# PRI - Musical Companion

## *Project*

### Introduction

This project is part of the PRI course at ENIB and it's developed with the guidance of M. Pierre De Loor. The main goal is to develop a real time Musical Companion so as to be able to play a melody that accompanies a human player. The way this works is by having a graph of musical notes in which we move from one note to the next, according to a probability value for each possible following note. In consequence, the melody is not always the same as it has a chance to choose a different option every time it arrives at the same node. The program also has to be able to be modified as it is played, in accordance to two different influences:

Inspired by ant colonies, we simulate "ants" that place pheromones on each edge they visit so as to increase the probability to traverse that same edge again. The second influence is the human player himself by placing new nodes and placing pheromones on the edges he traverses..

In the end, we should be left with a functional Musical Companion that responds to the way the music develops and the musician's choices of what to play next, by providing a melody that goes with the main melody.

### Ant Colony Optimization Algorithm

This algorithm was created by watching the way ants behave when looking for food and how they communicate to share the path they found. When an ant faces the choice of more than one way to reach its food, it is forced to decide and follow a certain path. By doing this, it leaves a certain amount of pheromones to indicate the next ant of the path taken by a previous one and try to help in the decision. When multiple ants have followed this path, we have an established probability for each different option and ants move in a certain way.

Why is this convenient for our specific problem? Well, because this gives us the possibility to modify all the graphs  at the same time as ants move and pheromones evapore at every beat of the music. This also allows us to have ants that leave pheromones in each node they visit but they don't play any notes, leaving this job to a particular ant whose only job is to follow existing pheromones and play the notes of the nodes it chooses.

# *User documentation*

## Dependencies

You will need:

- Python 3
- Python libraries:
    - mido
    - pyqt6
    - pydot
- Graphviz

## Configuring a session

The user can configure their session with a JSON file. It contains a json object with the following attributes:

- “bpm”: the tempo in beats per minute
- “decay_factor”: the evaporation rate of the pheromones
- “graphs”: a list of graph specifications
    - “path”: the path of the graph file
    - “nb_composers”: the number of composer ants to add
    - “user”: (Optional) a midi output object (see below)
    - “musicians” a list of midi output objects (see below)

Midi output object: An object with the following fields:

- “port”: midi port
- “channel”: midi channel

Example JSON session file:

**A GUI program “config_window.py” is provided to configure graphs without editing the json**

```
{
      "bpm": 80,
      "decay_factor": "0.1",
      "graphs": [
            {
                  "path": "demos/drums.json",
                  "musicians": [
                        {
                              "port": 0,
                              "channel": 9
                        }
                  ]
            },
            {
                  "path": "demos/hi_hats.json",
                  "nb_composers": 4,
                  "musicians": [
                        {
                              "port": 0,
                              "channel": 9
                        }
                  ]
            }
      ]
}
```

## Launching a session

To launch a session from your configuration file:

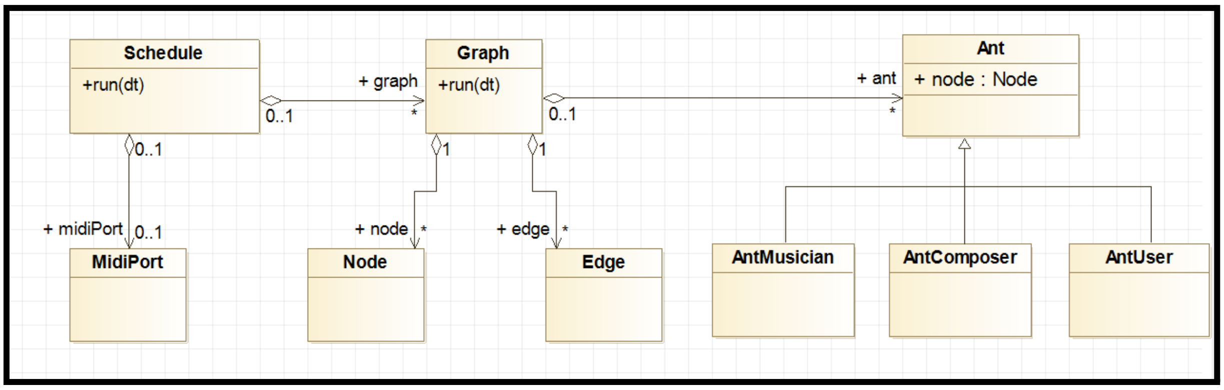
```
python Scheduler.py myConfigFile.json
```

To launch a graphical debugging session session from your configuration file:

```
python Gui.py myConfigFile.json
```

# *Technical documentation*

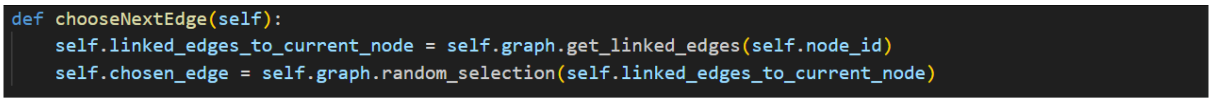
## Class Diagram



- The highest class in the hierarchy is the Scheduler. It's responsible for opening communication ports, handling beats per second and assigning the time and placement of each task. It runs each of the loaded graphs in order.
- Graph.py has all the functionalities regarding graph management. It loads and writes graphs in json files, creates a list of notes and builds the markovian chain with the edges between notes. As it is the one that has all the information of the graph, it's also in charge of applying pheromone evaporation and recalculating probabilities values according to placed pheromones.
- Ant.py condenses the tasks to be done by different kinds of ants in the code. Both Musician and Composer Ants use the same function to choose the next node to visit except that the musician implements midi functions to send the note to be played. Apart from that, the User Ant receives human inputs to be handled and written in the corresponding json file.
- Node and Edge are two classes created with the information from json files indicating the nature of a certain note and the edges linking it to different ones.

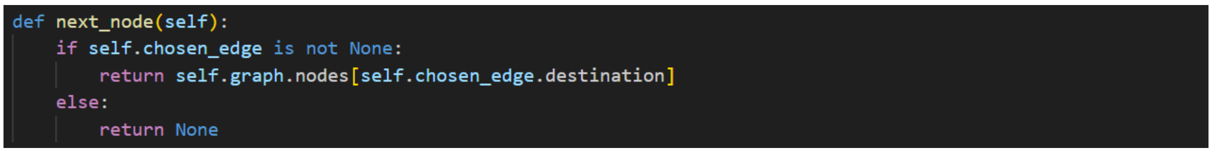
# *Functions and Working Process*

After creating a graph with the corresponding ants, Graph.py increments the loop_position, in other words, the beat in the pattern. Then, each ant, composer or musician, runs its logic to find the next node in their way. These are the main functions in the process:



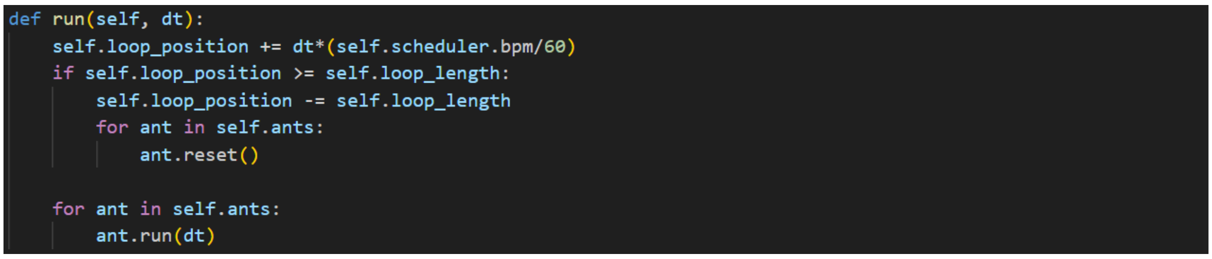
```python
def chooseNextEdge(self):
    self.linked_edges_to_current_node = self.graph.get_linked_edges(self.node_id)
    self.chosen_edge = self.graph.random_selection(self.linked_edges_to_current_node)
```

By using the attributes of the ant class, we can get information of the linked nodes to the current one, by passing the *self.node_id* as argument to the *get_linked_edges* function from graph. Then we use this list to make a random selection taking into account the probability of each edge, first with default values and later on with values derived from the amount of pheromones in each edge between nodes. We access the resulting node in the following function:



```python
def next_node(self):
    if self.chosen_edge is not None:
        return self.graph.nodes[self.chosen_edge.destination]
    else:
        return None
```

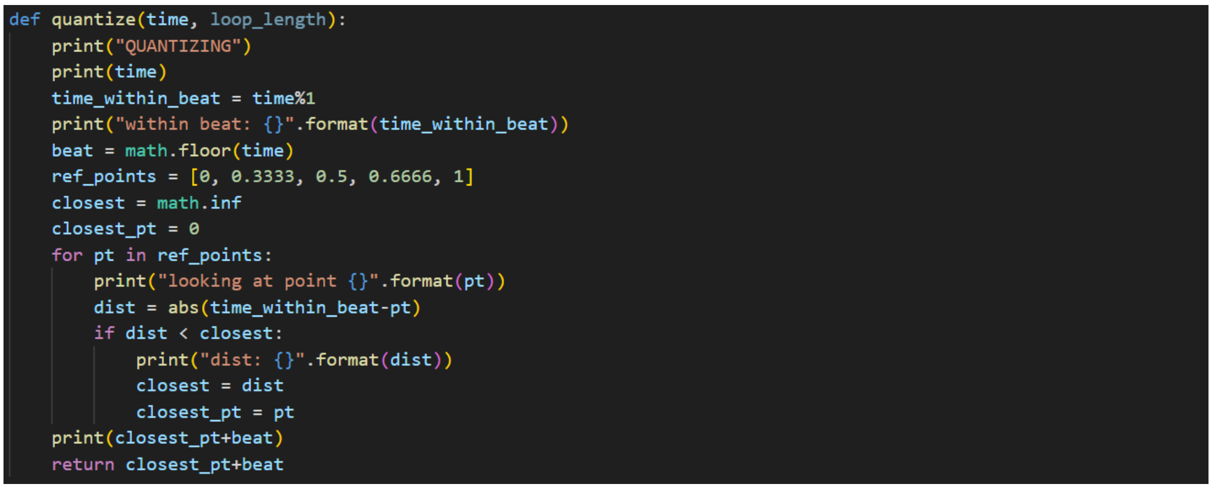
This way, the graph will be telling the ants to move from one node to the next until they get to the last one and they have to reset and come back to the first one. This is indicated by a variable indicating the ant's position in the loop, between 0 and a max value (*self.loop_lenght*).



```python
def run(self, dt):
    self.loop_position += dt*(self.scheduler.bpm/60)
    if self.loop_position >= self.loop_length:
        self.loop_position -= self.loop_length
        for ant in self.ants:
            ant.reset()

    for ant in self.ants:
        ant.run(dt)
```

This is the basic process followed by composers and musicians, but now we have to see how we include the user playing at the same time. To accomplish this, we define a callback function that first gets the position of the ants in the beat pattern, so as to know where to add the note played by the user. Sampling the beat unit with *ref_points* we can check which division is closest to the moment the person played a particular note and like this we can



```python
def quantize(time, loop_length):
    print("QUANTIZING")
    print(time)
    time_within_beat = time%1
    print("within beat: {}".format(time_within_beat))
    beat = math.floor(time)
    ref_points = [0, 0.3333, 0.5, 0.6666, 1]
    closest = math.inf
    closest_pt = 0
    for pt in ref_points:
        print("looking at point {}".format(pt))
        dist = abs(time_within_beat-pt)
        if dist < closest:
            print("dist: {}".format(dist))
            closest = dist
            closest_pt = pt
    print(closest_pt+beat)
    return closest_pt+beat
```