Experiments in swarm robotics
- Intelligent Robotic Systems -

Giulia Nardicchia
giulia.nardicchia@studio.unibo.it
Dept. of Computer Science and Engineering (DISI), Alma Mater Studiorum Università di Bologna

May 10, 2024

Aggregation behaviour

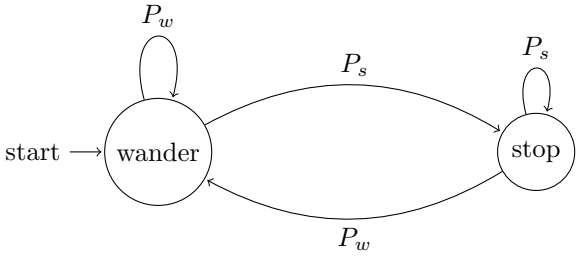
The task for Laboratory Activity 05 is to implement an aggregation behavior for swarm robotics.

Solution description

I implemented the control logic by conceptualizing the problem as a **probabilistic finite state machine** with two states: wander and stop.

- **Wander State**:
  - In this state, robots perform a random walk with collision avoidance.
  - The transition from wander to stop occurs with a probability P_s. This probability P_s depends on the number of nearby robots that are already stopped: the higher this number, the higher P_s, creating *positive feedback*.
- **Stop State**:
  - When in the stop state, robots remain stationary.
  - They can transition back to the wander state with a probability P_w. As the number of stopped nearby robots increases, P_w decreases, leading to a *negative feedback* mechanism in the swarm's dynamics.

You can see the probabilistic finite state automaton in the following image.



Probabilities

The probabilities P_s (probability of stopping) and P_w (probability of leaving the group) are defined as follows:

$$P_s = \min(P_s^{\max}, S + \alpha N) \tag{1}$$

$$P_w = \max(P_w^{\min}, W - \beta N) \tag{2}$$

Where:

- S is the spontaneous stopping probability;
- W is the spontaneous walking probability;
- N is the number of nearby stopped robots;
- α and β are feedback coefficients;
- $P_s^{\max}$ and $P_w^{\min}$ are the maximum and minimum probabilities, respectively.

These parameters ensure that the behavior of the swarm adjusts dynamically based on the density of nearby stopped robots, promoting aggregation.

These parameters in general depend upon the number of robots, the size of the arena and the maximal range used for the range-and-bearing.

Implementation description

The core of the control logic resides in the ‘step()‘ function:

- Initially, the ‘n_steps‘ variable is incremented.
- The function ‘CountRAB‘ is used to calculate the number of nearby robots that have stopped.
- The probabilities P_s and P_w are then calculated as described in 1 and 2, respectively. From these probabilities, the overall probability p is derived. p is the probability of transitioning to a different state and is calculated as the ratio of P_s to the sum of P_s and P_w.
- A random number t is generated using ‘robot.random.uniform()‘. If t is less than or equal to p, the robot transitions to the stop state (‘RobotState.STOP‘) and sets the range-and-bearing data with ‘robot.range_and_bearing.set_data(1,1)‘.
- Otherwise, the robot transitions to the wander state (‘RobotState.WANDER‘) and updates the range-and-bearing data with ‘robot.range_and_bearing.set_data(1,0)‘.
- Actions are executed based on the current state:
  - If the robot's state is WANDER, all LEDs are set to green. Then, the ‘wander_walk()‘ function is called to perform random movement.
  - If the robot's state is STOP, all LEDs are set to red. The ‘stop()‘ function is then called to halt the robot.

To implement the ‘wander_walk()‘ logic, I reused the motor schema mechanism from the previous lab. The following functions manage the robot's movement:

- ‘random_walk()‘: This function generates random movement for the robot. If the number of steps (‘n_steps‘) is a multiple of ‘MOVE_STEPS‘, a random angle between $-\frac{\pi}{5}$ and $\frac{\pi}{5}$ is chosen. It then returns a uniform potential field based on this random angle.
- ‘collision_avoidance()‘: This function handles collision avoidance. It identifies the closest proximity sensor value using the ‘search_highest_value‘ function from the ‘robotUtilities‘ module. It then calculates the translational and angular components of the movement vector based on the detected proximity and a predefined avoidance distance D. Finally, it returns a repulsive field using the calculated translational and angular velocities.
- ‘wander_walk()‘: This function combines random movement with collision avoidance to create the robot's wandering behavior. It first calculates the random movement vector from ‘random_walk()‘ and the collision avoidance vector from ‘collision_avoidance()‘. These vectors are summed to get a resultant vector, to which another uniform potential field is added to ensure consistent movement. The resultant vector is then converted into differential wheel velocities using the ‘converter_from_rototransational_to_differential()‘ function. These velocities are scaled by the maximum velocity and set as the robot's wheel velocities.

To stop the robot, the left and right wheel velocities are simply set to 0.

I used the suggested parameters. For a rectangular arena of 4x5 meters, with 50 robots and a maximal range of 30 cm, the parameters were set as follows: $W = 0.1$, $S = 0.01$, $P_s^{\max} = 0.99$, $P_w^{\min} = 0.005$, $\alpha = 0.1$, $\beta = 0.05$.

Observations

- Parameter values: I experimented with varying the parameter values, but the configurations I tried resulted in worse collective behavior compared to the initial parameters. Choosing the right parameters is crucial as it directly impacts the robot's behavior.
- Debugging: Testing the collective behavior was challenging. Identifying and fixing any code errors required significant time and effort. To visually inspect the overall behavior, I used LEDs to indicate the state of the automaton. I observed the system for a certain number of steps to verify the behavior.