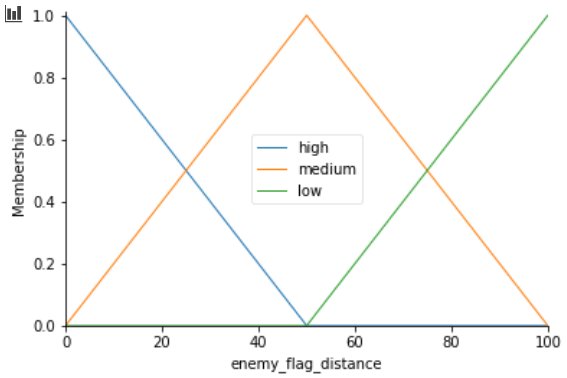
- **distance from its flag**: if an agent tends to don't attack and remains in its assigned zone, maybe it's not interested in playing the game properly. (note: it could be actually having a defensive strategy, so we need to combine this concept with other features)
- **distance from enemy flag**: idem but inverted
- **number of allies killed**: this is an interesting signal to consider. We take this value exploiting the chat analyzer. If a player tends to kill its team-mate it could be an impostor, trying to sabotage its team.
- **number of enemies killed**: it's allowed to kill team-mate erroneously, but it's auspicable to kill also the enemies. If a player doesn't kill enemies but keeps killing allies, this it's a signal that must be taken into account.

Mixing all these concepts lets us have a strong technique to evaluate each player according to some rules.
The membership function of the antecedents of our rules set are the following:

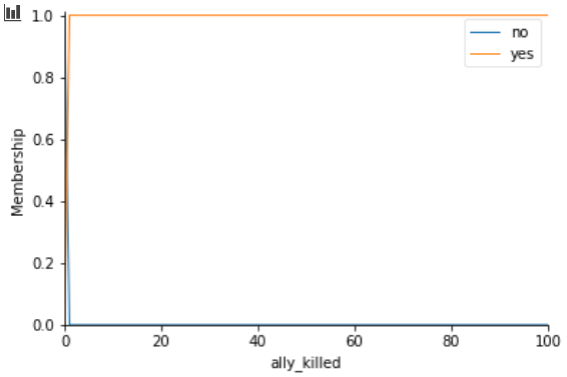
1. **ally_flag_distance** membership function accepts as input the percentage of how much is distant from the ally flag with respect to the enemy flag



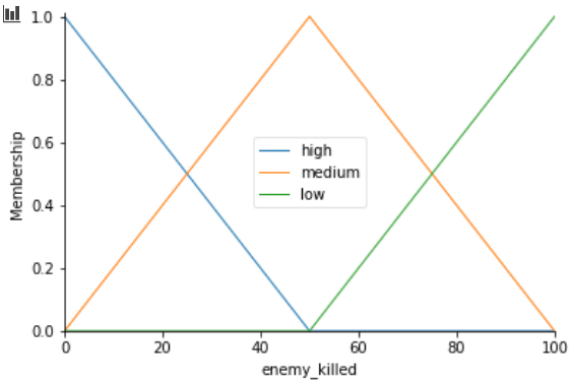
2. **enemy_flag_distance** membership function accepts as input the percentage of how much is distant from the enemy flag with respect to the ally flag



3. **ally_killed** membership function accepts as input the percentage of allies killed and returns an high value of membership if the ally killed are at least 1



4. **enemy_killed** membership function accepts as input the percentage of enemies killed



A rule in the rules set is composed by antecedents and consequents, described above. The rules are the following:

1. **IF** ally_killed['yes'] **THEN** impostor['high']
2. **IF** ally_flag_distance['high'] **OR** enemy_flag_distance['low'] **OR** enemy_killed['low'] **THEN** impostor['high']
3. **IF** ally_flag_distance['medium'] **OR** enemy_flag_distance['medium'] **OR** ally_killed['no'] **OR** enemy_killed['medium'] **THEN** impostor['medium']
4. **IF** ally_flag_distance['low'] **OR** enemy_flag_distance['high'] **OR** ally_killed['no'] **OR** enemy_killed['high'] **THEN** impostor['low']

6. User Manual

6.1. Configurations

Our system is completely customizable and adaptable to further changes with no hard-coded constant in the code. Everything, from delay values between commands to the agents that will be joined in the match, is in an external configuration script: *config.py*.

6.1.1 Network Manager and basic game configurations

```
SERVER_HOST = "margot.di.unipi.it"
SERVER_PORT = 8421
SERVER_TIMEOUT = 100
SERVER_DELAY_SEC = 0.50
TRAINING = False
if TRAINING:
   SERVER_DELAY_SEC_AFTER_JOIN = 0.05
else:
   SERVER_DELAY_SEC_AFTER_JOIN = 0.30
SERVER_NOP_SEC = 20
CMD_REQUEST_TIMEOUT_SEC = 60
MAX_RESEND = 3
SERVER_GAME_NAME = "game_match"
SINGLE_AGENT_MODE = False
CONNECTION_CLOSED = "Connection closed by remote host"
SHOOT_COOLDOWN_SEC = 5
FLAG_COOLDOWN_SEC = 30
```

These are the standard values, according to the game's protocol.
In order to change the match name, you can change the SERVER_GAME_NAME value. When using the interfaces built to test the game, in order to speed-up the training, you can set the TRAINING variable to True in order to automatically create matches with the -T flag. The variable CMD_REQUEST_TIMEOUT_SEC is the one used to wait a response from the server after the sending of a command, and the MAX_RESEND is to set up the max amount of resending. SINGLE_AGENT_MODE is used when you want to populate a match with all the agents needed to start it, and in the end manage only one of them, for test purposes.
The CONNECTION_CLOSED variable is in the end used to store the constant string returned from the server in case of closed connection. The last two are the cooldown present in the game, one for the “catch the flag” and one for the “hunting”.

6.1.2 Chat Handler

```
CHAT_SERVER_PORT = 8422
CHAT_SERVER_NOP_SEC = 250
GLOBAL_CH = "#GLOBAL"
CHAT_CH = "#CHAT"
LEAGUE_CH = "#LEAGUE"
LOGS_CH = "#LOGS"
DATA_CH = "#DATA"
STREAM_CH = "#STREAM"
NOP_CH = "#NOP"
```

These are the values needed by the Chat Handler, according to the chat’s protocol.

6.1.3 Chat Analyzer

```
HIT = "hit"
START_EMERGENCY_MEETING = 'EMERGENCY MEETING! Called by'
END_EMERGENCY_MEETING = 'EMERGENCY MEETING ended'
ACCUSES = "accuses"
```

These are the values needed by the Chat Analyzer, they’re basically strings that can be present in the chat.

6.1.4 Agents configuration

```
DEFAULT_AGENT = "graph" # or rl
USER_INFO = "AI_7"
AGENTS = [
    ["Agent_1", "fuzzy"],
    ["Agent_2", "simple"],
    ["Agent_3", "fuzzy"],
```

```
]
SINGLE_AGENT = ['Agent_1', 'fuzzy']
```

These values are critical for our infrastructure. The DEFAULT_AGENT can be “graph” if you want to use the GraphMotionAnalyzer or “rl” if you want to use a Reinforcement Learning based approach to move the agents. The USER_INFO is the one needed for joining a game, according to the game's protocol. AGENTS is then a list of all the possible agents that you want to let join the match. Each agent is a list composed of its name and a string that can be “fuzzy” if you want to use the FuzzyActionManager or “simple” if you want to use the SimpleActionManager.

6.1.5 Flag danger radius

```
SMALL_FLAG_RADIUS_Q = 5
SMALL_FLAG_RADIUS_W = 10
MEDIUM_FLAG_RADIUS_Q = 15
MEDIUM_FLAG_RADIUS_W = 20
BIG_FLAG_RADIUS_Q = 25
BIG_FLAG_RADIUS_W = 30
```

These are used in the FuzzyActionManager to compute the rate of danger for our flag, in order to choose the best strategy (attack or defense). According to all the possible kinds of maps, we propose different radius surrounding the flag, these are tunable.

6.2. Join with a single agent

If you want to join the game with a single agent, you can simply configure the SINGLE_AGENT variable in config.py, specifying your name and the kind of ActionManager that you want to use (e.g.: “fuzzy”) and then run the script *join_single_agent.py.* It will take care of the life cycle of the agent, but you’ll need to create the match manually.

6.3. Join with a multiple agent

We created a script that makes it possible to join a pre-created game with a variable number of agents (*k*) simply by setting the parameter of the game name in the *config.py* file and the names of the various agents to join the game.
If you want to join the game with multiple agents, this becomes critical because each agent must be independent and they must act in parallel.

Considering the problem of the *GIL* (Global Interpreter Lock) of Python, we decided to run each agent in its own process detached from the main script used to run the game: *join_game_with_k_agents.py*.

We achieved real **parallelism** by exploiting the *multiprocessing* library, avoiding the multithreading problem, and having standalone agents. Each agent is created starting from the data included in the variable AGENTS. Take into account that the number of agents to join have to be tuned according to the CPU you're using.
We tested that everything runs correctly on Ubuntu 20.04.1 LTS with a Intel® Core™ i7-6700HQ CPU @ 2.60GHz × 8, having 8 cores: we successfully joined up to 12 agents. By the way we can observe a slight downgrade of the performances of the entire system that also may influence the agents (e.g.: from 0.35s per action to 0.45s) when having more than 9 multiple agents at the same time.

6.4. Automatic tournament subscription

Another critical thing that we need to handle in this game is the tournament subscription. We created a script that automatically gets all the available tournaments, letting you decide in the end which one to join. We involved the use of the *requests* library in order to perform queries to the API exposed by the League Manager team.

```
Available tournament:
    Game type: None
    ID: LM-test3
    Matches start at: 2021-12-16 15:16:00.346000
    Subscriptions start at: 2020-12-16 15:16:00.346000

Available tournament:
    Game type: championship
    ID: LM-prova-7
    Matches start at: 2021-01-12 23:00:00
    Subscriptions start at: 2021-01-13 05:00:00.823000

Insert the ID of the tournament of which you want to subscribe: LM-prova-7
Check: True
Check: True
Check: True
Check: True
Check: True
Check: True
```

All the agents present in the variable AGENTS will be subscribed to it, and next you'll be able to join the game in the way exposed in the subsection 6.2 and 6.3.
After each subscription, we use the API to check the correctness of the subscription procedure, checking if the name of the agent is present in the list. In the image above you can see an example, where there are two available tournaments and after having typed the ID of the second, it subscribes 6 agents, returning a check of their subscription.

As **future work**, this could be exploited to keep a h24 running agent that automatically retrieves tournaments and scheduled matches, playing completely autonomously. This could be very interesting to experiment because in this way we could have many more played matches to be exploited for learning purposes or to assess the performances of all the agents during the time letting them fight continuously.

6.5. Automatic test (*test_game.py*)

To internally test we also developed a sequential script that acts similarly to *join_game_with_k_agents.py*, but being different in the fact that it also creates the game (which name must be specified in the configuration script) and after that lets the agents join. The main difference is that the agents here won't run in parallel.
This must be intended for internal testing only, when the speed of each agent is not crucial or when it's ok to let them act one after another.

7. Conclusions

The development of this project, starting from the brainstorming and finishing with the beta version, has been really challenging and at the same time really interesting. This lets us combine many concepts from many different scientific areas to build an autonomous agent that's able to play according to the rules we decided collectively during the Smart Application course.

The concept of "tournament" has been also interesting due the fact that we had to challenge other AI agents developed differently from ours: we analyzed each battle to find our weaknesses and iteratively enhance the whole infrastructure developed.

Involving humans in the loop of the creation of the AI let us discover many things that we couldn't learn by just playing against bots: we understood the kind of strategy that our agents pursued in different maps and situations. Thanks to their nature, being completely explainable, we were able to constantly analyze and understand what they were doing and why. We saw a slow, but constant, increase in our ability to score tournament after tournament. In the first ones we were mainly slow, but we enhanced our performances after that we developed all the other things, to first be sure of the correctness of the other things.
Measuring our agents, we are able to move at a maximum speed of 1 cell each 0.33s (of 0.06s if training instance), instead of 1 cell each 2s, that was the speed related to all the tournaments before the last. Doing friendly matches we showed a great improvement of our ability to catch the flag and also kill the enemies.

We also appreciated the analysis done by the DS team, that let us discover interesting patterns that differentiate the humans from the AIs, letting us include those to play better the Turing Game. Playing that, we can achieve a precision of up to 100% in balanced matches, but we also noted possible flaws and how to mitigate them. Everything related to the Turing Game can be read in its proper paragraph. We would have appreciated more if the report of the DS team was generated automatically after each tournament instead of waiting for the end of the whole development process: we think that with better cooperation with the teams the product could have been much better than how it is now.

We discovered that techniques different from the nowadays common Deep Neural Network can be successful in scenarios where there is no training data available or if you prefer to keep a high level of explainability. We intentionally decided to not involve machine learning, but we can consider that as a future work to experiment. We could use some machine learning techniques to predict the next position of the agents in order to reduce the amount of STATUS/LOOK commands sent to the server and so to speed up the process. We could also involve the computation of the minimum distance from enemy agents in terms of differences on the X or Y axis, in order to have a varying “refresh STATUS expiring time”, avoiding to send STATUS/LOOK command if there’s no possibility to encounter enemies in a specific range of time and then refreshing it more frequently if we’re near.
We could also use ML techniques in order to analyze the behavior of humans in the matches and replicate interesting strategies to fastly kill enemies. Anyway we consider the use of machine learning as complementary to the technique used right now, and not as a replacement.

We planned to involve swarm intelligence algorithms or genetic algorithms to tune the intervals related to the fuzzy rules but due to the few time remained we decided to invest the time in the enhancement of the already built algorithms, so we consider this as a future work that could be investigated and we think that it could give us a better improvement maximizing the ability of the agent.

We could also involve more sophisticated techniques coming from the Network Science in order to understand what are the more popular and congested areas of the map in order to avoid them (if the agent is stealth or not so aggressive) or to specifically go there with the aim of killing enemies (if the agent is particularly aggressive).

Another interesting thing that we plan to develop as future work is the automatic tournament system, that lets us keep the agent h24 awake listening to new tournaments starting. This could be really interesting because this makes the agent fully autonomous without involving humans, so letting us play every time it’s possible, generating many logs files that could be useful for the machine learning algorithms and also to assess the superiority of the agents by having continuous battles.

The Turing Game has been really challenging and we planned to include other interesting features that we discovered from the DS report, such as the shooting precision, in order to mitigate flaws we evidenced in our method and achieve a better accuracy.

As critical points and considerations about our agents, we noted that the optimization of the performances has been crucial. We needed to analyze and save each millisecond lost by cross-checking multiple data sources (e.g.: chat, LOOK, STATUS,

responses…), instead of trying to have the exact value. We preferred a sort of approximation instead of the precision, and this let us scale and save the time to invest into more reasoning and fast moves. We also invested some time to achieve a real parallelism in Python, avoiding the multithreading and preferring the multiprocessing. We found also useful the implementations of common parallelism design patterns, such as the “producer-consumer”.

Concluding, the resulting agent is really competitive considering the other developed, according to the latest friendly matches disputed after the second-last tournament.
We evidenced weaknesses and strengths of our AI, and also found interesting things that could be applied in the future.